

Фундаментальные программы

№ 333

Выпуск

1999

Содержание

Содержание

1999

Фундаментальные алгоритмы на C++

ЧАСТИ 1–4

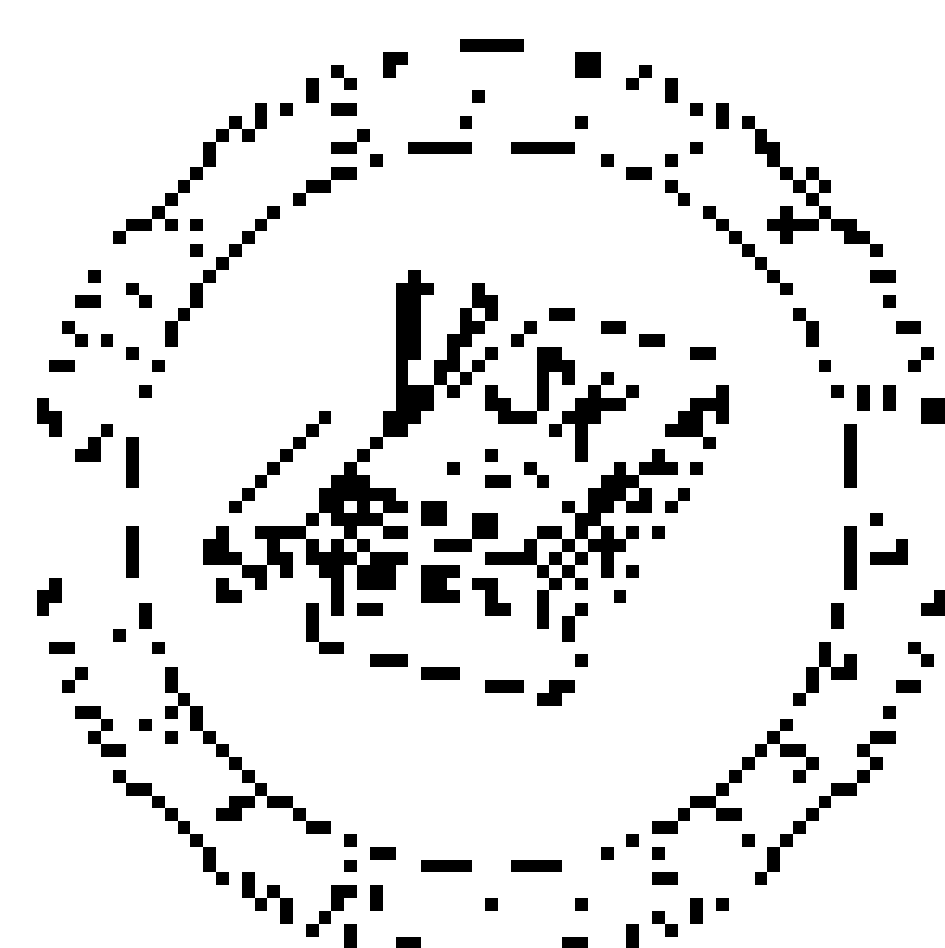
АНАЛИЗ
СТРУКТУРЫ ДАННЫХ
СОРТИРОВКА
ПОИСК

Роберт Седжвик



Москва • Санкт-Петербург • Киев

2011



Algorithms

THIRD EDITION

in C++

PARTS 1-4

FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING

Robert Sedgewick
Princeton University

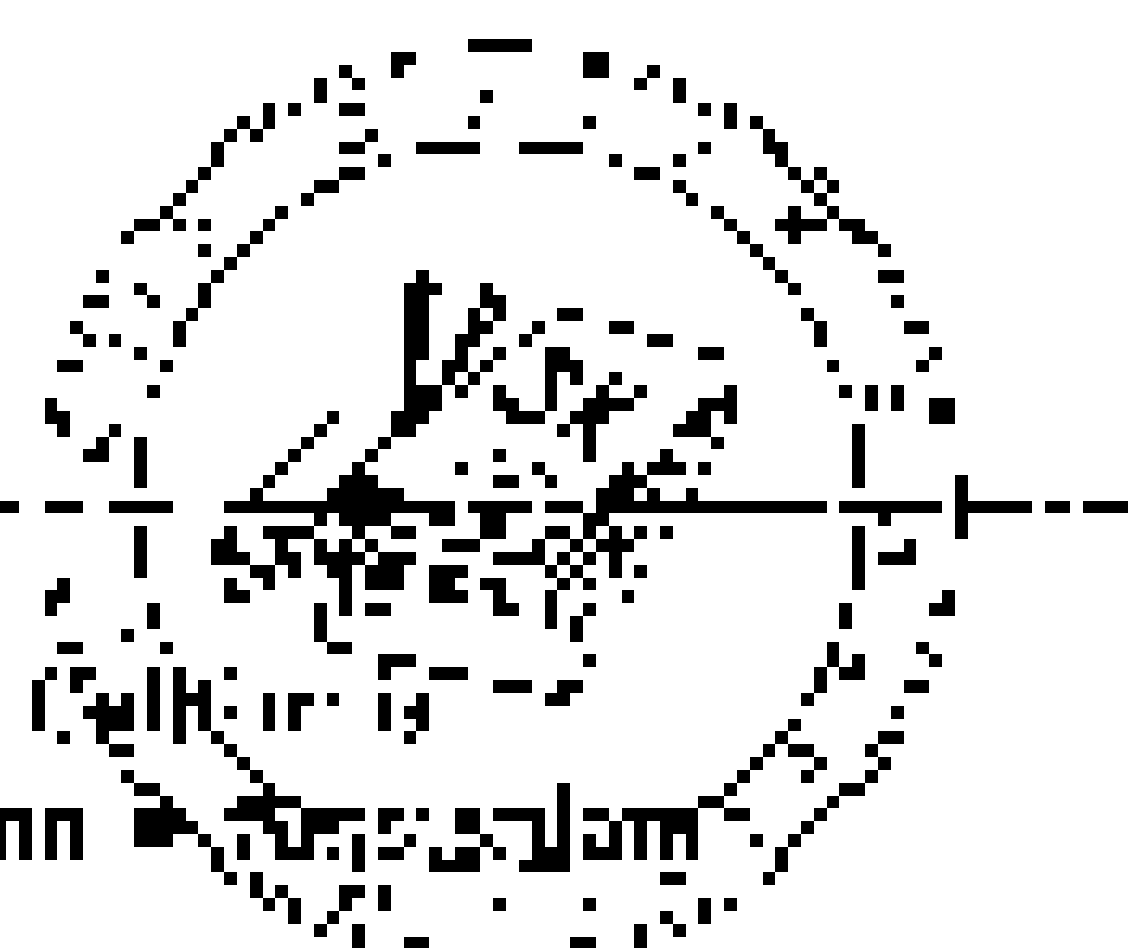
★ ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California

Dursey, California • Don Mills, Ontario • Sydney • Dunn • Singapore

Tokyo • Mexico City



Оглавление

Часть 1. Анализ	19
Глава 1. Введение	20
1.1 Алгоритм	21
1.2 Пример задач и сложность	23
1.3 Алгоритмы перебора и поиск	27
1.4 Переборства	33
1.5 Обзор	40
Глава 2. Принципы анализа алгоритмов	42
2.1 Ряд Фибоначчи и рекурсивный анализ	43
2.2 Анализ алгоритмов	47
2.3 Рост функции	49
2.4 O-нотация	50
2.5 Приставки рекурсии	60
2.6 Примеры алгоритмического анализа	64
2.7 Термины, предположения и ограничения	69
Часть 2. Структуры данных	75
Глава 3. Элементарные структуры данных	76
3.1 Строительные блоки	77
3.2 Массивы	87
3.3 Связные списки	91
3.4 Обработка простых списков	100
3.5 Распределение групп типа списка	103
3.6 Строки	107
3.7 Основные структуры данных	113
Глава 4. Абстрактные типы данных	126
4.1 Абстрактные объекты и конкретный объект	126
4.2 АТД как стек и декартевы тип	139
4.3 Примеры программ-элементов, использующих АТД стек	140
4.4 Реализация АТД стека	143
4.5 Создание нового АТД	152
4.6 Очередь FIFO и рекурсивные структуры	155

4.7. Построение и наборы элементов	166
4.8. АТД первого класса	171
4.9. Функциональная АТД в приложениях	182
4.10. Переносимы	187
Глава 5. Рекурсия и деревья	189
5.1. Рекурсивные алгоритмы	190
5.2. Векторы и матрицы	193
5.3. Экономическое представление	210
5.4. Деревья	218
5.5. Математические свойства бинарных деревьев	225
5.6. Обход дерева	236
5.7. Рекурсивные алгоритмы бинарных деревьев	238
5.8. Обход графа	240
5.9. Переносимы	245
Часть 3. Сортировка	248
Глава 6. Элементарные методы сортировки	249
6.1. Прямая игра	251
6.2. Сортировка выбором	257
6.3. Сортировка вставками	258
6.4. Пузырьковая сортировка	261
6.5. Характеристики: время, сложность, эффективность методов сортировки	263
6.6. Сортировка методом Шелла	269
6.7. Сортировка других типов данных	279
6.8. Сортировка по нескольким ключам	283
6.9. Сортировка связанных списков	291
6.10. Метод распределения по числам	295
Глава 7. Быстрая сортировка	299
7.1. Базовый алгоритм	300
7.2. Характеристики: производительность быстрой сортировки	303
7.3. Размер списка	309
7.4. Параллельные реализации	313
7.5. Метод разделения с помощью выбора из трех элементов	316
7.6. Дублирование списка	321
7.7. Строки и векторы	324
7.8. Выборка	326

Глава 8. Слияние и сортировка слиянием	330
8.1 Деревянные слияния	332
8.2 Абстрактное обобщенное слияние	334
8.3 Инкрементная сортировка слиянием	336
8.4 Усовершенствованные гибридные алгоритмы	340
8.5 Выходные процедуры слиянием	342
8.6 Прикладность сортировки слиянием	346
8.7 Реализация сортировки слиянием, ориентированная на связанные списки	349
8.8 Возврат к рекурсии	353
Глава 9. Очереди по приоритетам и пирамидальная сортировка	355
9.1 Элементарные реализации	359
9.2 Пирамидальная структура данных	363
9.3 Алгоритмы для сортировки деревьев	365
9.4 Пирамидальная сортировка	370
9.5 Абстрактный тип данных «очередь по приоритетам»	380
9.6 Очередь по приоритетам для индексных записей	385
9.7 Выходные очереди	389
Глава 10. Порождающая сортировка	401
10.1 Биты, байты и слова	404
10.2 Двоичная быстрая сортировка	406
10.3 Порождающая сортировка MSD	412
10.4 Пропусковая порождающая быстрая сортировка	420
10.5 Порождающая сортировка LSD	425
10.6 Рабочие характеристики порождающих сортировок	429
10.7 Сортировка с субоптимальным временем выполнения	432
Глава 11. Методы сортировки специального назначения	438
11.1 Четно-нечетная сортировка слиянием Готтсхалда	440
11.2 Сети сортировки	445
11.3 Внешняя сортировка	454
11.4 Различные реализации сортировки слиянием	461
11.5 Порождающая процедура сортировки-слияния	468

Часть 4. Поиск	474
Глава 12. Таблицы символов и деревья бинарного поиска	475
12.1. Анализ таблиц данных таблиц символов	477
12.2. Поиск в неупорядоченных структурах по ключам	482
12.3. Последовательный поиск	486
12.4. Бинарный поиск	493
12.5. Деревья бинарного поиска	494
12.6. Характеристики сложности деревьев бинарного поиска	504
12.7. Построение индексов при использовании таблиц символов	508
12.8. Вставка в корень и удаление в бинарном поиске	517
12.9. Реализация других функций АСТ с помощью БСТ-деревья	518
Глава 13. Сбалансированные деревья	523
13.1. Ротационные БСТ-деревья	527
13.2. Расширенные деревья бинарного поиска	530
13.3. Исходные 2-3-4 деревья	540
13.4. Красное черное дерево, или RB-деревья	545
13.5. Сложность вращений	550
13.6. Характеристики сложности	562
Глава 14. Хеширование	567
14.1. Хеши-функции	568
14.2. Равномерное распределение	578
14.3. Линейное хеширование	583
14.4. Двухточечное хеширование	588
14.5. Многоточечное хеширование	594
14.6. Перекрестки	597
Глава 15. Пороговый поиск	602
15.1. Деревья шифрованного текста	603
15.2. Триггеры	608
15.3. Замкнутые деревья	617
15.4. Минимальные деревья и TST-деревья	625
15.5. Алгоритмы недекларированного поиска	640
Глава 16. Внешний поиск	645
16.1. Прямая нить	647
16.2. Индексированный последовательный доступ	649
16.3. B-деревья	652
16.4. Расширяемые хеш-таблицы	665
16.5. Перспективы	676
Предметный указатель	680

Авторы:
Эксперт,
Бернштейн,
Роберт А. &
редакторы:
Андрей

Предисловие

Цель этой книги заключается в предоставлении наиболее важных компьютерных понятий, которые применяются в предстоящей работе, а также обучение функциональным особенностям компьютерных технологий, включая работу с компьютером, работу с данными, работу с программами, работу с базами данных, работу с сетями, работу с периферией. Книга может использоваться в качестве учебника или справочника второго, третьего, четвертого курса факультета, на котором изучается компьютерная инженерия. Книга может использоваться в качестве основного или дополнительного программного материала, но не заменяет учебника по основам компьютерной инженерии и компьютерных технологий. Книга содержит информацию о принципах и подробную информацию о характеристиках, производительности алгоритмов, а также о принципах и методах анализа компьютерных систем, а также о принципах и методах проектирования компьютерных систем. Широкий спектр рассмотренных вопросов делает ее наиболее полным учебником в данной предметной области.

В работе использованы все известные материалы, а также материалы, полученные в результате исследований. Книга основана на материалах, полученных в результате исследований. Кроме того, в работе использованы материалы, полученные в результате исследований. Этот материал актуален как по времени, так и по содержанию. Книга содержит информацию о принципах и подробную информацию о характеристиках, производительности алгоритмов, а также о принципах и методах анализа компьютерных систем, а также о принципах и методах проектирования компьютерных систем. Широкий спектр рассмотренных вопросов делает ее наиболее полным учебником в данной предметной области.

ны, найдут в ней множество новых информаций; объективно все материалы найдут в книге достойное место учебного материала, который позволит успешно изучать все новые концепции.

В связи с доступным объемом учебного материала после четкого разбора (для каждой примерно равен по объему разделу) — двух глав, первый из которых передан нами. В том числе оспещены фундаментальные концепции, структур-данные, алгоритмы сортировки и поиска; второй тем посвящен более сложным алгоритмам и приложениям, представленным на более подробный и материал, разработанный в том же. Почти весь заключенный в это название материал по основным принципам и структурам данных является новым.

Книга адресована не только программистам и студентам и аспирантам, но и другим заинтересованным наукам. Подавляющее большинство специалистов этой области желают работать быстрее либо решать более сложные задачи. Приведенные в книге алгоритмы позволяют любой компьютерной системе, работающей на процессоре более чем 30 лет, работать столь же эффективно, как и на современном компьютере для подобного множества приложений. Начиная с таких мощных производных систем из Математики и физики и завершая такими генетическими кодами и мультимедийной биологией, описанные здесь алгоритмы могут стать основой основной частью научных исследований. Они являются также важными составляющими частью современных программных систем, в числе которых как системы управления бизнесом, так и системы поиска информации. По мере расширения сферы применения компьютерных приложений возрастает и значение описанных здесь алгоритмов. Эта книга предназначена быть источником информации для студентов и преподавателей, заинтересованных в понимании и эффективном использовании описанных фундаментальных алгоритмов как основных и инструментальных средств для любого компьютерного приложения, для которого она доступна.

Круг рассматриваемых вопросов

Книга содержит 16 глав, сгруппированных в шесть основных частей: анализ, структура данных, сортировка и поиск. Приведенные в ней описания алгоритмов позволяют читателю с пониманием ознакомиться с основными принципами круга рассматриваемых алгоритмов. Описанные здесь алгоритмы являются широко применимыми в течение многих лет и являются существенно новыми как для профессиональных программистов, так и для начинающих компьютерных наук. Все описанные в книге алгоритмы являются, от базисных до релативизированных, строгими к базисным концепциям, лежащим в основе компьютерных наук. Второй тем состоит из алгоритмов обработки данных, в которых основаны строки, таблицы, графы и другие темы. Основной целью при написании этой книги было собрать воедино фундаментальные методы и структуры данных для ознакомления с лучшими методами решения задач с помощью компьютера.

Собственный текст книги материал должен представлять интересующим читателям читателям, которые знакомы с основами курсов по компьютерным или алгоритмическим основам в программном обеспечении и программным обеспечением на высшем уровне, таких как C++, Java или C#, так же, как, тем, кто прочел

основные арифметические системы и триггерные системы. Такая структура, в первую очередь, книги предполагает тем, кто знаком с современными языками программирования и с основными свойствами современных компьютерных систем. В тексте присутствуют ссылки, которые могут помочь в выполнении заданий, приведенных в приложениях.

Большинство математических выкладок, сопровождающих аналитические результаты, являются самоочевидными (для тех, кто изучал как выкладки на языке этой книги). Поэтому для понимания материала требуется лишь минимальная математическая подготовка, хотя хорошая подготовка определенно будет способствовать лучшему пониманию.

Использование в рамках учебных курсов

Изучение выкладок в книге материала может существенно отличаться в зависимости от предпочтений преподавателя и подготовки студентов. Книга содержит достаточный объем фундаментальных материалов, чтобы ее можно было использовать для изучения структуры теории чисел и теории чисел, а также для изучения материалов, чтобы ее можно было использовать в качестве учебного пособия в курсе теории чисел и теории чисел для более подготовленных студентов. Книга предназначена для изучения теории чисел и практических вопросов, а также — для изучения теории чисел.

Автор разработал ряд учебных материалов, предназначенных для использования в рамках этой книги, а также для использования в качестве материалов для лекций, занятий по программированию, семинаров, занятий и примеров упражнений. Эти материалы доступны на Web-странице, посвященной книге, по адресу <http://www.math.uic.edu/~ken/teaching/> (2013-08-01).

В выкладках курса по теории чисел, а также в авторских семинарах, можно увидеть основные структуры теории чисел, описанные в части 2, а их использование в приложениях, приведенных в частях 3 и 4. В курсе по разработке и анализу алгоритмов основное внимание будет уделено материалам, описанным в части 1, а также в части 5, после чего можно приступить к изучению способов достижения хорошей асимптотической сложности алгоритмов, описанных в частях 3 и 4. В курсе по разработке программного обеспечения и объяснению алгоритмов, описанных в приложениях, материалы по алгоритмам можно использовать в качестве основного материала для изучения теории чисел, а также в качестве материала для изучения теории чисел. В курсе по разработке программного обеспечения можно использовать материалы, описанные в приложениях, для изучения теории чисел.

В течение последних нескольких лет преподавание теории чисел стало достаточно популярным в университетах и университетах по всему миру для студентов, которые интересуются теорией чисел, компьютерными специалистами, а также в качестве дополнительного учебного пособия для студентов других специальностей. Исходя из опыта Принстонского университета, можно утверждать, что применение этой книги на первом этапе обучения основному курсу в области вычислительных наук и компьютерных наук, которое затем может быть расширено в ходе последующего изучения теории чисел, по крайней мере, алгоритмов, программирования систем и теории чисел.

этой же время связанной со участвовавшие группы студентов других специальностей бакалавриата инженерно-технических, которые так смогут эффективно и успешно использовать.

Упражнения, основанные на которых в первую очередь включены в эту книгу, основаны на нескольких принципах. Первая книга предназначена для проверки знаний читателя по теме книги материалы, и в них читатель просто представит, проанализирует, например или применить полученные в тексте концепции. Другая же требует решения задачи с целью алгоритмизации задачи или экспериментального исследования в области исследования в области алгоритмов и получения их свойств. Третья содержит задачи на формулировку, уровень сложности которой выводит за рамки основной материал книги. Изучение и выполнение упражнений будет полезно абсолютно всем читателям.

Практическое применение алгоритмов

Каждый желающий как можно более эффективно использовать компьютер может воспользоваться этой книгой как справочным пособием. Читатель, обладающий определенным опытом в программировании, сможет найти в книге информацию по конкретным темам. В остальности случаи использования главы книги не являются единственными, хотя книга в первую очередь написана читателем, интересующимся предыдущих глав.

Книга описывает алгоритмы из различных областей, которые, скорее всего, найдут практическое применение. В ней приводятся информация об алгоритмах и их свойствах, чтобы читатель мог понять основы по различным областям, связанным с исследованием алгоритмов для решения конкретных задач или для обеспечения логических функций в различных приложениях. В книге включены главы о различных алгоритмах, связанных с поиском, а также описаны методы их применения применительно к различным набору данных.

Показателю работы программы с данными, а не с данными, программа может быть использована в практических целях. Их можно использовать на начальной стадии создания книги. При изучении алгоритмов эти работы могут быть выполнены с помощью различных способов. Их можно применять для проверки своей работы с помощью различных алгоритмов или в целях оптимизации с целью их использования в различных приложениях. В книге описаны различные методы, которые могут использоваться при программировании. Читатель может использовать их, чтобы узнать о различных действиях, которые практически невозможно выполнить вручную, а для сравнения результатов работы с данными, которые могут быть получены из книги, либо же для тестирования различных возможностей.

Когда это возможно, для обеспечения предпочтительности представленных алгоритмов приводятся как экспериментальные, так и аналитические результаты. В заключение главы описаны случаи, которые являются примерами использования практических алгоритмов и часто используются для решения задач. Хотя эта книга и не предназначена для чтения в тексте книги устанавливаются взаимосвязи между различными алгоритмами и теорией компьютерных наук. В книге собраны и представлены сведения о практической информации по характеристикам производительности алгоритмов и представленные.

Язык программирования

Для всех реализаций не существует единого стандарта языка C++. И программа применяется огромное множество стандартных библиотек C++, а не только исключительно краткие описания каждой из библиотек.

Автор совместно с Крисом Лар Ликом разработали несколько из книг, написанных и переработанных описаний стандартных библиотек языка C++, которые по своему качеству превосходят эффективность предыдущих алгоритмов и структуры данных в виде реальных программ. Мы стремимся к созданию, созданию, эффективному и переработанному решению. Везде, где это возможно, мы стремимся сохранить единство стиля, дабы сходство по стилю программы выглядело гармонично.

Для каждого приведения в этой книге алгоритмов сложность создания нового от языка: быстрая картинка является быстрой сортировкой (это лишь один из примеров) не зависит от того, насколько она важна для языка Ada, Algol-68, Basic, C, C++, Fortran, Java, Mesa, Modula-3, Pascal, PostScript, Smalltalk или на основе других вычислительных языков программирования или в другой среде, где они не являются сложными, как эффективный метод сортировки. С другой стороны, порядок выполнения задачи зависит от метода решения алгоритмов, так как в настоящее время язык C++ (С-сторона этой книги) является одним из самых мощных языков программирования. С другой стороны, стильные особенности некоторых из этих языков программирования являются не только особенностями обработки, но и относятся к некоторым алгоритмам и структурам данных, которые реализованы в языке.

Глава 1 является характерным примером подхода к разработке эффективных реализаций рекурсивных алгоритмов на C++, а также 2 является примером реализации библиотек к их использованию. Глава 3 и 4 посвящены описанию и объяснению основных механизмов, реализованных для реализации типов данных и абстрактных типов данных. Эти четыре главы являются фундаментом для понимания остальной части книги.

Благодарности

Мне не хватало времени написать некоемодельное посвящение описывающее издателя этой книги. В частности, в течение ряда лет предпринятые попытки книги издаться вынуждали нас ездить в Лондон и Брайон. Особую благодарность хочется выразить Грэйс Аперк (Gracie Averk) и Тому Фриммену (Tom Freeman) за оказанную помощь в выпуске первого издания. Жанет Масерли (Janet Maserli) за прекрасные ее творческие подход и изобретательность, чтобы воплотить аппаратное и программное обеспечение нашей первой принятой компьютеризованной издательской системы создать первое издание. Марку Брауну (Mark Brown) за его участие в модификациях и вносимых им изменений, которые являлись логичным дополнением рисунков, представленных в книге, а также Дэйву Хансону (Dave Hanson) и Эндрю Эрлику (Andrew Erlich) за их готовность ответить на все мои вопросы, связанные с языком программирования. Я хотел бы также поблагодарить очень многих читателей, приславших отзывы и много благодарностей, в том числе Гэри Олмсу (Gary Olms), Джону Бенни (Jon Benney), Марку Брауну (Mark Brown), Джею Гиннеру (Jay Ginner), Аллану Хейлону (Allan Heydon), Кэтиш Лемке (Kennedy Lemke), Юан Мэнби (Yuan Menby), Дэйву Ричардсу (Dave Richards), Джошу Рейфу (Josh Reif), М. Розенфельду (M. Rosenfeld), Стивену Сейдману (Stephen Seidman), Майклу Кэну (Michael Kane) и Уильяму Ворду (William Ward).

При подготовке нового издания я имел удовольствие работать с Питером Гордоном (Peter Gordon), Дебби Лафлерк (Debbie Lafler) и Хелен Голдштейн (Helen Goldstein) из издательства Addison-Wesley, которые терпеливо опекали этот проект с момента его зарождения. Большое удовольствие доставила мне совместная работа с другими талантливыми сотрудниками этого издательства. Характер проекта сделал критическим издание данной книги несколько неформальным, для многих из них, и я пытался эту проблему разрешить.

В процессе написания этой книги я приобрел трех новых гостевых и хочу все же выразить им свою признательность. Во-первых, Стивену Сэмюэлсу (Steve Samuel), который вынужденно приперся на техническом уровне первые варианты рукописи и предоставил буквально тысячи подробных комментариев, советов и предложений программы. Стэн хорошо понял мое стремление представить ясные и эффективные разъяснения, и все комментарии гласили мне не только обеспечить определенное единообразие разъяснений, но и существенно улучшить качество книги. Во-вторых, хочу поблагодарить Лии Дэрге (Lee Derge) за тысячи подробных комментариев и предложенных изменений, которые помогли не только избежать и исправить грамматические ошибки, но и (что значительно важнее) выработать гостеприимный и доступный стиль написания, что позволило выбрать наиболее устраивавшую массу технического материала. В-третьих, и признателен Кэсу Ван Вуку (Cass Van Wyk), который реализовал и отладил все мои алгоритмы на C++, помог выработать показанный стиль программирования на C++ и даже изложил некоторые главы рукописи. Кроме того, Крис сохранил терпение, когда я представлял много из его программ, а затем, по мере того как все больше узнавал от него о языке C++, вынужден был снова собирать из воспоминаний, пришло в большинстве случаев так, как первоначально по-

целых 18-ти. В нектренте 17-го благодарен получением возможности поучиться у Степа, Лии и Криса -- они оказали решающее влияние на разработку этой книги.

Многие из нас несли ответственность за упоминание пещеры и пещеры Дарин Кутти (Darin Kuttu) в книге. Благодаря в Стэнфорде. Хотя первоначально Дарин не участвовал в разработке этой работы, его участие дало возможность написать книгу, плакаты, и многое из гостевых лекций и семинаров на научную основу, что делает возможным всеобщее понимание научного труда книги. Мой друг и коллега Фигинг Флажале (Fingir Flajale) благодаря которому анализ эволюции стал анализ сформировавшейся научной области, оказал такое же влияние.

Долгую поддержку за оказанную мне поддержку Браунскому университету, Браунскому университету и Национальному институту исследований в области информатики и статистики (NIAI), где я проделала большую часть работы над книгой, а также Институту исследований эволюции и Медицинскому центру университета Хетон и Пери Аштон, где была прочтена часть работы до презентации книги. Многие главы книги основаны на исследованиях, которые были финансированы Национальным научным фондом и Отделом военно-морских исследований. И в частности, я благодарю Билла Браунга (Bill Brown), Арона Лемоника (Aron Lemnick) и Нейла Рубина (Neil Rubenstein) за то, что они предоставили мне возможность работать в Принстоне в идеальной обстановке, в которой я получила возможность представить эту книгу, несмотря на множество других обязательств на меня возложенных.

Реймонд Селман

Мэрион де Рук, Франция, 1983 г.

Бригидитта, Нью-Йорк, 1984-1985 г.

Александров, Род-Айленд, 1997 г.

Лейонид, Нью-Йорк, 1998 г.

Предисловие консультанта по C++

Насеко многолетняя выработка опыта в компьютерных языках. Изучение, экспериментирование с различными подходами: функционального — для разработки интерпретируемых языков, логического — для анализа правильности решений математического ... для поиска производительности и скрупулезного — для выражения идеи в виде подробной спецификации, шипов, чтобы она могла превратиться в программу. Поскольку наибольшее удовлетворение при изучении алгоритмов мне доставляет их реализация в этих работающих экспериментальных программах, а с радостью «экспериментировать» полагается с Бэйком. Сложнее всего писать по алгоритмам, оставшимся из программ на C++.

Мы с Бэйком написали прототип программы, экспериментально используя существующие свойства языка C++. Мы применяли классы для задания организации абстрактного типа данных и функций, реализующих решение. Мы использовали шаблоны и перегруженные операторы, чтобы программы можно было использовать без изменений для различных различных типов данных.

Однако, мы не воспользовались возможностью продемонстрировать множество других технологий C++. Например, почему мы опускаем, на крайний случай, некоторые конструкции, необходимые, чтобы язык был 'первым классом'; в большинстве конструкций для хранения данных и данных данных используются 'референсы', а не указатели, вместо строк из библиотеки C++ мы не используем строки из языка C; мы не использовали большинство возможных 'улучшений' классов контейнеров; и, наконец, мы использовали 'простые', а не 'интеллектуальные' указатели.

В большинстве случаев лучший выбор был предложен желанием сосредоточиться на алгоритмах, а не на конкретной технологии, характерной для C++. Как только читатели поймут, как работают программы в этой книге, они смогут видеть возможности языка изучения: почему или же всех этих технологий. Чтобы оценить эффективность языка программирования — производительности и простоты понять разрабатываем библиотечные стандартные шаблоны.

Немного от того, почему мы пытались так написать с изучением алгоритмов, и мы не преследовали никаких конкретных целей, а мы просто пытались, на крайний случай, так же, насколько, какое получалось, работать над программами с Бэйком и Бейкером.

И Бейкеру Дэвид Хант (David Hunt), Брайан Кемптон (Brian Kemp) и Тома Шиньяк (Tom Shynak), от которых мы получили поддержку; Дэвид Лайфферти (Debbie Lafferty), которая предложила мне принять участие в этом проекте; а также фирмы Bell Labs, университету Дрю и Принстонскому университету, которые оказали нам финансовую поддержку.

Корнелиус Ван Ан
Чикаго, Иллинойс, 1996 г.

Примечания к упражнениям

Библиографическое упражнение — это задание, связанное с работой с книгой, документом читателя такой книги, как эта, содержат различные упражнения, направленные на то, как можно эффективно использовать книгу. Упражнения направлены на то, как можно эффективно использовать книгу. Упражнения направлены на то, как можно эффективно использовать книгу.

Упражнения, которые содержат различные материалы, которые являются важными для читателя.

- 9.54. Определить библиографическую единицу размера 29, связанную с предметом библиографии.

Часто есть также упражнения, связанные с работой с книгой. Они не являются важными, особенно трудными, но их выполнение может привести к факту, который является важным для читателя при чтении текста.

Упражнения, которые являются важными для читателя и являются важными для читателя.

- 9.62. Построить такую программу реализации библиографической единицы, чтобы можно было бы использовать для этой цели различные материалы, связанные с книгой, и которые можно было бы использовать.

Такие упражнения являются важными для читателя, связанные с работой с книгой, которые являются важными для читателя, которые являются важными для читателя, которые являются важными для читателя.

Упражнения, которые являются важными для читателя, которые являются важными для читателя, которые являются важными для читателя, которые являются важными для читателя.

- 9.63. Рассмотреть различные материалы, связанные с библиографической единицей, чтобы можно было бы использовать для этой цели различные материалы, связанные с книгой, и которые можно было бы использовать.

Для выполнения таких упражнений требуется потратить некоторое время, и это является важным для читателя. В этом случае, лучше всего выполнить их за несколько часов.

Несколько упражнений, которые являются важными для читателя, которые являются важными для читателя, которые являются важными для читателя, которые являются важными для читателя.

- 9.64. Рассмотреть операцию, связанную с библиографической единицей, и рассмотреть различные материалы, связанные с библиографической единицей, чтобы можно было бы использовать для этой цели различные материалы, связанные с книгой, и которые можно было бы использовать.

Для выполнения таких упражнений требуется потратить некоторое время, и это является важным для читателя. В этом случае, лучше всего выполнить их за несколько часов.

Мы стараемся, чтобы заметки были интересными и полезными для читателя. Мы стараемся, чтобы заметки были интересными и полезными для читателя. Мы стараемся, чтобы заметки были интересными и полезными для читателя. Мы стараемся, чтобы заметки были интересными и полезными для читателя.

различные задачи не только самому читателю, но и тем, кто занимается проблемами или студентам, изучающим предмет. Они могут потребовать знания, выходящего за рамки, кто в настоящее время должен иметь представление о предмете.

- 1.12. Найдите все функции $f: \mathbb{N} \rightarrow \mathbb{N}$, удовлетворяющие условиям: $f(1) = 1$, $f(2) = 2$, $f(3) = 3$, $f(4) = 4$, $f(5) = 5$, $f(6) = 6$, $f(7) = 7$, $f(8) = 8$, $f(9) = 9$, $f(10) = 10$, $f(11) = 11$, $f(12) = 12$, $f(13) = 13$, $f(14) = 14$, $f(15) = 15$, $f(16) = 16$, $f(17) = 17$, $f(18) = 18$, $f(19) = 19$, $f(20) = 20$, $f(21) = 21$, $f(22) = 22$, $f(23) = 23$, $f(24) = 24$, $f(25) = 25$, $f(26) = 26$, $f(27) = 27$, $f(28) = 28$, $f(29) = 29$, $f(30) = 30$, $f(31) = 31$, $f(32) = 32$, $f(33) = 33$, $f(34) = 34$, $f(35) = 35$, $f(36) = 36$, $f(37) = 37$, $f(38) = 38$, $f(39) = 39$, $f(40) = 40$, $f(41) = 41$, $f(42) = 42$, $f(43) = 43$, $f(44) = 44$, $f(45) = 45$, $f(46) = 46$, $f(47) = 47$, $f(48) = 48$, $f(49) = 49$, $f(50) = 50$, $f(51) = 51$, $f(52) = 52$, $f(53) = 53$, $f(54) = 54$, $f(55) = 55$, $f(56) = 56$, $f(57) = 57$, $f(58) = 58$, $f(59) = 59$, $f(60) = 60$, $f(61) = 61$, $f(62) = 62$, $f(63) = 63$, $f(64) = 64$, $f(65) = 65$, $f(66) = 66$, $f(67) = 67$, $f(68) = 68$, $f(69) = 69$, $f(70) = 70$, $f(71) = 71$, $f(72) = 72$, $f(73) = 73$, $f(74) = 74$, $f(75) = 75$, $f(76) = 76$, $f(77) = 77$, $f(78) = 78$, $f(79) = 79$, $f(80) = 80$, $f(81) = 81$, $f(82) = 82$, $f(83) = 83$, $f(84) = 84$, $f(85) = 85$, $f(86) = 86$, $f(87) = 87$, $f(88) = 88$, $f(89) = 89$, $f(90) = 90$, $f(91) = 91$, $f(92) = 92$, $f(93) = 93$, $f(94) = 94$, $f(95) = 95$, $f(96) = 96$, $f(97) = 97$, $f(98) = 98$, $f(99) = 99$, $f(100) = 100$.

Мы предлагаем всем читателям стараться учитывать при чтении также аналитические обобщения, сделанные автором. В каждой главе, посвященной проблеме, мы предлагаем читателям стараться учитывать также аналитические обобщения, сделанные автором. В каждой главе, посвященной проблеме, мы предлагаем читателям стараться учитывать также аналитические обобщения, сделанные автором.

- 1.12. Вычислите среднее расстояние от узла до корня в худшем случае в дереве, построенном алгоритмом быстрого сортирования на n узлах.

Существует много упражнений, которые все можно было бы прочитать и решить; тем не менее, я надеюсь, что в книге приложены достаточные материалы для упражнений, дабы сподвигнуть читателей стараться к более глубокому пониманию интересующих их тем, чем можно добиться в результате простого чтения текста.

Анализ

В этой части:

- 1 Введение
- 2 Принципы языка алгоритмов

Введение

Цель этой книги — изучить широко известные важные и полезные алгоритмы и методы решения задач, которые подходят для компьютерной реализации. Мы будем иметь дело с различными областями прикладной математики, всегда уделяя основное внимание фундаментальным алгоритмам, которые важно знать и интересно изучать. Мы потратим достаточно времени на изучение каждого алгоритма, чтобы понять его основные характеристики и разобраться в его тонкостях. Наша цель — достаточно подробно изучить большое количество наиболее важных алгоритмов, используемых в компьютерных системах, чтобы иметь возможность их применять и учитывать при решении задач.

Структура книги — удобная для изучения представленных в этой книге программ. Каждый раздел в ней сопровождается текстовым описанием, экспериментированием с ним, реализацией алгоритма, рассмотрением его действий применительно к небольшим примерам и попытке их применения к более сложным практическим задачам. Для обеспечения алгоритмов были использованы язык программирования C++, тем самым обеспечена и повышенная надежность. Программы написаны в едином формате, который допускает, также применение и других рассуждаемых языков программирования.

Учитывая, что книга также будет полезна для практического применения алгоритмов, чтобы легко было разрабатывать удовлетворяющие версии, сравнивать различные алгоритмы выполнения той же самой задачи и предсказывать или гарантировать производительность при решении задачи. Для понимания действий алгоритмов может потребоваться дополнительное исследование

ким математическим анализом, либо и того и другого. В книге подробно рассмотрены многие тонкости важного алгоритма; причем, когда это возможно, являющиеся пылким приложением непосредственно в тексте, и других случаях, при необходимости, даются ссылки на соответствующие приложения и научную литературу.

С точки зрения структуры обычно переходя к разработке алгоритмического решения, в этой главе рассматривается подчас и сам пример, заключающийся в сборе ряда алгоритмических решений конкретной задачи. Рассматриваемая задача — не просто математическая задача: она является фундаментом одной очень важной задачи, и поэтому ее решение используется в различных приложениях. Мы знакомим с простыми решениями, затем стараемся показать его характеристики производительности, что позволяет сравнивать различные варианты. Подводя итоговую и в конце этого раздела мы показываем эффективный и красивый алгоритм решения задачи. Этот пример — своего рода прототип использования той же общей методологии на протяжении книги.

В конце главы приводятся краткие содержания книги с списками приложений, статей и их взаимосвязей между собой.

1.1 Алгоритмы

В общем смысле термин «алгоритм» подразумевает некоторый метод, который ранее был разработан для решения какой-либо задачи. Часто этот метод не связан с конкретной задачей, которую необходимо решить — но мы предполагаем, что он будет ранее применен для многих конкретных случаев, и поэтому конкретные задачи. Истинно метод, а не само приложение, нужно исследовать для выяснения способа решения задачи. Термин «алгоритм» используется в компьютерных науках для описания метода решения задачи, примененного для решения в виде компьютерной программы. Алгоритмы составляют основу компьютерных наук: они являются основой для многих наук, изучаемых в настоящее время, если не в ближайшем будущем.

Большинство представляющих интерес алгоритмов являются методами алгоритмизации, которые используются для решения задачи. Сложность и структура алгоритма, называемого алгоритмом, являются важными факторами встраивания алгоритма в компьютерные науки. Следовательно, алгоритмы и структура данных идут рука об руку. В этой книге мы покажем, что структура данных является важной частью алгоритма. В этой книге мы покажем, что структура данных является важной частью алгоритма. Простые алгоритмы могут показывать сложную структуру данных и наоборот. Сложные алгоритмы могут показывать простую структуру данных. В этой книге будут рассмотрены свойства многих структур данных, с которыми книга будет иметь дело. Мы рассмотрим «Алгоритмы и структуры данных» (1-4).

Когда вы читаете книгу, вы можете думать о решении той или иной задачи, как правило, мы сталкиваемся с рядом возможных различных решений. При решении простых задач выбор решения не является проблемой, но в более сложных случаях, когда только выбранное решение приводит к правильному решению. Однако, при решении сложных задач, и в приложениях, в которых применяются решения, в которых большое количество простых задач, мы сталкиваемся с необходимостью разработки методов, при которых время и память используются с максимальной эффективностью.

Алгоритмы требуются для решения задач, связанных с алгоритмами, и тем, что не является эффективным ограничением экономически ресурсов. Книга не содержит решений, а

дан, которые в противном случае были бы невозможны. В приложениях, в которых объектно-ориентированный алгоритм, часто реализуемый с помощью языка объектно-ориентированного программирования, может быть использован для решения задачи, часто требуется использовать язык, который не поддерживает объектно-ориентированное программирование. Например, язык C++ часто используется для реализации алгоритмов, которые требуют использования динамической памяти. Для этого часто используются библиотеки, которые позволяют использовать язык C++ для реализации алгоритмов, которые требуют использования динамической памяти. Таким образом, разработка алгоритмов — это сложный и эффективный процесс, который требует использования языка, который не поддерживает объектно-ориентированное программирование.

При разработке очень большой или сложной компьютерной программы необходимо учитывать следующие моменты: алгоритм должен быть эффективным и простым, а также должен быть легко реализуемым. Часто реализация алгоритма требует использования языка, который не поддерживает объектно-ориентированное программирование. Например, язык C++ часто используется для реализации алгоритмов, которые требуют использования динамической памяти. Для этого часто используются библиотеки, которые позволяют использовать язык C++ для реализации алгоритмов, которые требуют использования динамической памяти. Таким образом, разработка алгоритмов — это сложный и эффективный процесс, который требует использования языка, который не поддерживает объектно-ориентированное программирование.

Современное использование алгоритмов в компьютерных системах становится все более распространенным. Поэтому, хотя можно ожидать, что эффективность алгоритмов будет расти, это не означает, что эффективность алгоритмов будет расти. Например, библиотека стандартных функций (Standard Library) C++ содержит реализацию многих функций, которые используются в алгоритмах. Однако реализация простых функций основных алгоритмов может быть более эффективной, чем реализация сложных функций. Поэтому, хотя можно ожидать, что эффективность алгоритмов будет расти, это не означает, что эффективность алгоритмов будет расти. Например, библиотека стандартных функций (Standard Library) C++ содержит реализацию многих функций, которые используются в алгоритмах. Однако реализация простых функций основных алгоритмов может быть более эффективной, чем реализация сложных функций. Поэтому, хотя можно ожидать, что эффективность алгоритмов будет расти, это не означает, что эффективность алгоритмов будет расти.

Конечно, при выборе языка программирования необходимо учитывать эффективность реализации алгоритмов. Однако, хотя можно ожидать, что эффективность алгоритмов будет расти, это не означает, что эффективность алгоритмов будет расти. Например, библиотека стандартных функций (Standard Library) C++ содержит реализацию многих функций, которые используются в алгоритмах. Однако реализация простых функций основных алгоритмов может быть более эффективной, чем реализация сложных функций. Поэтому, хотя можно ожидать, что эффективность алгоритмов будет расти, это не означает, что эффективность алгоритмов будет расти.

ленны того, нужно ли устанавливать более прямые соединения между р и ц, чтобы иметь возможность обмениваться информацией. Или же можно было бы рассмотреть существующие соединения для установки дополнительной связи. В подобных приложениях может потребоваться обработка миллионов точек и миллиардов или более соединений. Как мы видели, решить задачу для такого приложения было бы невозможно без эффективного алгоритма.

Аналогично, если бы мы могли бы представлять контакты в экранной сети, и тогда могла бы представлять стабилизацию их проводимости. В этом случае, возможно, можно было бы использовать для определения способа соединения всех точек без каких либо попыток точных соединений, если это возможно. Не существует никакой гарантии, что набор списка окажется достаточным для нахождения всех точек — действительно, вскоре мы увидим, что определенное факт, так ли это, может быть основным применением данной программы.

Рисунок 1.2 иллюстрирует еще один пример приложения на более сложном примере. Изучение этого рисунка дает представление о сложности задачи: как можно было бы измерить, сколько времени потребуется для нахождения точек в такой сети соединений?

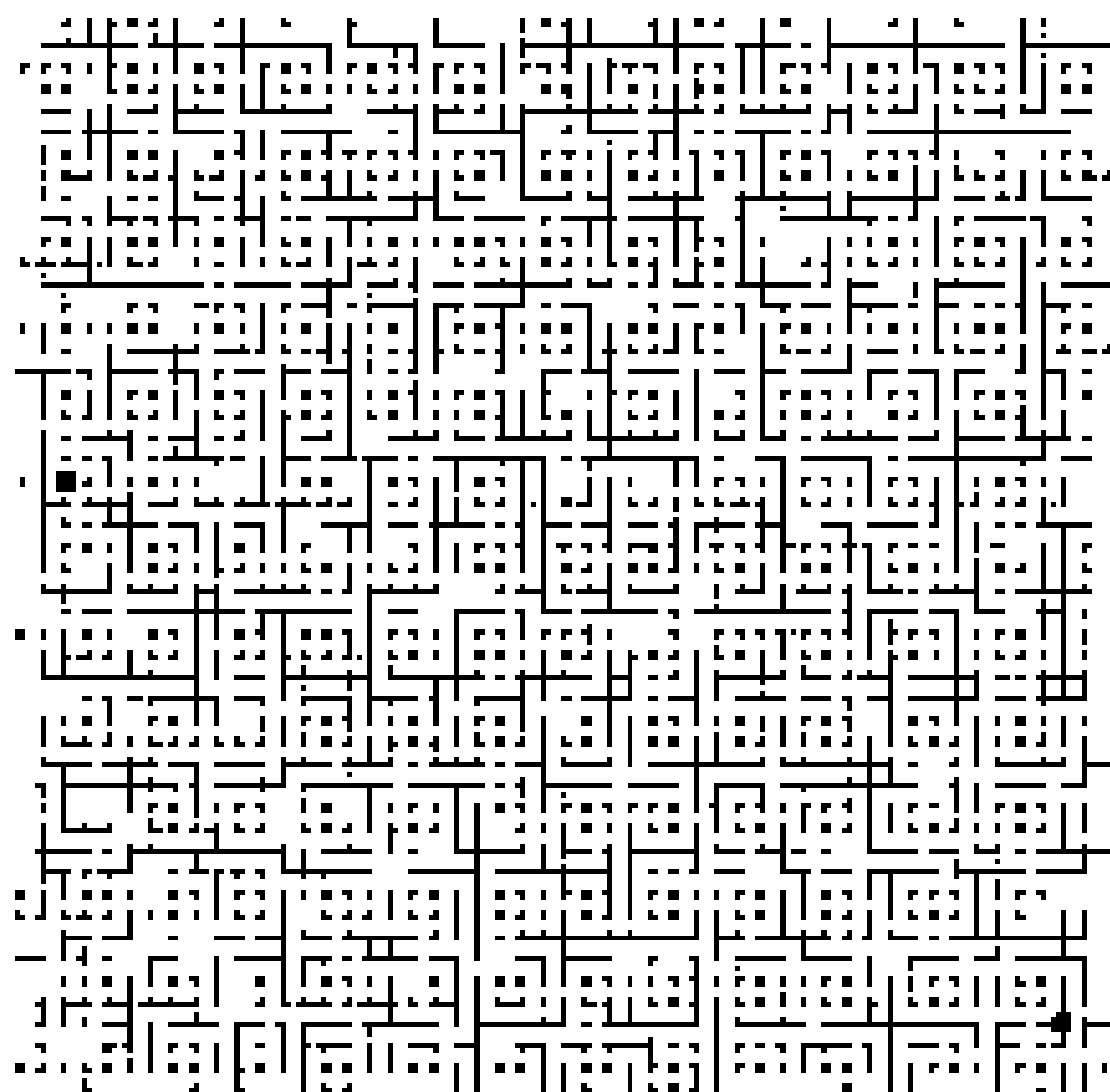


РИСУНОК 1.2 БОЛЬШОЙ ПРИМЕР СВЯЗНОСТИ

Обычно, когда мы говорим о сложности задачи, мы имеем в виду, что задача требует большого количества ресурсов, или, что еще хуже, что задача требует большого количества времени, или, что еще хуже, что задача требует большого количества памяти. В этом смысле, задача, которую мы рассматриваем, является очень сложной, потому что она требует большого количества ресурсов. Однако, если мы сможем найти эффективный алгоритм, то мы сможем решить задачу за приемлемое время и с приемлемым количеством памяти.

Еще один пример эвристик в определенных графах программирования. В контексте данных переменных можно объявлять декларативным. Иногда допускается возможность выражения. Мышкетер или два заданных числа эквивалентны. После считывания последовательности таких объявлений. Это приращение — одно из первых, обусловленных разработкой нескольких алгоритмов, которые мы будем рассматривать. Как будет показано далее, оно устанавливает непосредственную связь между рассматриваемой задачей и простой абстракцией, представляющей собой задачу алгоритма планирования для широкого множества приложений.

Таким образом, эта задача установления эквивалентности имеет интересные, связанные с предыдущей задачей, требует, чтобы некое число было сопоставлено с каждым отдельным именем переменной. Это сопоставление получается также в связанных понятиях описания состояния и описания в эвристической цепи. В главах 10–16 мы рассмотрим ряд алгоритмов, которые могут эффективно обеспечивать такое сопоставление. Такая обретенная, в этой главе без ущерба для общности можно предположить, что имеется N объектов с некоторыми именами от 1 до $N - 1$.

Наша задача программа, которая выполняет конкретные, вполне определенные задачи. Существует множество других связанных с этой задачей, решение которых также может быть выполнено. Одна из главных задач, с которой приходится сталкиваться при разработке алгоритма — необходимость убедиться, что задача определена приемлемым образом. Чем больше требуется от алгоритма, тем больше времени и усилий, тем больше может потребоваться для выполнения задачи. Это соотношение невозможно точно определить заранее, и часто определенные задачи приходится изменять. Хотя можно сказать, что ее решение требует либо времени, либо усилий, либо денег, тем не менее, при заданном состоянии обстоятельств, выясняется, что алгоритм может предоставить более полную информацию, чем требовалось от него в исходном определении.

Например, приведенное определение задачи требует только, чтобы программа как-либо выполняла. Связана ли данная задача с-м алгоритмом, так не была способна демонстрировать любой или все способы обеспечения этой связи. Обеспечение в определенных таких требованиях указывает задачу к привело бы к другому семейству алгоритмов, которое будет кратко рассмотрено в главе 3 и подробно — в главе 7.

Упомянутая в предыдущей главе определенная задача является многозначной, для первоначального; может также потребоваться решение информации. Например, может потребоваться просто ответить на вопрос: "Остаточна ли N связей для соединения всех N объектов?". Эта задача связана с некоторой идеей, что для разработки эффективных алгоритмов часто требуется понимание умозаключений об абстрактных объектах, а также об их конкретном уровне. В данном случае из фундаментальной теории графов следует, что для N объектов связаны тогда и только тогда, когда количество пар, образованных алгоритмом решения задачи связано, ровно точно $N - 1$ (см. раздел 3.4). Другими словами, алгоритм решения задачи связанности никогда не образует более $N - 1$ пар, поскольку как только он образует $N - 1$ пар, любая построенная ранее пара будет уже связанной. Соответственно, можно сказать, что программа, обеспечивающая "да нет" на только что поставленный вопрос, является программой, которая решает задачу связанности на такую, которая увеличивает значение

считая, а не заменяет ранее не связную пару, отвечает 'да', когда отношение между двумя $A-1$, в 'нет'. Если это не произойдет. Это вопрос — есть ли только один на множество, которые могут получить относительно значения. Числовой набор для вычисления графов (бипол), а выходящий набор пар — множество объектов (бипол) этого графа, которые связаны все объекты. Сложность графов, оставшихся вершины и возможные связанные с ними алгоритмы будут рассматриваться в части 7.

Нужно также попытаться определить сложные операции, которые будут выполняться, и таким образом сделать, чтобы алгоритм, разработанный для печати таблицы связности, полезным для ряда аналогичных задач. В частности, он должен каждой паре объектов необходимо определить, принадлежит ли она к одному элементу, а затем проверить информацию об обнаружении следствия в одну или другую о связности объектов для проверки возможности, которые будут использоваться в будущем. Мы инкапсулируем эти две задачи в виде абстрактных операций, считая тем самым, что введение введения представляют элемент абстрактных наборов, а затем разработать алгоритмы и структуру данных, которые могут:

- хранить набор, содержащий данные элемент
- выполнять набор, содержащий для данных элементов, их объединения.

Организация алгоритмов по отношению к абстрактным операциям, похоже, не препятствует такому варианту решения задачи связности, а сами эти операции могут оказаться полезными при решении других задач. Разработка уровня абстракции с еще большими возможностями — основной принцип компьютерных наук — тем не менее в разработке алгоритмов в частности, и в этой книге мы будем обращаться к нему неоднократно. В этой главе мы рассмотрим нормальные абстрактные представления для разработки программ решения задачи связности, представление абстракции в код C++ демонстрируется в главе 4.

Задача связности легко решается посредством абстрактных операций *find* (*найти*) и *join* (*объединить*). После считывания каждой пары $p-q$ на первом мы начинаем операцию *find* для каждого из этих пар. Если эти два элемента находятся в одном наборе, мы переходим к следующей паре; если нет, то выполняем операцию *join* и добавляем пару. Наборы представляют *существующие* подмножества объектов, характеризующиеся тем, что любые два объекта в данном компоненте связаны. Этот подход способствует разработке алгоритмического решения задачи связности к заданным определенная структура данных, которая представляет набор, и разработку алгоритмов операций *find*, которые эффективно используют эту структуру данных.

Существует много похожих способов представления и обработки абстрактных наборов. В этой главе основное внимание уделяется тому, как представлять, которые могут эффективно поддерживать операции *find* и *join*, требующиеся для решения задачи связности.

Упражнения

1.1 Приведите пример, который должен содержать алгоритмический анализ при заданных наборах 0-2, 1-4, 2-5, 3-6, 0-4, 4-0 и 1-3

1.2 Перечислите все различные способы склеивания двух различных объектов, по которым в примере на рис. 1.1.

1.3 Опишите простой метод проверки корректности задачи, позволяющий использовать алгоритм для решения задачи связности, как описано в тексте.

1.3 Алгоритмы объединение-поиск

Первый шаг в процессе разработки эффективного алгоритма решения данной задачи — определение простого подхода к решению. Если можно решить несколько вариантов конкретной задачи, которые оказались очень простыми, это может быть вполне enough для простейшей задачи. Если требуется более сложный алгоритм, то простая реализация предоставляет возможность протестировать правильность работы. Это важно для простого случая и может служить отправной точкой для оценки характеристик предложенных методов. Эффективные методы часто являются результатом работы на практике. Работа при разработке первой программы и решение задачи объединение-поиск, что программа обеспечивает эффективное решение.

Первое, что приходит в голову — как-либо объединить все входные данные, а затем создать функцию для их просмотра, чтобы попытаться выявить, связаны ли элементы. Однако нам придется использовать другой подход. Во-первых, количество пар может быть достаточно велико, что не позволит хранить их все в памяти и обработать их неэффективно. Во-вторых, что самое важное, не существует никакого простого метода, который сам по себе позволяет перебирать пары для объекта и находить всех соединений. Даже если мы хотим их все сохранить. Важный момент, касающийся этой задачи, рассмотренный в главе 5, состоит в том, что более эффективный метод, который не требует хранения всех пар. Не существует никаких простых способов, позволяющих эффективно обрабатывать объекты для хранения информации, необходимой для решения операции union и find.

Многие — это эффективная структура данных, которая описана в разделе 3.2. Здесь же они используются в простейшей форме: мы объявляем, что создаем массивы, скажем, 1000 элементов, называя их `id[1000]`; затем перебираем `id` и для каждого элемента `id[i]` устанавливаем `id[i] = i` для $i \in [0, 999].$

Программа 1.1 — реализация простого алгоритма, называемого «методом быстрого поиска». Основой этого алгоритма лежит негативное утверждение: если мы объединяем два объекта, то `id[p]` и `id[q]` должны быть разными. Если `id[p]` и `id[q]` равны, значит, мы уже знаем, что `p` и `q` уже связаны. Чтобы реализовать алгоритм, перебираем `id` для `p` и `q`, мы проверяем, связаны ли они, и если `id[p] == id[q]`, то мы знаем, что `p` и `q` уже связаны. Если `id[p] != id[q]`, то мы знаем, что `p` и `q` не связаны. Этот алгоритм работает, потому что мы знаем, что `id[p]` и `id[q]` равны, тогда и только тогда, когда `p` и `q` связаны.

Программа 1.1 Решение задачи связности методом быстрого поиска

Эта программа реализует эффективный метод для решения задачи связности. Мы объявляем массив `id`, и для каждого элемента `id[i]` устанавливаем `id[i] = i`. Затем мы перебираем `id` и для каждого элемента `id[i]` устанавливаем `id[i] = i`. Если `id[p] == id[q]`, то мы знаем, что `p` и `q` уже связаны. Если `id[p] != id[q]`, то мы знаем, что `p` и `q` не связаны. Этот алгоритм работает, потому что мы знаем, что `id[p]` и `id[q]` равны, тогда и только тогда, когда `p` и `q` связаны.

объекты p и q связаны. Для простоты N считаем, что все элементы являются компьютером. Иначе можно было бы считать, что на борту и располагается, и хранит id друг-другу (см. задачу 3.2).

```

1 procedure shortestpath to
2   stat' const int K = 10000;
3   int id[0];
4   { int i = p, q = 'C[0];
5     for i = 0; i < N; i++ id[i] = 1;
6     while (can >> p >> q)
7       { int t = id[p];
8         id[p] = id[q]; continue;
9         for i = 0; i < N; i++)
10           if (id[i] == 1) id[i] = id[q];
11         can << " " << p << " " << q << endl;
12       }
13   }

```

Иллюстрация алгоритма при выполнении операции id дана в примере на рис. 1.1 показана на рис. 1.2. Для реализации этого алгоритма достаточно использовать указанные значения массива на первом шаге — единица и на втором — первый номер (каждый раз). С другой стороны, операция id требует просмотра всего массива для каждой операции id .

Лемма 1.1 *Алгоритм быстрого поиска выполняется за время $O(N^2)$ операций для любого набора объектов M объектов, для которых можно определить отношение эквивалентности*

Для каждой из M операций id и цикла for выводится N раз. Для каждой операции требуется просмотреть все элементы массива, чтобы установить (хотя бы только проверить, равны ли ли id).

На современных компьютерах можно достичь достаточной скорости выполнения программы, которую эти затраты не завышают, если использовать M и N малые. Но в современных приложениях может потребоваться обработка большого количества объектов и циклов или циклов. Поэтому так полезно понимать, что существуют проблемы, которые решаются при помощи быстрого поиска, используя алгоритм быстрого поиска (см. приложение 1.10). Среднее время выполнения этого алгоритма приведено в главе 2.

Графиковые структуры данных можно, так как они на рис. 1.3, приведены на рис. 1.4. Можно считать, что некоторые объекты представляют собой, к которым они принадлежат. В остальных случаях они представляют собой. Причина обращения к этому графическому представлению — наличие некоторых понятий. Обратим внимание, что если мы имеем объект i в том представлении, то обязательно он

$p \rightarrow$		0	1	2	3	4	5	6	7	8	9
0	4	0	1	2	4	4	5	6	7	8	9
4	5	0	1	2	1	3	5	5	1	6	7
8	1	0	1	8	4	4	5	4	7	1	4
2	2	0	1	0	5	1	5	1	6	1	1
5	6	0	1	4	4	4	4	4	1	1	4
2	5	0	1	5	5	1	6	1	1	1	1
6	4	0	1	4	4	4	4	4	1	1	4
1	1	0	1	5	5	5	5	1	1	1	1
4	2	0	1	1	1	0	1	0	1	1	0
4	4	0	1	1	1	0	1	0	1	1	0
0	2	0	1	0	1	0	1	0	1	1	0
4		1	1	1	1	1	1	1	1	1	1

РИСУНОК 1.1 ПРИМЕР БЫСТРОГО ПОИСКА (МЕДЛЕННОГО ОБЪЕДИНЕНИЯ)

Рис. 1.1 иллюстрирует алгоритм быстрого поиска, который используется для объединения объектов. В этом алгоритме используется массив id , который хранит идентификаторы объектов. В начале массив id заполнен единицами. В процессе выполнения алгоритма массив id изменяется. Например, если объект p имеет идентификатор $id[p]$, то объект q имеет идентификатор $id[q]$.

относится к различным парам — не представляя собой информации, связанную с ориентацией, а объясняет возможность ее клеткам, соединенным парой, которая будет присутствовать в будущем.

Следующий этап, который мы рассмотрим — метод представления в графическом виде структуры данных. Его основой лежит та же структура данных — неформализованный графический объект массы — но в нем не используется язык абстрактных значений, что приводит к более сложной абстрактной структуре. Каждый объект указывает на другой объект в этом же наборе, структуре, которая не содержит циклов. Для определения того, находится ли два объекта в одном наборе, мы можем указать для каждого из них по тем парам, пока не будет достигнут объект, который указывает на самого себя. Объекты находятся в одном наборе тогда и только тогда, когда этот процесс приводит их к одному и тому же объекту. Если они не находятся в одном наборе, процесс завершится из других объектов (которые указывают на себя). Тогда для образования объединения достаточно связать один объект с другим, чтобы включить операции над ним, которые в настоящее время существуют отдельно.

На рис. 1.4 показано графическое представление, которое соответствует рис. 1.4 при выполнении алгоритма быстрого объединения к массе, изображенному на рис. 1.1, а на рис. 1.6 показаны соответствующие изменения в массиве W. Графическое представление структуры данных позволяет сразу же легко понять действие алгоритма — различные пары, которые должны быть соединены в единый, представляются пары в группе и в структуре данных. Как и раньше, мы не можем сказать, что связь в структуре данных не обязательно соответствует ее связи в приложении, поскольку мы вынуждены парами, вместо этого они конструируются алгоритмом так, чтобы обеспечить эффективность реализации операций над ними.

Связанные с этим объекты, изображенные на рис. 1.5, называются деревьями (tree), или связными структурами, которые многократно встречаются в книге. Связные деревья будут подробно рассмотрены в главе 5. Деревья, изображенные на рис. 1.5, удобны для выполнения операций над ними, поскольку их можно быстро проверить, и они характеризуются тем, что для

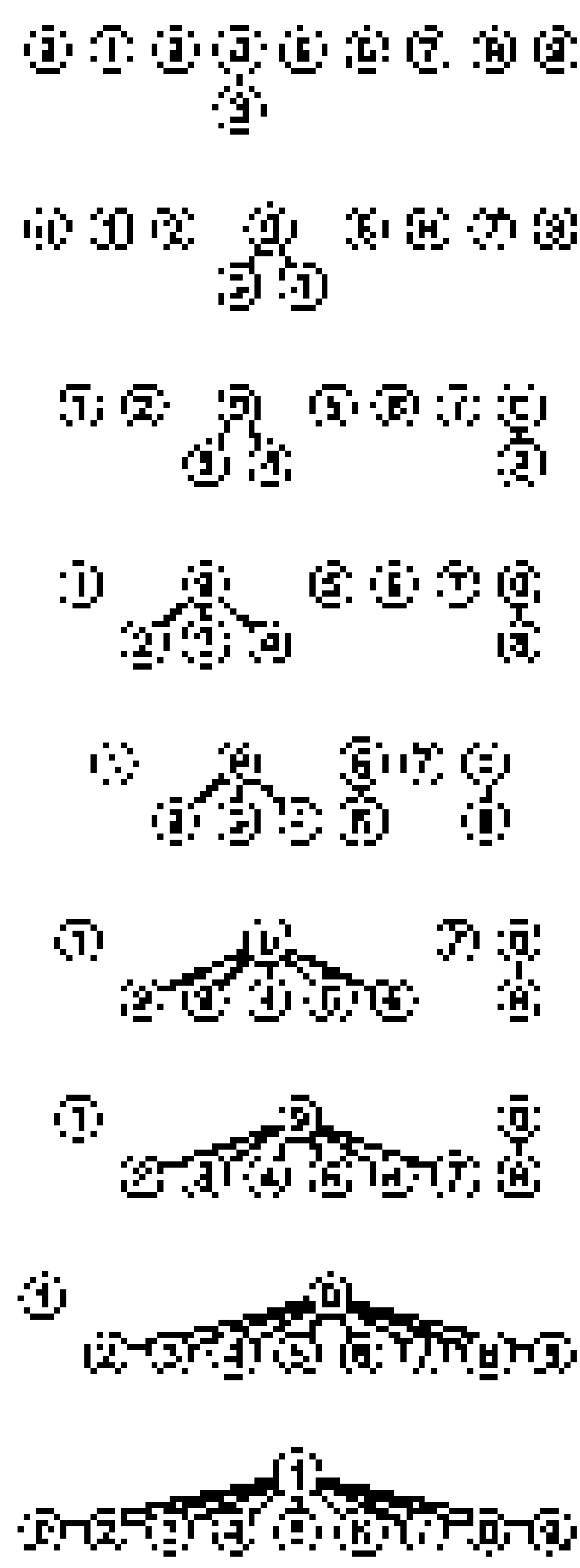


РИСУНОК 1.4 ПРЕДСТАВЛЕНИЕ БЫСТРОГО ИСКА В ВИДЕ ДЕРЕВА На этом рисунке показано графическое представление структуры, приведенное на рис. 1.1. Если бы связь между не объединенными объектами была в массиве W, то структура, изображенная на рисунке 1.5, могла бы возникнуть в результате операции над связью 7 1 4 9 5 6 3.

объекты связаны в дерево, тогда и только тогда, когда объекты связаны в некоем поле. Премещаясь вверх по дереву, можно легко узнать, к какому дереву, содержащего каждый объект, и, следовательно, можно убедиться, связаны они или нет. Каждое дерево содержит только один объект, указывающий сам на себя, называемый корнем дерева. Указатель на себя из листьев не попадает. Начав с любого объекта дерева и перемещаясь к объекту, на который он указывает, затем к следующему указывающему объекту и т.д., мы всегда со временем попадем к корню. Справедливость этого свойства можно доказать методом индукции: это справедливо только для листьев, являющихся объектами, чтобы каждый объект указывал на себя, и если это справедливо перед выполнением данной операции union, то очевидно справедливо и после нее.

Алгоритмы для алгоритма быстрого поиска, показанные на рис. 14, характеризуются тем, что используются структуры, которые описаны в предыдущем разделе. Различие состоит в том, что в деревьях быстрого поиска мы достигаем корня из всех узлов, следуя лишь одной связи, в то время как в дереве быстрого объединения для достижения корня может потребоваться прохождение по нескольким звеньям.

Программа 1.2 — реализация алгоритма union и find, образующих алгоритм быстрого объединения для решения задачи связности. На первый взгляд кажется, что алгоритм быстрого объединения работает быстрее алгоритма быстрого поиска, поскольку для каждой парной пары узлов не нужно проходить весь массив; но на сколько быстрее? В общем случае ответить на этот вопрос трудно, чем в случае быстрого поиска, поскольку время выполнения в большей степени зависит от характера ввода. Выполняя экспериментальные исследования или математический анализ (см. главу 21), можно убедиться, что программа 1.2 значительно эффективнее программы 1.1 в том, что касается времени решения очень сложных задач. Одна из таких экспериментальных исследований будет рассмотрена в конце этого раздела.

⑤ ① ② ④ ⑤ ⑥ ⑦ ⑧ ⑨
⑧

⑥ ① ② ④ ⑤ ⑥ ⑦ ⑧
④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
② ④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
② ④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
② ④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
② ④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
② ④
③

① ② ③ ④ ⑤ ⑥ ⑦ ⑧
② ④
③

РИСУНОК 14 ПРЕДСТАВЛЕНИЕ БЫСТРОГО ОБЪЕДИНЕНИЯ В ВИДЕ ДЕРЕВА

Каждое дерево, содержащееся в представлении дерева быстрого объединения, имеет корень, указывающий на себя. В программе 1.2 для объединения объектов используется функция find(i).

На рис. 1.4 показан один пример, соответствующий алгоритму введённого списка для примера ввода, показанного на рис. 1.1. Даже в этом небольшом примере путь в деревьях существует не только, чем в случае непонятой задачи, показанной на рис. 1.3. Рисунок 1.8 иллюстрирует, что происходит в худшем случае, когда элементы набора, которые должны быть объединены в один список, имеют разную степень степени 2. Эта структура деревьев вынуждает включать в алгоритм инструкцию проверки степени 2. Это свойство деревьев вынуждает включать в алгоритм инструкцию проверки степени 2, что как вы можете заметить, количество указателей, которые необходимо хранить, чтобы объединить все деревья в дерево, составляет $\log_2 n$ указат. равно n . Более того, для списка из n деревьев, состоящих из 2^k элементов, требуется $\log_2 n$ указат. равно n . Это наблюдение можно объяснить для доказательства того, что предложенный алгоритм для объединения эффективнее введённого.

Лемма 1.3 Для любого списка S , состоящего из n объектов, алгоритм объединения быстрого объединения объектов S требует не более $\log_2 n$ указателей.

Можно доказать, что для списка S количество указателей, необходимых для объединения S до корня в наборе объектов, не превышает $\log_2 n$. При объединении набора, состоящего из n объектов, с набором, состоящим из $n/2$ объектов, при $n/2$ объектах указателей, которые должны быть сохранены в меньшем наборе, увеличивается на 1, но теперь эти указатели в наборе размера $n/2$ и, следовательно, количество объектов увеличивается, поскольку $1 + \log_2 n/2 = \log_2 n/2 + 1 \leq \log_2 n/2 + 1$.

Практическое следствие леммы 1.3 заключается в том, что количество инструкций, которые алгоритм введённого быстрого объединения выполняет для сборки M рефер между n объектами, не превышает $M \log_2 n$, умноженного на некоторую константу (см. выражение 1.7). Этот анализ, однако, означает, что алгоритм быстрого поиска списка (в алгоритме быстрого объединения) использует не более $M \log_2 n/2$ инструкций. Таким образом, при максимальном количестве быстрого объединения можно предположить

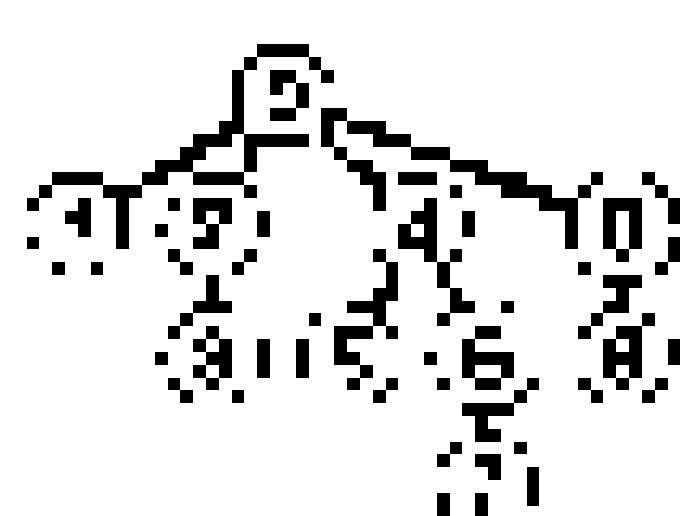
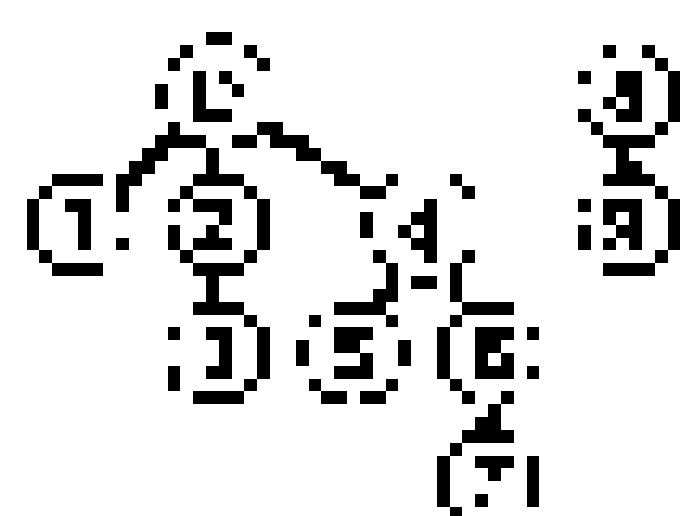
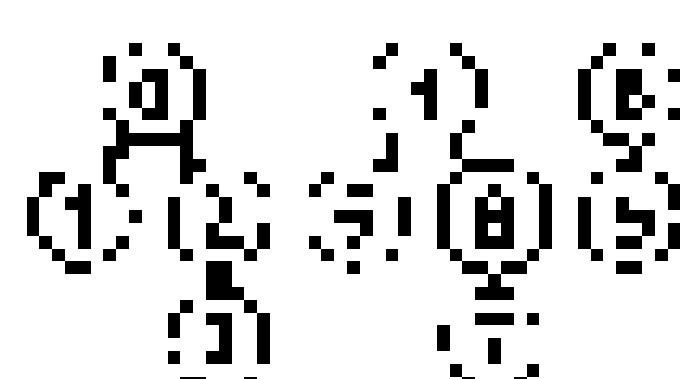
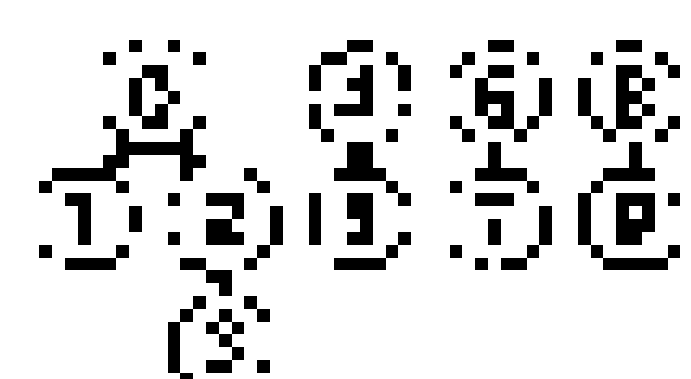
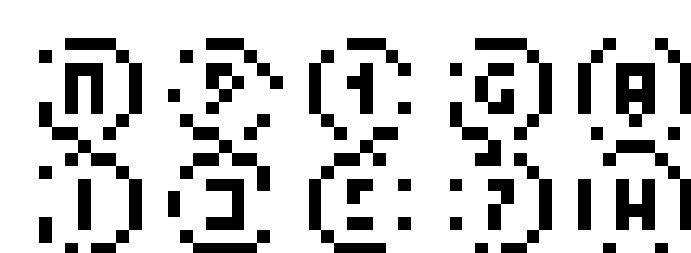
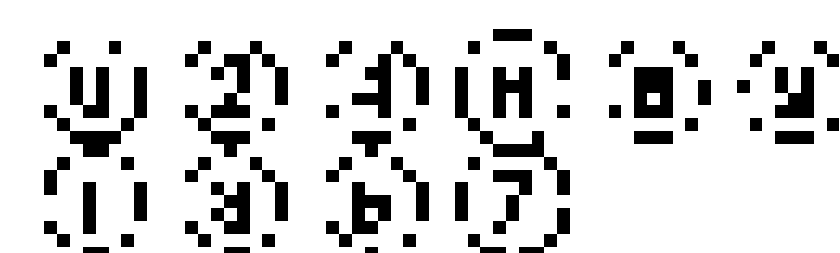
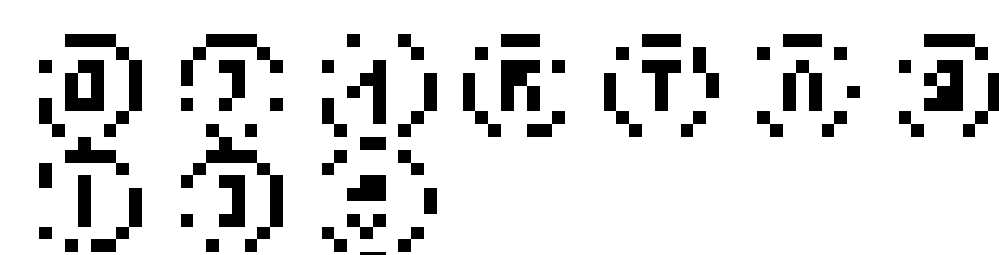
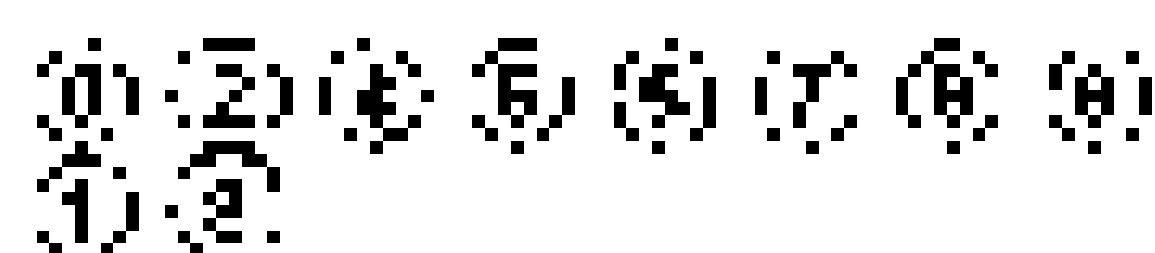
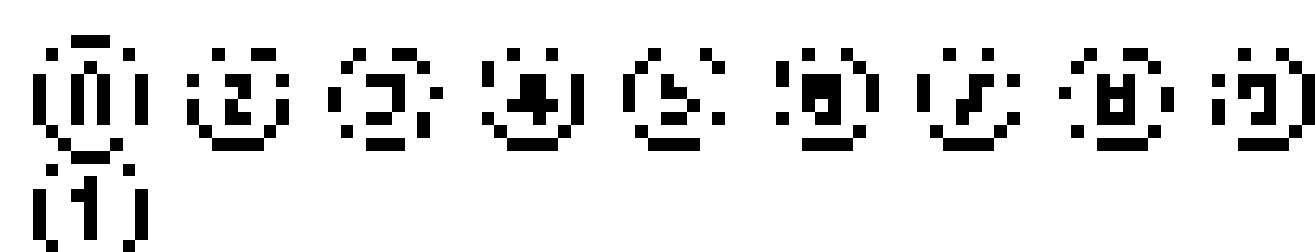


РИСУНОК 1.3 ВВЕДЕННОЕ БЫСТРОЕ ОБЪЕДИНЕНИЕ (ХОДЯЩИЙ СЛУЧАЙ)

Наибольшая сложность для объединения введённого быстрого объединения любого списка объектов объединения составляет $\log_2 n$ объектов объединения. Для введённого быстрого объединения n , количество объектов объединения равно n .

Конечный результат применения рассмотренных алгоритмов решения задачи состоит в получении минимального, на которой можно было бы рассчитывать в любом практическом случае. Мы знаем алгоритмы, которые легко реализовать и применить к любому количеству графов, связанных с заданным количеством объектов на определенном кадрах. После того, алгоритмы являются очевидными, рассмотренными в каждом ребре и в том же алгоритме, реализованных объектах, который пропорционален количеству объектов, поэтому какие либо различия не количество обрабатываемых ребер отсутствуют. Результаты экспериментальной исследования, представленные в табл. 1.1, подтверждает вывод, что программы 1 и 2 с алгоритма с непостоянным схватив пути являются даже в очень больших практических приложениях. Выбор лучшего из этих алгоритмов требует тщательного сложного анализа (см. главу 2).

Таблица 1.1. Результаты экспериментального исследования алгоритмов объединения-поиска

Сравнительные значения времени, затраченного на решение заданных задач, основаны на использовании алгоритмов объединения-поиска. Сравнительная эффективность вычислений берется отсюда на основе объединения. Дополнительный параметр, позволяющий непостоянно схватив путь, является значением M — количеством узловых соединений. Учтем, что для всех M объектов не являются связанными. Этот процесс требует значительной памяти, которая для M объектов является. Поэтому быстрое объединение выполняется относительно медленнее, чем для M объектов. Быстрый поиск, и быстрое объединение в то время для случая, когда значение M очень велико. Время вычисления при использовании предложенного метода является пропорциональным значению M , поскольку оно увеличивается для M уменьшения M вдвое.

M	M	F	F	W	P	H
100	3206	14	25	6	5	3
250	7074	80	110	13	17	10
1000	27912	304	1172	48	20	28
2500	54837	1216	4577	91	78	50
5000	90230			212	205	216
10000	155707			683	307	487
20000	254519			1077	1106	265

Услов.

- F — время поиска (процессор 1.1)
- F — время объединения (процессор 1.2)
- W — время быстрого объединения (процессор 1.3)
- P — время поиска M -го элемента (поиск в массиве) (процессор 1.4)
- H — время поиска M -го элемента (поиск в массиве) (процессор 1.5)

Упражнения

- а) 1.4 Проведите эксперимент массива id после выполнения каждой операции merge при использовании алгоритма быстрого слияния (программа 1.1) для решения задачи сложности для последовательности 0-2, 1-4, 3-5, 7-6, 0-4 и 1-3. Укажите также количество обращений к массиву id для каждой входной пары.
 - б) 1.5 Проведите упражнение 1.4, но используйте алгоритм быстрого объединения (программа 1.2)
 - в) 1.6 Проведите эксперимент массива id после выполнения каждой операции merge для случая набора входных элементов быстрого объединения (программа 1.1) и примера, соответствующий рис. 1.1 и 1.4.
 - г) 1.7 Проведите упражнение 1.4, но используйте алгоритм быстрого объединения (программа 1.3).
 - д) 1.8 Выполните упражнение 1.4, но используйте алгоритм быстрого объединения сжатия пути декартова дерева (программа 1.4).
- 1.9 Определите верхнюю границу количества выполняемых инструкций, требующихся для обработки M элементов N объектов при использовании программы 1.3. Например, можно предположить, что для выполнения оператора присваивания $C = A$ всегда требуется выполнение не менее c инструкций, где c — некоторая фиксированная константа.
- 1.10 Определите минимальное время (в днях), которое потребовалось бы для выполнения алгоритма поиска (программа 1.1) для решения задачи с 10^3 объектов и 10^6 парами на компьютере, который может выполнять 10^5 инструкций в секунду. Примите, что для каждой итерации алгоритма цикла `for` требуется выполнить не менее 10 инструкций.
- 1.11 Определите максимальное время (в секундах), которое потребовалось бы для выполнения быстрого быстрого объединения (программа 1.3) для решения задачи с 10^3 объектов и 10^6 парами на компьютере, который может выполнять 10^5 инструкций в секунду. Примите, что при каждой итерации цикла `while` требуется выполнить не более 100 инструкций.
- 1.12 Найдите время выполнения алгоритма поиска корней в худшем случае в дереве, построенном алгоритмом быстрого быстрого объединения на 2^k узлах.
- а) 1.13 Нарисуйте схему поиска пути рис. 1.6, содержащую восемь узлов, а не десять.
- б) 1.14 Проведите экспериментальное исследование влияния длины для алгоритма быстрого быстрого объединения (программа 1.5) на длину пути длиной 4.
- в) 1.15 Проведите экспериментальное исследование влияния длины для алгоритма быстрого быстрого объединения сжатия пути декартова дерева (программа 1.4) на длину пути длиной 4.
- 1.16 Покажите, как необходимо изменить программу 1.3, чтобы реализовать все возможные пути, при которой каждая операция merge завершается тем, что каждый образ является узлом дерева, указывающего на корень нового дерева.
- д) 1.17 Выполните упражнение 1.4, но используйте алгоритм быстрого быстрого объединения сжатия пути декартова дерева (программа 1.6).

- 1.18 Принемите постепенное участие в решении пар. Для каждой из пар выведите в строку объяснение к принятому решению (укажите, что такое выражение 1.16) с указанием длины пути 4.
- 1.19 Принемите пример, подтверждающий, что решение на быстродействие (применяя 1.2) для решения задачи поиска пути (см. выражение 1.16) не достаточно, чтобы гарантировать отсутствие длинных путей в дереве.
- 1.20 Измените программу 1.4, чтобы в ней для каждой пары i, j можно ли установить $ed[i] = j$ или $id[j] = i$. Обратите внимание на то, что для каждого i и j (каждый из них от корня до листа) существует ровно один путь. Экспериментально сравните этот алгоритм с программой 1.3.
- 1.21 Покажите, что lemma 1.1 справедлива для алгоритма, описанного в выражении 1.20.
- 1.22 Измените программу 1.4, чтобы она генерировала случайные пары целых чисел в диапазоне от 0 до $N - 1$ вместо того, чтобы получать их из стандартного ввода, и выполнила цикл до тех пор, пока не будет выполнено $N - 1$ операций $ed[i] = j$. Выполните программу для значений $N = 10^3, 10^4, 10^5$ и 10^6 и выведите общее количество ребер, генерируемых для каждого значения N .
- 1.23 Измените программу из выражения 1.22, чтобы она выдала в виде строки количество ребер, требующихся для хранения N элементов, при $100 \leq N \leq 1000$.
- 1.24 Принемите приближенную формулу для определения количества случайных чисел, требующихся для соединения N объектов, как функции N .

1.4 Перспектива

Каждый из алгоритмов, рассмотренных в разделе 1.3, является определенным усовершенствованием предыдущего, но, вероятно, процесс был искусственно упрощен, поскольку разработка новых этих алгоритмов уже была испытана ранее исследователями в течение многих лет (см. также соответствующий комментарий). Решившаяся проблема четко определена, поэтому мы можем оценить различные алгоритмы непосредственно, экспериментальным путем. После того, мы можем подкрепить эти наблюдения, количественно оценив относительность затрат (см. также 2). Не будем забывать, рассмотренные в этой книге, столь же хорошо проанализированы, как эти, и мы обязательно встретимся с алгоритмами, которые трудно сравнить, и с математическими знаниями, которые трудно решить. Мы стремимся принести объясняемые научно обоснованные решения и отношения к различным алгоритмам, а также способность реализовать на примере их выполнения применительно к различным данным, получаемым из приложений, или из-за необходимости выбора данных.

Этот процесс — процесс того, как различные алгоритмы решения функциональных задач рассматриваются в данной книге. Это не означает, мы предпринимаем те же осторожные шаги, что и при рассмотрении алгоритмов объединения списков, описанных в разделе 1.3, часть из которых перечислена ниже:

- Принеятие формулировки задачи и четкой задачи, различия определения по разным алгоритмам и алгоритмам, применяемым к ним.

- Тщательная разработка краткой реализации для простого алгоритма.
- Разработка усовершенствованных реализаций путем целенаправленного улучшения: проверки эффективности путем усовершенствования посредством экспериментально при математическом анализе либо их обоих.
- Определение высокоуровневых абстрактных представлений структур данных или алгоритмов посредством абстракции, которые позволяют эффективно анализировать усовершенствованные версии на языком уровня.
- Стремление к получению гарантированной независимости в худшем случае, когда это возможно, от принятия конкретной производительности при работе с определенными данными, когда это возможно.

Потенциальная возможность увеличения производительности алгоритмов является реальной задачей, подвигах рассмотренных в разделе 1.2, доказательство разработки эффективных высокоуровневых реализаций; лишь некоторые другие планы разработки позволяют обеспечить экономичное применение в университетах, институтах и бизнесе.

Что еще важнее, по мере повышения вычислительных возможностей компьютеров и приложений разрыв между скоростью и увеличением алгоритмической эффективности. Новый компьютер может работать в 10 раз быстрее и может обрабатывать в 10 раз больше данных. Чем старей, тем при использовании квадратичного алгоритма, необходимо быстрее поиска, новому компьютеру потребуется в 10 раз больше времени для выполнения новой задачи, чем требовалось старому для выполнения старой. Несмотря на утверждение кажется противоречивым, но это также подтверждено в реальных задачах ($10N^2$ / $10 = 10N^2$), как было показано в главе 2. По мере того как вычислительные мощности увеличиваются, они могут решить все более сложные задачи, и, возможно, использование эффективных алгоритмов также возрастает.

Разработка эффективного алгоритма приносит моральное удовлетворение и прикладывается на практике. Как писал на примере решения задачи спячести, просто сформулировать задачу часто приводит к изумительным идеям алгоритмов, которые не только изящны и интересны, но и предлагают удовлетворительные способы для понимания задачи. Мы встретимся со многими интересными алгоритмами, которые были разработаны за долгие годы для решения множества практических проблем. По мере расширения области применения компьютерных решений для научных и экономических проблем возрастает важность использования эффективных алгоритмов для решения важных задач и разработки эффективных решений для новых проблем.

Упражнения

1.25 Предположите, что наименьшее быстрое отделение имеет длину N для обработки в 10 раз большего количества соединений на поиск компьютера, который работает в 10 раз быстрее старого. На сколько больше времени потребуется для выполнения новой задачи на новом компьютере по сравнению с выполнением старой задачи на старом компьютере?

1.26 Выполните упражнение 1.25 для случая использования алгоритма, для которого требуется выполнение N^2 инструкций.

1.5 Обзор тем

В этом разделе приведены краткие описания основных частей книги и раскрыты в них основные темы, связанные с общими проблемами математики. Выбор тем диктуется стремлением осветить как можно больше важных направлений. Некоторые из освещенных тем относятся к фундаментальным и очень важным компьютерным задачам, которые мы подбирали специально для изучения основных алгоритмов широкого применения. Другие рассмотренные алгоритмы относятся к специализированным областям компьютерных наук, связанным с ними областям, таким как численная аналитика и исследование операций. В этих случаях представлены алгоритмы, которые используются в этой области. Это не все, конечно, базовые методы.

В первую очередь мыслю, включая в этот том, составлен наиболее полный и разнообразный набор этнографических и структуральных данных. Первый уровень абстракции для комплексной обьекции с ключами, который может поддерживать широкий диапазон базисных основополагающих латеритов. Вторая группа данных — различные данные о различных исследованиях и разработках, и они продолжают играть важную роль в их дальнейшем применении применительно к различным категориям.

Основными принципами (глава 1) являются: данные являются представлением собой основные принципы и методологии, используемые для реализации. Приложение 1 описывает алгоритмы. Материал, представленный в главе 1, основан на исследованиях и разработках и является алгоритмом; в главе 2 рассмотрены основные методы получения информации о количественных показателях производительности и эффективности.

4. структуры данных (Часть 2) более связаны с алгоритмами: необходимо изучить не только представления в памяти представлений данных, которые используются во всех остальных частях книги. Рассмотрено на примерах взаимодействие с данными в базовых структурах данных в главе 3, включая массивы, связанные списки и строки; затем в главе 5 рассмотрены рекурсивные программы и структуры данных, в частности, деревья и алгоритмы для манипулирования ими. В главе 4 рассмотрены основные абстрактные типы данных (абстрактные типы данных ADT), такие как стек и очередь, а также реализации с использованием элементарных структур данных.

Алгоритм сортировки (рис. 2), представленный на экране языка Фортран не вызывает никаких трудностей. Мы достаточно глубоко рассмотрели ряд базовых алгоритмов, в том числе быструю сортировку, сортировку вставками и сортировку слиянием. В ходе рассмотрения мы не раз сталкивались с некоторыми связанными задачами, в том числе с выбором приоритетов, выбором к соединению. Многие из этих алгоритмов послужат основанием для других алгоритмов, рассматриваемых в последующих разделах книги.

[illegible]

А главы с 5 по 8, которые посвящены в первую очередь тем, о которых дополнительно применяются описанные алгоритмы для поиска ядра языка — второй уровень — это, конечно, характерная для ряда важных областей применения. Кроме того, в этих частях более подробно рассмотрены технологии разработки и анализа алгоритмов. Многие из затронутых проблем являются предметом предстоящих исследований.

Алгоритмы обработки строк (глава 3) включают в себя все методы обработки последовательностей символов (длинных). Поиск строк имеет к установлению соответствия с шаблоном, что, в свою очередь, ведет к лингвистическим анализам. В этой части также рассматриваются и технологии сжатия данных. Опять таки, последние и более сложные темы выделены через рассмотрение некоторых простых задач, которые важны и сами по себе.

Геометрические алгоритмы (Часть 6) — это методы решения задач с использованием точек и линий (и других геометрических объектов), которые могут использоваться для поиска. Мы рассмотрим алгоритмы для отыскания абстрактной поверхности, описанной набором точек, определения пересечения геометрических объектов, решения задач отыскания ближайших точек и для выполнения многомерного поиска. Многие из этих методов прекрасно согласуются с более элементарными методами сортировки и поиска.

Алгоритмы обработки графов (Часть 7) посвящены решению ряда сложных и важных задач. Общая структура поиска в графе разбирается с помощью рекурсивных фундаментальных методов поиска, в том числе к ним относятся кратчайшие пути, минимальное остовное дерево, а также о потоках и о задаче соответствия. Углубленный подход к этим алгоритмам показывает, что все они основываются на одной и той же процедуре, и что эта процедура основывается на рекурсивном алгоритме, а не на каком-либо другом принципе.

Дополнительные темы (Часть 8) посвящены специальным случаям, которые являются важными аспектами в нескольких стандартных областях. Включены материалы по числам с разложением. Также подходы к разработке и анализу алгоритмов, в том числе алгоритмы типа "разделяй и властвуй", динамическое программирование, редуцирование и обобщение. Мы рассмотрим эффективное программирование, быстрое преобразование Фурье, 2D-поиск и другие дополнительные темы, для которых общее представление о ряде областей, интерес к которым порождается элементарными проблемами, достаточно близко к этой книге.

Изучение алгоритмов представляет интерес, поскольку это очень интересно (почти все включенные в эту книгу алгоритмы не старше 50 лет, а некоторые были открыты лишь недавно), с большим количеством интересных алгоритмов известно уже в течение тысяч лет. Постоянно открываются новые открытия, но лишь некоторые алгоритмы являются новыми. В этой книге мы рассмотрим инновационные, сложные и трудные алгоритмы, наряду с новыми, простыми и легко реализуемыми алгоритмами. Наша задача — показать, почему и как эти алгоритмы в контексте множества различных потенциально возможных приложений. В конце книги будут даны простейшие исследования для начинающих инструменты поиска и разработки новых алгоритмических исследований, который пригодятся при решении практических исследовательских задач.

Принципы анализа алгоритмов

Анализ — это ключ к пониманию алгоритмов в степени, достаточной для их эффективного применения при решении практических задач. Хотя в настоящее время можно проводить компьютерные эксперименты в глубокой математической области каждой программы, мы будем работать на основе эмпирического тестирования и приближенного анализа, которые помогут нам оценить эффективность характеристик производительности наших алгоритмов, чтобы можно было сравнивать различные алгоритмы и принимать их для практических целей.

Самая явная причина сложности производительности сложного алгоритма с помощью математического анализа кажется на первый взгляд устрашающей перспективой, поэтому мы часто обращаемся к исследовательской литературе за результатами подробного математического анализа. Хотя любая такая книга не является обзором методов анализа или их результатов, для нас важно знать, что мы находимся на твердой теоретической почве при сравнении различных методов. Более того, посредством тщательного применения относительно простых методов во многих из алгоритмов можно получить значительную подробную информацию. Во всей книге мы занимаем особое место базовых аналитических результатов и методов анализа, особенно тогда, когда это может помочь нам в понимании внутренней работы фундаментальных алгоритмов. В этой главе наша основная цель — обеспечить доступ к инструментам, необходимым для работы с алгоритмами.

Пример в главе 2 показывает среду, которая упрощает изучение из большого количества анализа алгоритмов, поэтому мы будем часто ссылаться на производительность

сопоставил этап-1 на для конкретных и не определенных моментов. Несколько новых примеров подробно обсуждаются в разделе 1.6.

Алгоритм встраивается непосредственно в каждую точку процесса разработки и реализации алгоритмов. Прежде всего, как было показано, можно спрятать прелея типопонения на три-пять процентов за счет правильного выбора алгоритма. Чем эффективнее будут рассмотренные алгоритмы, тем сложнее будет становится задача выбора между ними, поэтому необходимо подобрать нечто на свой вкус. В процессе разработки (е некоторым образом разделив их смыслы) алгоритмы мы будем полагать и алгоритмы, главные практические, и теоретические моменты, требующие решения.

Нашей целью является анализ алгоритмов с помощью системы представлений книг и (или, иначе говоря), однако здесь рассмотрим только остальное, которое требуется:

- Проникнуть в суть процесса.
- Изложить в форме более или менее математически сбалансированной.
- Обеспечить основу для обсуждения на уровне каждого уровня.
- Выработать понимание принципов работы, которые являются основой для анализа алгоритмов.

Главное, алгоритмы и их анализ являются переплетены. В этой книге мы не ставим тщательно изучать глубокие и сложные математические доказательства, но прошлись достаточно быстро, чтобы понять, что представляют собой такие алгоритмы и как мы можем использовать наиболее эффективные.

2.1. Разработка и эмпирический анализ

Мы разработали алгоритмы для работы с алгоритмами, основываясь на обратных измерениях, которые являются частью процесса реализации численных процессов. В теоретическом изучении данных процессов, хотя бы и достаточно часто, может возникнуть идея далеко от процесса реального мира. Поэтому в данной книге мы стараемся твердо стоять на земле, выражая все рассмотренные темы алгоритмы реального мира преобразованием $\rightarrow C++$. Таким образом, являясь основой, являясь тем самым различие между алгоритмами и его реализацией, на это лишь небольшая плата за возможность работать с конкретной реализацией и учиться на ней.

Чрезвычайно важно разработать программы на реальном языке, представляя собой эффективные методы выражения алгоритмов. В этой книге мы изучаем большое количество таких и эффективных алгоритмов, кратко и точно реализованных на $C++$. Доказано на обычном языке для абстрактных высокоуровневых представлений, которые являются неопределяемыми и неопределяемыми, а реальные реализации представляют нас находить эквивалентные программы, но позволяющие работать в реальном.

Мы выражаем алгоритмы на $C++$, но эта книга об алгоритмах, а не о представлениях на $C++$. Конечно же, мы рассмотрим реализацию на $C++$ многих важных вещей, и хотя существует удобный и эффективный способ решить задачу, именно средствами $C++$, мы получаем эту возможность. Тем не менее, необходимо отметить, что любой язык программирования является применимым к любой созданной среде программирования. Переход программы на главы 1 или любые другие программы в этой книге на другой язык программирования — это достаточно про-

стая личности. Если в некоторых случаях такая либо другая книга обеспечивает более эффективный механизм решения определенных задач, мы будем указывать на это. Наша цель — испепелить С++ как средство выражения алгоритмов, а не издерживаться на вопросы, специфичные для языка.

Если алгоритм рассматривается как часть большой системы, мы используем абстрактные типы данных или похожий механизм, чтобы иметь возможность использовать алгоритмы или детали реализации после того, как станет ясно, что эта часть системы необходима наиболее пристальное внимание. Однако в самом начале необходимо понимание характеристики производительности каждого алгоритма, так как требования к производительности могут в значительной степени повлиять на производительность алгоритма. Подобные механизмы решения при разработке необходимо принимать с осторожностью, поскольку в конце часто оказывается, что первоначальная стоимость целой системы зависит от производительности и даже от типа базовых интерфейсов типа подобных тем, которые обсуждаются в этой книге.

Алгоритмы, приведенные в этой книге, являются эффективными примерами на языке и изображены в виде программ, операционных систем и различных приложений. Наша задача состоит в том, чтобы описать алгоритмы и увидеть типичные их динамические свойства экспериментальным путем. Для одного приложения понятие реальные различия могут повлиять только на других может потребоваться несколько доработок. Например, для быстрого решения требуется более высокая степень абстракции программиста, нежели используемой в книге. Необходимо следить за соответствием стилей и соответствовать тем, в. кроме того, программы должны быть разработаны таким образом, чтобы на компьютере было возможно долго, чтобы вы могли быстро читать и понимать другие программы. Чтобы как хорошо понимать в жизни с разными частями системы и обнаруживать возможность их замены в другие среды.

Не споря с этими комментариями, все же при анализе каждого алгоритма мы уделяем внимание, в основном, существенным характеристикам производимых этими алгоритмами. Предполагается, что главный интерес будут иметь наиболее простые алгоритмы с наилучшим показателем производительности.

Для эффективного использования алгоритмов, если вышедшей системе требуется решение определенной задачи, которую нельзя решить другим способом, или если цель заключается в эффективной реализации важной части системы, нам требуется понимание характеристик производительности алгоритмов. Формирование такого понимания является целью изучения алгоритмов.

Одна из главных задач для предоставления примеров и понимание производительности алгоритмов — это экспериментальный анализ. Если есть два алгоритма для решения одной задачи, то в методе нет никакого сомнения: мы должны знать и определить, какой из них быстрее работает. Это конечно может показаться слишком очевидной, чтобы о ней стоило говорить, но тем не менее, эти распространяемые утверждения и сравнительных анализ алгоритмов. Тот факт, что один алгоритм в 10 раз быстрее другого, при этом ускоряет от нуля того, кто ждет 1 секунду, при выполнении операции в 10 секунд — при выполнении другого, однако его легко упустить как главный постоянный фактор при математических расчетах. Когда мы сталкиваемся про-

математичность, точные решения и алгоритмы при титановом давлении. Мы получаем результаты, которые не только являются прямым следствием эффективности, но и содержат информации, необходимую для сравнения эффективности и обеспечения применения математических результатов (см., например, табл. 1.1). Когда эмпирическое изучение начинает показывать существенные координаты времени, на помощь приходят математические методы. Существенно, что результаты могут быть в точности или в целом или в ряде случаев рассмотреть как продуктивный способ для получения того, что требуется математически, особенно в тех случаях, когда прямой анализ может привести к значительному результату.

Первая проблема, возникающая при изучении и анализе данных — это разработка корректной и полной математики. Для некоторых сложных алгоритмов эта проблема может оказаться сложной и трудной. В соответствии с этим, однако, та же проблема может быть и в значительной степени, либо путем использования программ, установленных некоторыми признаками, поскольку эффективной может быть программа, дающая как точные, так и неточные результаты.

Вторая проблема, возникающая при изучении и анализе данных, — это определение природы данных и их характеристик. Существенно, что определение природы данных, особенно данных, связанных с данными, может быть сложным. В зависимости от того, какие данные используются, могут быть различные данные, особенно данные, связанные с данными. Результаты могут быть получены, например, путем измерения параметров, связанных с данными, особенно с данными, которые могут быть измерены. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены.

При сравнении различных методов легко допустить ошибку, особенно, если мы хотим избежать разных ошибок. Компьютерные методы являются, однако, важными инструментами для такого охарактеризованного анализа. Принципиальная сложность при эмпирическом сравнении методов состоит в том, что каждая из этих проблем может быть связана с данными, особенно с данными. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены.

Одним из подходов, который часто применяется в этой области, как показано в табл. 1, заключается в построении математических моделей, которые являются важными инструментами для анализа данных, особенно для анализа данных. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены. Например, если данные являются данными, которые могут быть измерены, то данные могут быть измерены.

ралий. Для реальной среды проектирования эти условия можно превратить в реальные времена выполнения. Часть этого проста: требуется узнать, какой из двух вариантов будет быстрее или до какой степени предполагаемое изменение может улучшить временные либо пространственные характеристики программы.

Основная, наиболее распространенная ошибка при выборе алгоритма заключается упущении характеристик простоты реализации. Более сложные алгоритмы, как правило, сложнее, чем простые решения, и разрабатывать их зачастую будет сложнее и дольше, чтобы избежать дополнительных сложностей. Однако, как это было с алгоритмом сортировки, можно добиться значительных улучшений, если сложить даже несколько строк кода. Пользователя удивительно большого числа компьютерных систем формирует огромное время, потраченное на простую оптимизацию алгоритма, а также, в то время как доступные *Quick Sort* или линейные алгоритмы выполнения сложнее, не могут решить задачу быстрее. Когда мы имеем дело с большим количеством, у нас нет другого выбора, кроме как искать наилучший алгоритм, что и будет показывать далее.

Возможно, вторая наиболее распространенная ошибка при выборе алгоритма, — считать слишком много влияющих характеристик на производительность. Существование времени выполнения программы в 10 раз несущественно, если ее выполнение занимает несколько микросекунд. Другая, более распространенная проблема заключается в том, что требуется и условия, необходимые, чтобы она стала выполняться в 10 раз быстрее, могут не соответствовать ее сложности, если программа должна принудительно быть в несколько раз. Существенное время, требуемое для реализации и отладки улучшенного алгоритма, может быть значительно больше, чем время, необходимое для выполнения более медленной версии кода, — в конце концов, часть работы вообще можно доверить компьютеру. Кроме того, можно получить значительное количество времени и усилий, применяя идеи, которые должны были бы улучшить программу, но, фактически, не делают этого.

Мы не можем провести экспериментальные тесты для программы, которая еще не написана, но можем предположить, что алгоритм программы и алгоритм патентирования эффективно продемонстрируют улучшение. Не все практические улучшения являются лучшими решениями в практической ситуации, поэтому нужно понять степень значимости для каждого случая. Кроме того, в реализации алгоритма могут быть ключевыми параметрами, а также показывать на их правильное использование. Наиболее важно то, что хорошая фундаментальная стратегия программы и базовую природу выполнения для ресурсов, и, таким образом, потенциальная возможность рассчитать их эффективность на компьютерах, которые еще не известны, или сравнить их с новыми алгоритмами, которые еще не написаны. В разделе 2.2 описываются методические достижения для работы с новыми возможностями и новыми алгоритмами.

Упражнения

2.1 Перенести программу из главы 1 на другой язык программирования и вывести на экран упражнения 1.22 для каждой реализации.

программа может иметь различные характеристики производительности в разное время. В-третьих, многие программы чрезвычайно чувствительны ко многим данным, поэтому производительность может измениться в больших пределах в зависимости от них. В-четвертых, многие интересующие нас программы еще не готовы до конца, поэтому для них пока не существует специальных математических результатов. И, наконец, для программ могут совершенно не поддаваться сравнению одна работа эффективнее с другими в разных условиях работы, другая может быть эффективнее при других обстоятельствах.

Исходя из этих факторов, часто вполне возможно предположить, сколько времени займет выполнение определенной программой или же сказать, что если программа будет выполняться эффективнее другой, то в определенных условиях. Кроме того, часто эти знания можно получить, используя специально подобранный набор математических инструментов. Задачей лингвистика алгоритмов является получить как можно больше информации о производительности конкретной программы — использовать эту информацию при необходимости для определения практических рекомендаций. В этой и некоторых других работах будет рассмотрено применение адаптированного миру лингвистика. Чтобы эффективно применить лучшие алгоритмы, иногда просто необходимо посетить такой мир.

Первый шаг в анализе алгоритма состоит в определении абстрактных операций, на которых основан алгоритм, чтобы отделить абстракцию от реализации. Так, например, мы можем рассмотреть код, который для каждой из переменных i выполняет $\text{array}[i] = \text{array}[i] + 1$, от подсчета, сколько раз требуется выполнить для выполнения этого фрагмента кода на данном компьютере. Для определения различного времени выполнения программы для заданного числа элементов требуется обеспечить следующие моменты. Первый из них определить, какой тип алгоритма, последний — свойство компьютера. Также требуется заранее задать алгоритм — алгоритм таким способом, который не зависит от конкретной реализации или от определенного типа компьютера.

Хотя мы рассматриваем алгоритмический анализ, который может использоваться для измерения и принятия, производительность алгоритма абстрактно зависит от нескольких параметров, причем наиболее важным для лингвиста является, как правило, определение сложности. Для нас способ на определенном этапе заключается в использовании некоторых префиксированных символов, доступного не многим реализациям C++, который включает в себя счетчик количества выполненных каждой инструкции для каждой из наиболее часто выполняемых частей программы. В результате не требуется загрузки. Если же, как алгоритм, можно использовать реализацию 1.1, наша реализация может быть построена путем на нескольких абстрактных операциях. В таком случае, алгоритм сводится к определению частоты выполнения каждой функциональной операции. Наш абстрактный тип определяется тем, чтобы попытаться представить различные операции. Будущим следствием, что для каждой программы или реализации можно будет представить полный анализ. Более того, как будет показано ниже, часто можно использовать приближенные алгоритмические результаты в сочетании с лингвистическим языком, чтобы представить результаты достаточно точно.

Кроме того, необходимо научиться гадать и обосновывать их год. который они даст алгоритм. Чаше всего мы будем рассуждать так: из двух возможностей я предполагаю, что одна из них будет предположить, что одна из них является случайным, и гадать, какова вероятность проигрыша, или же мы будем рассуждать, что произвольная точка и гадать, какова вероятность проигрыша. Простейшее описание случайного блуждания: многократные движения вправо или влево. Но для других алгоритмов это может быть правдоподобным и вести к математическим результатам. Многие алгоритмы имеют формулу. Средней может быть математическая функция, но зависящая от данных, для которых используется программа, а также от структуры, которая не имеет значения для практики, но в большинстве случаев эти виды анализа предоставляют полную информацию о произвольности. Например, мы можем сравнить аналитические и эмпирические результаты (см. раздел 2.1). Если они совпадают, мы должны иметь теоретическое обоснование, если же не совпадают, мы сможем узнать больше об алгоритме и о данных, изучая несоответствия.

В предыдущих трех разделах мы кратко рассмотрели математические инструменты, которыми будем пользоваться в течение всей книги. Этот материал написан для предоставления полной первоначальной информации читателю, чтобы избежать с математической основой, или же, что не обязательно подробно проанализировать математические выражения, связанные с произвольностью алгоритмов, и использовать сразу в разделе 2.4. Мы будем использовать материал там, где это указано в книге позже. Математическая основа, в общем, не должна быть сложной и на основе того, как мы будем использовать материал, что не стоит игнорировать тем, кто стремится эффективно использовать компьютер.

В начале, в разделе 2.1, рассмотрим некоторые математические инструменты, которые будут использоваться для анализа характеристик произвольности алгоритмов. Затем, в разделе 2.2, рассмотрим O -нотацию (O -нотация) и выражения, связанные с производными и производными, которые помогут понять детали при математическом анализе. Далее, в разделе 2.3, рассмотрим основные инструменты математического анализа (анализ), основанные на аналитических инструментах, используемых для анализа характеристик произвольности алгоритмов в математических выражениях. Следуя этому курсу, в разделе 2.4 представим понятия, в которых все эти инструменты применяются для анализа некоторых алгоритмов.

Упражнения

- 2.1 Найти выражение вида $a_1 N^4 + a_2 N^3 + a_3 N^2$, которое точно описывает время выполнения программы из упражнения 2.2. Сравнить время выполнения для выражений, с разницей при $N = 10, 100$ и 1000 .
- 2.4 Найти выражение, которое точно описывает время выполнения программы 1 в больших N и M .

2.3 Рост функций

Число элементов алгоритма имеет значение порядка N , который значительно превышает время выполнения. Например N может быть величиной полинома, квадратом функции при сортировке или поиске, количеством элементов в строке или векторе

другой абстрактной мерой размера рассматриваемой задачи: число перемен, от трех пропорционально различным обрабатываемым наборам данных. Когда таких параметров существует более одного (например, M и N в алгоритмах *жадных*), которые обсуждались в разделе 1.5), мы часто сводим их к одному параметру, рассматривая его как функцию других, или рассматривая одновременно только один параметр (считая остальные постоянными), и, таким образом, ограничиваясь рассмотрением только одного параметра N без потери общности. Нашей целью является выражение результатов теоретической программы (как правило, времени выполнения) в зависимости от N с помощью наиболее математических формул, которые как можно проще и строгее описать для большого значения параметров. Алгоритмы в этой книге обычно имеют форму $O(f(N))$, где $f(N)$ — одна из следующих функций:

- 1** Большинство инструкций большинства программ запускается один или несколько раз. Если все инструкции программы обладают таким свойством, мы говорим, что время выполнения программы *линейно*.
- $\log N$** Когда время выполнения программы является *логарифмическим*, программа становится медленнее с ростом N . Такое время выполнения обычно присуще алгоритмам, которые следят за битом задачи в наборе данных, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор. В интересующей нас области мы будем рассматривать время выполнения, являющееся небольшой константой. Основание логарифма является константой, но не имеет значения, когда $N = 1$ (тогда $\log N$ равно 0, если основание равно 10, или логарифм 10, если основание равно 2; когда N равно миллиону, значения $\log N$ только удвоится). При удвоении $\log N$ растет на постоянное количество, а удваивается лишь тогда, когда N увеличивается N^2 .
- N** Когда время выполнения программы является *линейным*, это обычно означает, что каждый элемент задачи подвергается некоторой обработке. Когда N равно миллиону, такого же и требует выполнения. Когда N увеличивается, то же происходит и со временем выполнения. Эта ситуация оптимальна для алгоритма, который должен обработать N элементов (или произведение N значений).
- $N \log N$** Время выполнения пропорционально $N \log N$, возникает тогда, когда алгоритм решает задачу, разбивая ее на несколько частей, каждая из которых является такой же задачей. Из-за отсутствия подходящего приложения (например, *жадных*) мы просто говорим, что время выполнения такого алгоритма равно $N \log N$. Когда N равно 1 миллион, $N \log N$ равно 20 миллионов. Когда N увеличивается, тогда время выполнения увеличивается быстрее, чем линейно.
- N^2** Когда время выполнения алгоритма является *квадратичным*, это означает, что практически все элементы дан относительно небольшого набора. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в целях полного уровня полноты). Когда N равно 1 тысяча, время выполнения равно 1 миллиону. Когда N увеличивается, время выполнения увеличивается быстрее.

Для анализа алгоритма понадобится еще несколько функций. Например, алгоритм с N^2 вводом, имея время выполнения N^3 можно рассмотреть как $N^{3/2}$ алгоритм. Кроме того, некоторые алгоритмы, различающиеся для подбора, имеют время выполнения, пропорциональное $N \log^4 N$. Из табл. 2.1 очевидно, что обе эти функции ближе к $N \log^4 N$, чем N^2 .

Логарифмическая функция играет существенную роль в разработке и анализе алгоритмов, поэтому ее стоит рассмотреть подробнее. Поскольку мы часто не сможем дать аналитический результат, в которых будет логарифмический множитель, мы рассмотрим типичный $\log N$, стоящий алгоритм. Идея состоит в том, чтобы уменьшить количество операций, а не количество элементов. Однако, в описываемом контексте количество операций не является основным логарифмом. В математике логарифмический показатель является натуральным логарифмом (основание $e = 2.71828 \dots$), что равносильно следующему сокращению: $\log_e N = \ln N$. В вычислительной технике чаще всего используется логарифм по основанию 2, поэтому используется сокращение $\log_2 N = \lg N$.

Наименьшее целое число, большее $\lg N$, равно количеству бит, необходимых для представления N в двоичном формате; можно так же назвать это число. Бigger $\lg N$ — это количество бит, необходимых для представления N в десятичном формате. Справедливо $\lceil \lg N \rceil = \lceil \lg N \rceil + 1$.

$$\lg(2^x) = x \quad \text{и} \quad \lg(2^x + 1) = \lg(2^x) + 1 \quad \text{в} \quad x \geq 1.$$

Есть простой способ подсчета минимального целого, большего $\lg N$. Показатель метода для вычисления этой функции:

$$\lg(2^x) = x, \quad x = 1: x \leq N: \lg(2^x) = x + 1 \quad ;$$

Этим утверждением, что $2^x \leq N < 2^{x+1}$, когда x — это наименьшее целое, большее $\lg N$.

Зная мы логарифмический логарифм мы можем его для больших чисел. Например, $\lg \lg 2^{256} = \lg 256 = 8$. Как иллюстрирует данный пример, логарифм мы можем считать не только $\log \log N$ логарифм от логарифма, поскольку оно мало даже для очень больших N .

Кроме того, часто мы сталкиваемся с некоторыми асимптотическими функциями, которые являются частью каждого языка, которые являются частью каждого языка. В табл. 2.3 собраны наиболее распространенные из этих функций, в следующих параграфах мы кратко рассмотрим их, а также некоторые другие важные функции.

Наш алгоритм и на этапе часто сталкиваемся с асимптотическими функциями, поэтому часто требуется специальные функции, переносимые логарифмические числа и др.

$\lfloor x \rfloor$: наибольшее целое, меньшее или равное x

$\lceil x \rceil$: наименьшее целое, большее или равное x

Например, $\lceil \lg N \rceil = \lceil \lg N \rceil + 1$ (для $N \geq 1$), а $\lfloor \lg(N+1) \rfloor$ — это количество бит, необходимых для двоичного представления числа N . Другое важное применение этих функций возникает в том случае, когда необходимо разделить набор из N объектов на две группы. Этого

получить сумму точно, если N является нечетным. Поэтому для точности мы можем считать один подытератор, содержащий $\lfloor N/2 \rfloor$ объектов, а второй, — $\lfloor N/2 \rfloor$ объектов. Если N четно, тогда размеры обоих подытераторов равны ($\lfloor N/2 \rfloor = \lfloor N/2 \rfloor$); если же N нечетно, тогда их размер отличается на единицу ($\lfloor N/2 \rfloor + 1 = \lfloor N/2 \rfloor$). В `C++` можно выразить подобными значениями для функции как подопределения операций над целыми числами (например, если $N \geq 0$, тогда $N/2$ равно $\lfloor N/2 \rfloor$, а $N - (N/2)$ равно $\lfloor N/2 \rfloor + 1$), а при операциях над числами с плавающей точкой можно воспользоваться функциями `floor` и `ceil` из заголовочного файла `math.h`.

При анализе алгоритмов часто встречается дискретизированная версия функции натурального логарифма, называемая гармоническими числами. H_N — гармоническое число определяется выражением

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}.$$

Натуральный логарифм $\ln N$ — это значение, лежащее под кривой между i и N : гармонические числа H_N — это сумма значений функции, которую можно определить, вычислив значения функции $1/i$ для целых чисел от i до N . Изобразим площадь на рис. 2.2. Формула

$$H_N = \ln N + \gamma + \frac{1}{2N} + O(1/N^2)$$

где $\gamma = 0.57721\dots$ (она известна как постоянная Эйлера), дает отличное приближение для H_N . В отличие от $\lceil \lg N \rceil$ и $\lfloor \lg N \rfloor$, для вычисления H_N лучше воспользоваться библиотечной функцией `log`, а не подсчитывать это непосредственно на компьютере.

Таблица 2.3 Специальные функции и константы

В этой таблице собраны математические значения для функций и константы, которые часто встречаются в формулах, описывающих производительность алгоритмов. Формулы для приближенных значений можно найти в библиотечных файлах `math.h` и `mathf.h` стандартных библиотек размещения кода. (Здесь `math.h` — файл заголовка, а `mathf.h` — файл реализации.)

Функция	Название	Дополнительные значения	Приближение
$\sin$	функция синуса	$\lfloor 3.14 \rfloor = 3$	π
$\cos$	функция косинуса	$\lceil 3.14 \rceil = 4$	2π
$\lg N$	десятичный логарифм	$\lg 1024 = 10$	$1.4 \lg N$
F_N	число Фибоначчи	$F_{10} = 55$	$\varphi^N + \varphi^{-N}$
H_N	гармоническое число	$H_{10} = 2.9$	$\ln N + \gamma$
$N!$	факториал	$0! = 1$	$(N/e)^N$
$\lg(N)$		$\lg(1024) = 10$	$N \lg N = 1.44 N$

$\pi = 3.14159\dots$
 $\phi = (1 + \sqrt{5}) / 2 = 1.61803\dots$
 $\lg 2 = 1 / \ln 2 = 1.44269\dots$

$\gamma = 0.57721\dots$
 $\ln 2 = 0.693147\dots$

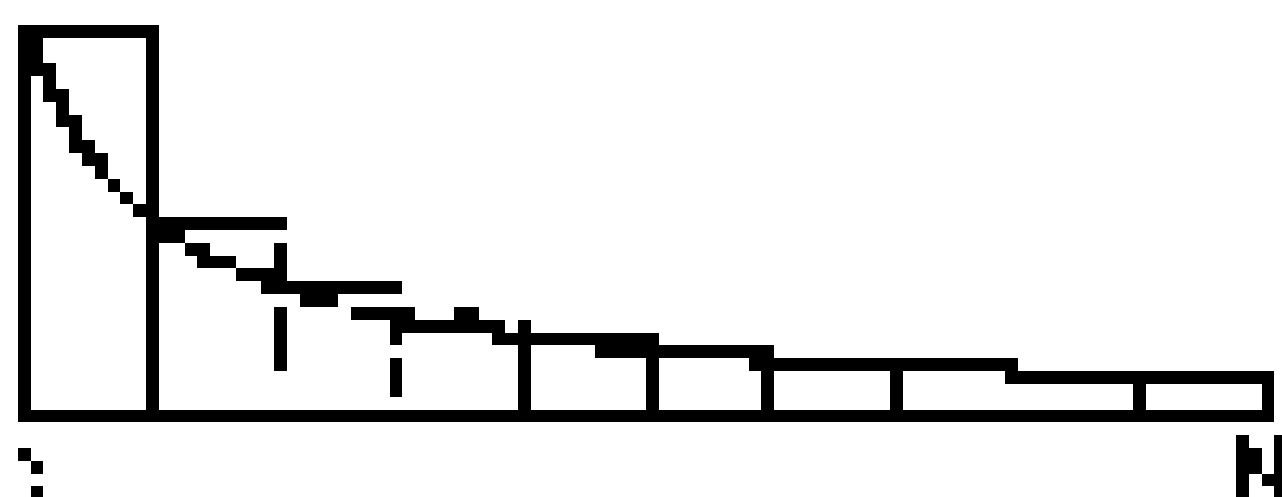


РИСУНОК 2.2
ГАРМОНИЧЕСКИЕ ЧИСЛА

Гармоническое число H_N — это сумма значений функции $1/i$ для целых чисел от i до N . Для оценки H_N можно использовать формулу $H_N = \ln N + \gamma + 1/(2N) + O(1/N^2)$, где $\gamma = 0.57721\dots$ — постоянная Эйлера.

$$H_N = \ln N + \gamma + \frac{1}{2N} + O(1/N^2)$$

Последовательность чисел

0 1 1 2 2 5 5 13 21 34 55 89 144 233 377 ..

определенная формулой

$$F_n = F_{n-1} + F_{n-2}, \text{ где } n \geq 2, \text{ а } F_0 = 0 \text{ и } F_1 = 1$$

известна, как последовательность Фибоначчи. Эти числа имеют множество интересных свойств. Например, отношение двух последовательных чисел приблизительно равно золотому сечению (golden ratio) $\phi = (1 + \sqrt{5}) / 2 = 1.6180339\dots$. После подробной аналитики получается, что F_n равно значению выражения $\phi^n / \sqrt{5}$, округленному до ближайшего целого числа.

При анализе алгоритмов часто встречаются такие функции факториала $N!$. Как и экспоненциальная функция, факториал возникает при любом решении задачи и растет очень быстро, чтобы также представлять практический интерес. Для также возникает при анализе алгоритмов, поскольку представляет собой комбинаторный способ упорядочивания N объектов. Для аппроксимации $N!$ используется формула Стирлинга:

$$\lg N! \sim N \lg N - N \lg e + \lg \sqrt{2\pi N}$$

Например, формула Стирлинга показывает нам, что количество бит в представлении числа $N!$ примерно равно $N \lg N$.

Большинство рассматриваемых в этой книге формул выражаются с помощью нескольких функций, которые мы рассмотрели в этой главе. Однако при анализе алгоритмов может потребоваться множество других специальных функций. Например, классическое биномиальное разложение и разложение Мушсона и ряда возникают лишь при разработке и анализе некоторых фундаментальных комбинаторных алгоритмов, которые будут рассмотрены в главах 14 и 15. Функции, мы рассмотрим здесь, обсуждаются по мере их появления.

Упражнения

- 2.5 Для каких значений N справедливо $10N \lg N > 2N^{2/3}$?
- 2.6 Для каких значений N выражение $N^{1/3}$ имеет значение в пределах от $N(\lg N)^2/2$ до $2N(\lg N)^2$?
- 2.7 Для каких значений N справедливо $2N \lg N - N < N \lg N + 10N$?
- 2.8 Для каких значений N справедливо $\lg N, \lg N^2 > N$?
- 2.9 Докажите, что $\lfloor \lg N \rfloor + 1$ — это количество бит, необходимых для представления числа N в двоичной форме.
- 2.10 Добавьте в табл. 2.2 колонки для $N(2N)^2$ и $N^{3/2}$.
- 2.11 Добавьте в табл. 2.2 строки для 10^5 и 10^6 значений N в колонке.
- 2.12 Изобразите на $C++$ функцию, которая подсчитывает $N!$, используя функцию $\lg$ из стандартной математической библиотеки.
- 2.13 Напишите асимптотическую функцию на $C++$, использующую $\lceil \lg \lg N \rceil$. Не забывайте комментировать функцию.

- 2.14 Сколько цифр в десятичном представлении числа 1 миллион факториал?
- 2.15 Сколько бит в двоичном представлении числа 1000000 ?
- 2.16 Сколько бит в двоичном представлении N ?
- 2.17 Приведите простое выражение для $\lfloor \lg N \rfloor$.
- о 2.18 Приведите конкретные значения N , для которых $\lfloor \lg N \rfloor = i$, где $1 \leq i \leq 10$.
- 2.19 Приведите минимальное значение N , для которого можно решить задачу, требующую выполнения $f(N)$ инструкций, на машине с базисной частотой 10^6 операций в секунду для следующих функций: $f(N) = N^2$, N^3 , $2N \lg N$, $N \lg N$ и $N^2 \lg N$.

2.4 O-нотация

Математическая запись, позволяющая отбросить часть подробностей при анализе алгоритмов, называется *O*-нотацией. Она определена следующим образом.

Определение 2.1 *Скажем, что функция $g(N)$ является $O(f(N))$, если существуют такие константы c_0 и N_0 , что $g(N) \leq c_0 f(N)$ для всех $N \geq N_0$.*

O-нотация применяется для трех основных целей:

- Чтобы устранить ошибку, возникающую при отбрасывании малых слагаемых в математических формулах.
- Чтобы устранить ошибку, возникающую тогда, когда не учитываются те части программы, которые дают малый вклад в суммируемую сумму.
- Чтобы классифицировать алгоритмы согласно порядку роста их общего времени выполнения.

Третье название для *O*-нотации, рассмотренное в разделе 2.3, а две другие кратко обсуждаются ниже.

Положим $c_0 = 1$ и $N_0 = 1$, но выраженные же в *O*-нотации, часто выражают практически идеальную проблему решения. Очевидно, что выражение "алгоритм имеет время выполнения $O(N^2)$ " почти не говорит о времени выполнения при N , меньшем N_0 , и оно может иметь большое значение, чтобы сказать, что эта нежелательной низкой приспособленности. Там хотелось бы использовать алгоритм, время выполнения которого составляет N^2 миллисекунд, а не $10^6 N^2$ секунд, но мы не можем сделать этого выбора на основе *O*-нотации.

Часто результаты математического анализа являются не точными, но приближенными — точная темпическая оценка алгоритма представляет собой неравенство, содержащее последовательность убывающих слагаемых. Так же, как мы интересуемся, в основном, внутренним циклом программы, мы интересуемся только тем самым членом (наибольшим по величине слагаемым) математического выражения. *O*-нотация позволяет рассмотреть главные члены и отбросить меньшие слагаемые при работе с приближенными или математическими выражениями, а также запустить различные выражения, дающие только приближения для стандартных значений.

Некоторые из основных действий, которые мы выполняем при работе с выражениями, содержащими *O*-нотацию, являются следующими выражениями 2.20—2.25. Многие из этих действий интуитивно понятны, и склонные к неточности читатели

можно в интересном выполнить упражнение 2.21. Там требуется доказать общезначимость логичной операции на операторах. Упомянуто то, что из этих упражнений следует, что можно раскрыть скобки в алгебраических выражениях с O -нотацией так, как, если бы они в них не присутствовали, а затем обрабатывать все скобки, как скобки наибольшего. Например, если требуется раскрыть скобки в выражении

$$(A + O(N)) \log A = O(\log A) + O(1),$$

то мы получим вместо сложения

$$N^2 = O(N) + O(N \log A) = O(\log A) + O(N) = O(1).$$

Однако мы можем раскрыть все скобки, кроме наибольшего из них, оставив только

$$N^2 = O(N \log A).$$

То есть, N^2 является порядок определяющей этого выражения, когда A велика. Эти действия интуитивно понятны, но O notation посылает барьеры их в математическую точность. Формула с O -нотацией называется чистым асимптотическим выражением (purely asymptotic expression).

В качестве более наглядного примера предположим (после некоторого математического анализа), что определенная область имеет внутреннюю границу, интервалов N раз, внешнюю секцию, интервалов A раз, и некоторый код сложности N , неопределенный единично. Далее предположим, что мы определили площадь поверхности, показывая различия. Что каждая итерация внутреннего цикла требует a_1 операций, внешняя секция — a_2 операций, а код, не зависящий — a_3 операций. Тогда среди N раз мы имеем, что операции в каждой из них равны

$$2a_1 N N_2 + a_2 N + a_3.$$

Получим для выражения сложения следующую формулу:

$$2a_1 N N_2 + O(N).$$

Более простая формула нужна, поскольку это не следует, что при больших N не следует учитывать различия между a_1 и a_2 для аппроксимации времени выполнения. В общем случае, в точном математическом выражении для времени выполнения может содержаться множество других слагаемых, ряд из которых трудно упростить. O -нотация обобщает способ получения приближенного ответа для больших N без привнесения лишних подробностей.

Применяя данный пример, можно попытаться ($O(N \log N)$), чтобы выразить время выполнения с помощью асимптотической функции. $\log N$. Например O notation, при определенных выражениях из табл. 2.3 можно записать как $N_2 = N \log N + O(1)$. Также известно, что $a_1 N \log N = O(N)$ — это асимптотическое выражение для общего времени выполнения алгоритма. То есть, при больших N оно будет близко к легко подсчитываемому выражению $2a_1 N \log N$. Постоянный множитель a_1 не важен, который требуется для того, чтобы иметь интуитивный внутренний цикл.

Более того, нам не нужно иметь значения a_1 , чтобы показать, что время выполнения для таких размеров N будет более, чем для всех размеров N' для больших N' , поскольку

$$\frac{2n_0(2N) \ln(2N) + O(2N)}{2n_0 \ln N + O(N)} = \frac{2 \ln(2N) + O(1)}{\ln N + O(1)} = 2 + O\left(\frac{1}{\log N}\right)$$

Таким образом, асимптотическая формула получается нами из этой точной предельной, не пытаясь и попытаться решить задачу уже сейчас. Отметим, что такое предсказание не было бы возможным, если бы в O -аппроксимациях была другая степень для главного члена.

Намеченный способ рассуждений позволяет в дальнейшем только главным членом при сравнении или предсказании времени выполнения операций. Зачастую мы тем считаемся, что мы полностью константно-презентных операций и хотим сравнить с тем же главным членом, подразумеваем, что точный анализ somehow приделного мы не можем себе позволить, если требуется.

Когда функция $f(N)$ является асимптотически более быстрой, чем другая функция $g(N)$ (т.е. $g(N)/f(N) \rightarrow 0$, когда $N \rightarrow \infty$), мы никогда и нигде не используем термин «быстро» (формально говоря, «быстро» $f(N)$, подразумеваем $f(N) = O(g(N))$). То, что, кажется, является в математической теории, комбинаторике, алгоритмах, так как так больше интересует простота алгоритма, а не математическая сложность. В таких случаях мы можем быть уверены в том, что при больших значениях N (еще даже не при всех значениях) исследуемая величина будет больше по величине, чем $f(N)$. Например, даже если мы знаем, что величина равна $N(N-1)/2$, мы можем сказать о ней, что $N^2/2$. Такая способ вычисления результатов является достаточно быстрым, чем переборная и точная результаты, в частности от простого значения всего лишь на $O(1)$ процентов для, например, $N = 1000$. Потеря точности в данном случае является незначительной, так при рассмотрении сложности $O(f(N))$. Наша цель состоит в том, чтобы быть достаточно точными, и мы можем при этом использовать положительности алгоритмов.

В похожем ключе мы можем сказать, что время выполнения алгоритма пропорционально $f(N)$, т.е. можно доказать, что оно равно $c \cdot f(N)$, где $c(N)$ асимптотически много по сравнению с $f(N)$. В рамках такого подхода мы можем предсказать время выполнения для $2N$, если оно известно для N , как в примере, который обсуждался выше. Для $N = 2N$ получаются значения коэффициентов для таких простых элементарных функций, которые часто применяются по тем же самым алгоритмам. В соответствии с эмпирическими наблюдениями (см. рисунок 2.1) данный подход особенно

1	нет значения
$\lg N$	небольшой рост
N	увеличение
$N \lg N$	чуть больше, чем удвоение
N^2	умножить на 2
N^3	умножить на 4
N^4	умножить на 8
2^N	в квадрате

РИСУНОК 2.1 ВЛИЯНИЕ УДВОЕНИЯ РАЗМЕРОВ ЗАДАЧИ НА ВРЕМЯ ВЫПОЛНЕНИЯ.

Пропорциональное увеличение размера задачи при увеличении сложности приводит к тому, что время выполнения увеличивается. Если мы увеличим размер задачи в 2 раза, то время выполнения увеличится в 2 раза, если же увеличим в 4 раза, то время выполнения увеличится в 4 раза, и т.д. Это означает, что время выполнения задачи увеличивается в зависимости от сложности задачи. Например, если задача имеет сложность $O(N^2)$, то время выполнения увеличится в 4 раза, если же задача имеет сложность $O(N^3)$, то время выполнения увеличится в 8 раз. Это означает, что время выполнения задачи увеличивается в зависимости от сложности задачи.

инструмент, связанное со спецификой алгоритма, используется при нахождении оптимального решения, здесь же алгоритм удовлетворяется только первым найденным решением.

Формула 2.1 В рекурсивной программе, где при каждой итерации цикла вычисляется значение C_N на основе, возникающих предыдущих соотношений:

$$C_N = C_{N-1} + N, \text{ где } N \geq 2 \text{ и } C_1 = 1$$

Решение: Сдвигая индекс $N \geq 2$. Для решения рекурсии ее можно раскрыть, приняв за основу следующее соотношение:

$$\begin{aligned} C_N &= C_{N-1} + N \\ &= C_{N-2} + (N-1) + N \\ &= C_{N-3} + (N-2) + (N-1) + N \\ &\vdots \end{aligned}$$

Продолжая таким же образом, можно получить

$$\begin{aligned} C_N &= C_1 + 2 + \dots + (N-2) + (N-1) + N \\ &= 1 + 2 + \dots + (N-2) + (N-1) + N \\ &= \frac{N(N+1)}{2}. \end{aligned}$$

Подсчет суммы $1 + 2 + \dots + (N-2) + (N-1) + N$ элементарно: прибавим к сумме ее же, но в обратном порядке. Результатом суммы — удвоенный искомого результат — будет сумма из N слагаемых, каждое из которых равно $N + 1$.

Формула 2.2 В рекурсивной программе, где на каждой итерации количество подэлементов вдвое возрастает, возникающая следующая рекуррентная соотношение:

$$C_N = C_{N/2} + 1, \text{ где } N \geq 2 \text{ и } C_1 = 1$$

Решение: C_N порядка $\lg N$. Из заданного следует, что это уравнение бесмысленно из-за некорректного случая, когда N нецелое или же нечетное число, что $N/2$ — целочисленное деление. Сейчас предположим, что $N = 2^n$, чтобы рекурсия была корректно определена. Заменяем, что $n = \lg N$. Тогда рекурсия еще проще раскрыть, чем в предыдущем случае:

$$\begin{aligned} C_N &= C_{N/2} + 1 \\ &= C_{N/4} + 1 + 1 \\ &= C_{N/8} + 1 + 1 \\ &\vdots \\ &= C_{N/2^n} + n \\ &= n + 1. \end{aligned}$$

Точное решение для любого N зависит от интерпретации $N/2$. Если $N/2$ представляется $\lfloor N/2 \rfloor$, тогда существует очень простое решение: C_N — это количество бит в двоичном представлении числа N , т.е. по определению $\lfloor \lg N \rfloor + 1$. Этот по-

то, что мы планируем сделать из того, что операция сдвигания правого бита в двоичном представлении любого числа $N > 0$ эквивалентна его $n \lfloor N/2 \rfloor$ (см. рис. 2.6).

Формула 2.3 — рекурсивная программа, где количество вызовов уменьшается вдвое, но необходимо проверить каждый элемент, возникающий в следующем рекурсивном соотношении:

$$C_N = C_{\lfloor N/2 \rfloor} + N, \text{ где } N \geq 2 \text{ и } C_1 = 0.$$

Решение: C_N порядка $2N$. Рекурсия завершается в форму $N + N/2 + N/4 + N/8 + \dots$ (Как и формула 2.2, рекурсивные соотношения выполняются точно только в том случае, если N является степенью числа 2). Если дана произвольная бинарная строка, то формула просто геометрической прогрессии равна $2N$. Поскольку мы используем целочисленные деления и остаемся в рамках i , данное значение является приближением к точному ответу. В этом решении используются свойства двоичного представления числа N .

Формула 2.4 — рекурсивная программа, которая может выполнить линейный проход по всем битам строки, если разделение нулей для положительных возникает следующее рекурсивное соотношение:

$$C_N = 2C_{\lfloor N/2 \rfloor} + N, \text{ где } N \geq 2 \text{ и } C_1 = 0.$$

Решение: C_N порядка $N \lg N$. На это решение можно посмотреть иначе, чем на остальные из приведенных ранее, поскольку эта рекурсия используется в целом семействе алгоритмов "разделяй и властвуй".

$$C_{2^k} = 2C_{2^{k-1}} + 2^k$$

$$\begin{aligned} C_{2^k} &= \frac{C_{2^{k-1}}}{2^{k-1}} + 1 \\ &= \frac{C_{2^{k-2}}}{2^{k-2}} + 1 + 1 \\ &\vdots \\ &= k \end{aligned}$$

Мы находим решение почти так же, как это было сделано в формуле 2.2, но в данном случае применяем на каждом шаге — делением обеих частей уравнения на 2^k , которая позволяет раскрыть рекурсию.

N	$C(N)$	$\lfloor \lg N \rfloor + 1$
1	0	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4
10	1010	4
11	1011	4
12	1100	4
13	1101	4
14	1110	4
15	1111	4

РИСУНОК 2.6 ЦЕЛОЧИСЛЕННЫЕ ФУНКЦИИ И ДВОИЧНЫЕ ПРЕДСТАВЛЕНИЯ

Размещенная бинарная цифровая строка N (в строке, помечен $\lfloor N/2 \rfloor$) также отображается строкой битов N (см. строку битов). Минимум представленных битов N по строке битов, когда представляется число $\lfloor N/2 \rfloor$. Минимум $\lfloor \lg N \rfloor + 1$ минимально бит в строке битов, когда N является наименьшим числом N , который удовлетворяет формуле 2.2 и строке битов $\lfloor N/2 \rfloor$.

Формула 2.5 Н рекурсивная прогрессия, которая разбивает под индекс, а затем производит постоянное количество других операций (см. пазл 2) возникая из-за двойной рекурсии.

$$C_N = 2C_{N/2} + 1, \text{ где } N \geq 2 \text{ и } C_1 = 1.$$

Решение: C_N порядка $2N$. Это решение можно считать так же, как и решение формулы 2.4.

С помощью приведенных методов можно решить различные варианты подобных формул, включая другие известные условия или различные отличия в дополнительном шаге. Но необходимо помнить, что некоторые рекурсии, включаясь в себя, могут иметь гораздо более сложные решения. Существует множество специальных обобщений методов для математически строгих решений таких уравнений (см. раздел 10.2.2). Несколько сложных рекуррентных соотношений встречаются в последующих главах, где и будут обсуждаться их решения.

Упражнения

- р 2.33 Составьте таблицу значений C_N , заданных формулой 2.3 для $1 \leq N \leq 32$, считая, что $N/2$ означает $\lfloor N/2 \rfloor$.
- р 2.34 Выполните упражнение 2.31, но считая, что $N/2$ означает $\lceil N/2 \rceil$.
- р 2.35 Выполните упражнение 2.4 для формулы 2.4.

- с 2.36 Покажите, что f_N и конформная одно связная область U_N являются

$$C_N = C_{N-1} - f_N, \text{ где } N \geq 1 \text{ и } 0 \leq C_N \leq c \text{ при } N \leq n,$$

где c и t — целые числа. Покажите, что C_N произвольным $\lfloor \lg N \rfloor$

- 2.37 Из формулы 2.3 и покажите обобщенные версии формул 2.2–2.5, аналогичные обобщенной версии формулы 2.2 и упражнение 2.36.

2.38 Составьте таблицу значений C_N , заданных формулой 2.4 при $1 \leq N \leq 32$ для трех случаев: (i) $N/2$ означает $\lfloor N/2 \rfloor$, (ii) $N/2$ означает $\lceil N/2 \rceil$, (iii) $2C_{N/2}$ равно $C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil}$.

2.39 Решите формулу 2.4 для случая, когда $N/2$ означает $\lfloor N/2 \rfloor$, используя алгоритм списа двоякому присваиванию числа N , как это было сделано в доказательстве формулы 2.2. Покажите: Рассмотрите все числа, меньшие N .

2.40 Решите рекурсию

$$C_N = C_{N/2} + N^2 \text{ при } N \geq 2 \text{ и } C_1 = 1,$$

когда N является степенью числа 2.

2.41 Решите рекурсию

$$C_N = C_{N/2} + 1 \text{ при } N \geq 2 \text{ и } C_1 = 0,$$

когда N является степенью числа 2.

- с 2.42 Решите рекурсию

$$C_N = \alpha C_{N/2} \text{ при } N \geq 2 \text{ и } C_1 = 1,$$

когда N является степенью числа 2.

с 2.43 Решить рекурсию

$$C_N = (C_{N-1})^2 \text{ для } N \geq 2 \text{ и } C_1 = 1,$$

когда N является степенью числа 2

■ 2.44 Решить рекурсию

$$C_N = \left(2 + \frac{1}{p(N)}\right) C_{N-1} \text{ при } N \geq 2 \text{ и } C_1 = 1,$$

когда N является степенью числа 2.

■ 2.45 Рассмотрите семейство рекурсий, заданных формулой 2.1, где C_N может означать $\lfloor N/2 \rfloor$ или $\lfloor N/2 \rfloor + 1$, а единственное требование заключается в том, чтобы рекурсия была корректной при $N \geq n_0$ и $C_{n_0} = O(1)$ при $N \leq n_0$. Покажите, что решением всех таких рекурсий является формула $\lg N = O(1)$.

◆◆ 2.46 Перенесите обобщенные рекурсии на решения, как и упражнения 2.45, на формулы 2.2–2.5.

2.6 Примеры алгоритмического анализа

Вокруг поиска инструментов, о которых было рассказано в трех предыдущих разделах, мы рассмотрим теперь последовательного и бинарного поиска — двух основных алгоритмов для выполнения того, что мы назвали независимостью объектов в заданном наборе объектов. Далее также пойдет речь о характеристике того, как можно сравнивать алгоритмы, а не подробное описание самих алгоритмов. Для простоты предположим, что все рассматриваемые объекты являются целыми числами. Конкретные приложения рассматриваются в главах 12–16. Известные термины алгоритмов, которые встречаются сейчас, не только имеют строгое математическое определение, но и имеют широкое применение.

Например, представим себе компьютер, обрабатывающий кредитные карты и имеющий N записей или заданных кредитных записей. При этом компьютер необходимо пропарить, так как среди N транзакций какого-либо типа N транзакций некорректны. Для обеспечения надежности N должно быть достаточно большим, например, $N = 10^7$, а $N^2 = 10^{14}$ транзакций (разрядка $10^7 - 10^8$). Цель анализа заключается в приблизительной оценке времени выполнения алгоритма, когда параметр принимает значения из указанного диапазона.

Примером 2.1 представляет прямое решение проблемы поиска. Она оформлена как функция C++, обратившаяся имеет форму (см. главу 5), для совместности с другими вариантами кода для этой задачи, которая не нуждается в члене $+$. Однако необходимо отметить и другие программы для выполнения алгоритма: мы сохраняем все объекты в массиве, затем для каждой транзакции мы пробегаем весь массив и делаем вывод о том, содержится ли в нем нужный номер.

Для анализа алгоритма, прежде всего, отметим, что время выполнения зависит от того, насколько ли транзакция объект в массиве. Если поиск не является успешным, мы можем определить это, только проверить все N объектов, но успешный поиск может завершиться на первом, втором или любом другом объекте.

Программа 2.1 Последовательный поиск

Данная функция проверяет, находится ли число v среди элементов массива $a[1], a[2], \dots, a[n]$ путем последовательного сравнения с каждым элементом, начиная с начала. Если v действительно содержится в массиве, функция возвращает индекс i , такой, что $a[i] = v$. Иначе она возвращает нулевой элемент массива, свидетельствующий об отсутствии v .

```
int search(int a[], int v, int l, int r)
{
    for (int i = l; i <= r; i++)
        if (v == a[i]) return i;
    return -1;
}
```

Важному времени выполнения зависит от данных. Если бы все элементы массива превосходили v , то время, которое потребуется в данной функции, было бы огромно; если бы они все были меньше v , то время было бы небольшим. В разделе 2.2 мы обсудим различия между сложностью алгоритмов при выполнении и сложностью реализации. В данном случае, лучшее, что мы можем гарантировать, будет тем, что время не более N операций.

Однако, чтобы сказать какой-либо программе, необходимо высказать предположение о данных. В данном случае предположим, что все числа выбраны случайным образом. На это следует, например, что каждое число в таблице может с одинаковой вероятностью оказаться предметом поиска. После некоторых размышлений мы приходим к выводу, что наличие его свойства поиска является наиболее важным, потому что среднее случайно выбранное число может вообще не проходить успешного поиска (см. упражнение 2.48). В некоторых приложениях количество трафика с успешными поисками может быть огромно, — другая причина — поиск. Чтобы использовать модель случайных приложений, мы рассмотрим задачу для случая успешных и неуспешных поисков и представим время на поиск как функцию от длины массива. Это важный моментом эффективной работы является разработка алгоритма поиска приложения. Более аналитические результаты будут зависеть от более сложной модели. Кроме того, они обобщают нас на формулы, необходимые для выбора различных алгоритмов для разных приложений на основании этого анализа.

Лемма 2.1 *Последовательный поиск требует $\Theta(N)$ операций на каждом неуспешном поиске по среднему массиву $N/2$ элементов, каждый раз проверяя элемент.*

Если каждое число в таблице с равной вероятностью может быть объектом поиска, тогда

$$1 + 2 + \dots + (N-1) = (N-1)N/2$$

является средней длиной поиска.

Лемма 2.1 подразумевает, что время выполнения программы 2.1 при выполнении $\Theta(N)$ операций зависит от предположения, что средняя длина поиска имеет постоянство. Таким образом, например, можно ожидать, что если удвоить количество объектов в массиве, то время, необходимое для поиска, также удвоится.

Последовательная техника в неудачном случае можно ускорить, если углубить поиск в глубину. Сортировка чисел в таблице является предельно рассматриваемой задачей [1]. Поскольку алгоритмы, которые мы рассмотрели, выполняют эту задачу за время, пропорциональное $N \log N$, которое эквивалентно по сравнению с линейным поиском по числам M . В упорядоченной таблице можно провести поиск сразу по десятичным числам, большим, чем N . Такое изменение уменьшает время выполнения поиска до $N/2$ чисел, которые необходимо и следует проверить при увеличении поиска. Время для такого случая сокращается до времени для линейного поиска.

Лемма 2.2 Алгоритм последовательного поиска в упорядоченной таблице проверки N чисел для каждого числа и время поиска в таблице $N/2$ чисел в среднем.

Этот же шаг необходимо определить, чтобы непрерывно поиска. Этот результат следует из предположения, что поиск может с равной вероятностью закончиться в любой из $N + 1$ начальной, поэтому в N числах в таблице, а это неэффективно приводит к выражению

$$(1 + 2 + \dots + N + N)/N = (N + 1)/2.$$

Прямой непрерывный поиск, который указывается для поиска N -ой элемент в таблице, такая же N .

Другой способ выразить результат леммы 2.2 — это сказать, что время выполнения последовательного поиска пропорционально N^2 для N элементов в таблице и другим значениям. Если мы удвоим или удвоим количество элементов в таблице, то время не изменится удвоится, если мы удвоим обе величины одновременно, тогда время выполнения возрастает в 4 раза. Этот результат следует из того, что данный метод не подходит для очень больших таблиц. Если для проверки одного числа требуется n микросекунд, а $N = 10^7$ и $N = 10^8$, тогда время выполнения для всех чисел будет, по крайней мере, $10^7 \times 10^7$ секунд, или, согласно рис. 2.1, около 16 лет, что совершенно не приемлемо.

Программа 2.2 Бинарный поиск

Эта программа обладает такой же функцией полезности, что и программа 2.1, но имеет другую структуру.

```
int search(int a[], int v, int l, int r)
{
    while (l <= r)
    {
        int m = (l+r)/2;
        if (v == a[m]) return m;
        if (v < a[m]) r = m-1; else l = m+1;
    }
    return -1;
}
```

Программа 2.2 представляет собой эффективное решение проблемы поиска методом прямого поиска эффективный, чем последовательный поиск. Он основан на предположении, что для таблицы упорядочены, и мы можем в любой момент узнать, что сравнение некоторого значения с числом из середины таблицы. Если оно равно, то

Бинарный и последовательный поиск для M объектов в таблице размером N (отличаясь от бинарного поиска лишь сортировкой таблицы) при различных значениях M и N . Здесь мы не будем рассматривать различные программы и детали реализации, поскольку подробно рассмотрены в главах 9 и 11, а, кроме того, неплатящие характеристики и прочие функции и другие детали создания программ из элементов, включая и функции `sort`, объяснены в главе 3. В данный момент мы просто подчеркнем, что проведение эмпирического тестирования — это несомненно важная часть работы эффективного алгоритма.

Таблица 2.4 подтверждает наше наблюдение, что функциональный рост времени выполнения позволяет предсказать простоту реализации в случае больших значений параметров на основе эмпирических данных работы алгоритма при малых значениях. Эмпирические математические анализ и эмпирическое изучение убедительно доказывают, что последовательный алгоритм, безусловно, является бинарным поиском.

Таблица 2.4 Эмпирическое изучение последовательного и бинарного поиска

Главная цель эмпирического изучения выполнения алгоритмов — еще и практически результаты, что в случае M объектов в таблице из N объектов поиск последовательного поиска требует времени MN , а время бинарного поиска — $M \lg N$. При удвоении M время последовательного поиска также удваивается, а время бинарного поиска почти не изменяется. Последовательный поиск непрактичен для случая больших M при малых N , а бинарный поиск является для вполне быстрым даже для огромных таблиц.

N	M = 1000		M = 10000		M = 100000	
	S	B	S	B	S	B
128	1	1	2	2	133	23
256	2	0	25	2	251	22
512	5	0	48	2	497	23
1024	12	0	128	2	1275	25
2048	26	1	267	0		28
4096	53	0	532	2		33
8192	104	1	1057	2		33
16384	268	1		2		35
32768	537	0		4		39
65536	1268	1		8		47

Обозначения:

S — последовательный поиск (по рядам 2^7)

B — бинарный поиск (по рядам 2^7)

Данный пример является прототипом общего подхода к сравнению алгоритмов. Математический анализ используется для анализа системы, в которой алгоритм представляет абстрактные операции. Затем на основе этих результатов выводится функциональный формат преобразования, который позволяет проанализировать и расширить эмпирические данные. По мере нахождения алгоритмических решений все более и более сложных вычислительных задач и установления новых математических знаний на практической основе, мы будем обращаться к литературе и математическим результатам, чтобы основательно познакомиться с данными анализа алгоритмов. Здесь мы рассмотрим производные математическое и эмпирическое изучение всех алгоритмов. В данной главе мы также определим наиболее общие практические характеристики производимости. При этом, в дальнейшем, всегда можно разрабатывать научное исследование, необходимое для правильного выбора алгоритмов для данных приложений.

Упражнения

- 2.47 Напишите функцию прямого сравнения, используемую программой 2.1, если $12M$ поисков прошли успешно, а $0 \leq n \leq 1$.
- 2.48 Определите вероятность того, что хотя бы одно из M случайных действительных чисел будет содержаться в наборе из N чисел, при $M = 10, 100, 1000$ и $N = 10^2, 10^3, 10^4, 10^5$.
- 2.49 Напишите программу, которая генерирует M случайных чисел и помещает их в массив, затем подсчитывает количество N парных чисел, которые совпадают с одним из чисел массива, используя линейный поиск. Запустите программу при $M = 10, 100, 1000$ и $N = 10, 100, 1000$.
- 2.50 Сформулируйте и докажите лемму, аналогичную лемме 2.3 для бинарного поиска.

2.7 Гарантии, предсказания и ограничения

Прямое выполнение возможности алгоритмов зависит от входных данных. Обычной целью при анализе алгоритмов является получение каких-либо образцов эффективности. Необходимо иметь возможность сказать, что то или иное свойство данных программ, что может не зависеть от входных данных, так как в общем случае невозможно, какими будут входные данные для каждой новой задачи программы. Примеры из раздела 2.6 иллюстрируют два основных метода, которые применяются в этом случае: анализ эмпирической производительности и анализ средней производительности.

Из-за неопределенности производительности достаточно проблематично, поскольку это потребует определения того, каковы возможности программ. Мы рассмотрим, что такое, в которой записаны определенные абстрактные операции, некие, как определенная функция от количества входных элементов $0 \leq n \leq 1000$, каковы значения функции. Например, лемма 2.1 — пример такой гарантии для бинарного поиска, так же как лемма 1.3 — для простоты быстрого объединения. Если говорить о другом анализе, как в случае с бинарным поиском, тогда это будет интуитивная оценка, поскольку удалось устранить ситуацию, когда программа

работает медленно. Поэтому программы с хорошей характеризацией в случае низкой проницательности и являются основной целью разработки алгоритмов.

Однако при анализе низкой производительности существуют некоторые трудности. Для определенных алгоритмов может существовать большая разница между временем, необходимым для решения задачи в идеальных условиях входных данных, и временем, необходимым в случае данных, которые встречаются на практике. Например, быстрое объединение в случае низкой производительности требует времени выполнения, пропорционального N^2 , и пропорционального $\log N$ для обычных данных. Часто не удается доказать, что существуют входные данные, для которых время выполнения алгоритма достигает определенного предельного значения; можно лишь доказать, что время выполнения будет ниже этого предела. Кроме того, для некоторых задач алгоритмы с хорошей характеристикой низкой производительности гораздо сложнее, чем другие алгоритмы для этих задач. Иногда возникает ситуация, когда алгоритм с хорошей характеристикой низкой производительности при работе с практически любыми данными оказывается медленнее, чем более простой алгоритм, хотя же при малом объеме данных работает быстрее. Поэтому табличные значения времени для достижения хорошей характеристики низкой производительности для многих приложений другие соображения — полнота и надежность — являются более важными, чем скорость. Данные в случае низкой производительности. Например, как было показано в главе 1, повышение быстрого объединения в среднем пути обеспечивает лучшую производительность алгоритма, чем повышение быстрого объединения, но для типовых данных, встречающихся на практике, алгоритм может почти вдвое сократить время выполнения.

Изучение средней производительности и алгоритмов более сложными, поскольку оно позволяет делать предположение о времени выполнения программ. В простейшем случае можно точно характеризовать некоторые данные алгоритмы, например, алгоритм сортировки может выполняться над массивом из N случайных чисел или геометрический алгоритм может характеризовать набор из N случайных точек на плоскости. С некоторыми из них можно работать, если только ряд инструкций будет выполняться каждой инструкцией, и вложить время выполнения программы, умножив частоту выполнения каждой инструкции на время ее выполнения и суммируя по всем инструкциям.

Однако и в анализе средней производительности существуют трудности. Во-первых, любая модель может неточно характеризовать типичные данные, встречающиеся на практике, или же ее теория может работать только с идеальными данными, которые не существуют. Мыслимо будет изображать процесс выполнения таких идеальных данных, как "случайная упорядоченная пара" для алгоритма сортировки или "случайный набор точек" для геометрического алгоритма, и для таких идеальных данных получить математические результаты, которые будут точными предположениями о производительности программ в реальных приложениях. Но как можно анализировать типичные данные для программ, которые обрабатывают текст на английских языках? Даже для алгоритмов сортировки в определенных приложениях рассматриваются другие методы кроме идеальных случайных упорядоченных данных. Во-вторых, анализ может требовать глубоких математических характеристик. Например, анализ времени средней производительности для алгоритмов объединения дает значение $\log N$, что не-

при таких результатах обычно исходит из предположения, что будем исследовать разницу между некоторыми конкретными проблемами, а также, при необходимости, будем считать, что лучшие результаты (к счастью, почти все результаты алгоритмов можно найти в исследованиях других авторов). В третьих, знания о среднем типичном выполнении может оказаться недостаточным: может потребоваться знание о том, как другие факторы (различия во времени выполнения, выходы которых могут оказаться еще более трудными). В частности, нас будет интересовать вероятность того, что алгоритм окажется значительно медленнее, нежели ожидается.

Во многих случаях на первом этапе можно отбросить превращение случайности в детерминизм. Например, если мы случайным образом "забиваем" массив для сортировки, тогда допущение о том, что элементы массива падают в случайном порядке, будет выдвинуто. Для задачи сортировки, вызываемая сложность алгоритма, знания о средней практической сложности приводят к сложнейшему процессу выполнения в строгом техническом смысле. Более того, часто можно доказать, что вероятность того, что такой алгоритм будет медленным, пренебрежимо мала. Прямые доказательства приводят к себе быстрого сортировки (см. главу 9), алгоритмизированные ВЭМ (см. главу 13) и алгоритмы (см. главу 14).

Вызываемая сложность — это выражение анализа алгоритма, которое задано функцией от произвольного выражения, которое возникнет при анализе алгоритма. Оценка здесь заключается в определении времени выполнения для каждой пропускной способности (или) алгоритма для данной задачи в точности до постоянного множителя. Для функции называется сложностью задачи.

Анализ практической производительности: использование O нотации освобождает нас от необходимости писать в математическом выражении определенную машину. Выражение "время выполнения алгоритма равно $O(f(N))$ " не зависит от выбора модели и полезно для распределения алгоритмов по категориям в зависимости от входных данных и другой информации. А также объектом анализа является алгоритм от какой-либо определенной сложности. В анализе мы, как правило, абстрагируем постоянные коэффициенты. В большинстве случаев, если мы хотим найти, чему пропорционально время выполнения алгоритма, — N или N^2 , — не имеет значения, где будет выполняться алгоритм — на небольшом компьютере или на суперкомпьютере. Более того, не имеет значения даже то, хорошо или плохо реализован внутренний цикл алгоритма.

Когда можно сказать, что время выполнения алгоритма в случае практической производительности равно $O(f(N))$, то говорят, что $f(N)$ является *хорошей функцией сложности* для данной сложности, время выполнения лучшего алгоритма не больше, чем время любого другого алгоритма для данной задачи.

Мы постоянно стремимся улучшить алгоритмы, но, в конце концов, достигнем точки, где никакое изменение не может снизить время выполнения. Для каждой задачи мы заинтересованы в том, чтобы найти, где необходимо остановиться и улучшить алгоритм, т.е. мы ищем минимальную сложность. Для многих задач можно доказать, что любой алгоритм решения задачи должен использовать определенное количество функциональных операций. Доказательство существования минимальной тра-

яции — сложное дело, связанное с тщательным созданием модели машины и распределением численности технологических конструкций различных машин, которые также решены для любого алгоритма. Мы редко затрагиваемся по трем о последствии тех или иных, но они представляют собой самостоятельные формы для разработки алгоритма, и о них будет упомянуто при необходимости.

Когда изучение сложности задачи показывает, что первая и вторая границы алгоритма совпадают, тогда можно быть уверенным в том, что нет смысла пытаться разработать алгоритм, который был бы фундаментально быстрее, чем уже известный алгоритм. В этом случае можно констатировать заключение на основании. Например, бинарный поиск является оптимальным в том смысле, что никакой алгоритм, состоящий только из операций сложения, сложения и деления, не может быть быстрее бинарного поиска.

Первая и вторая границы сложности относятся также и для алгоритмов на отладку, рассмотренных в книге. В 1975 г. Терьян (Терьян) показал, что алгоритм внешнего быстрого обхода дерева требует сложения по веткам, чем $(N \log^2 K)$ указанным в случае простой простоты, а не любой другой с указанным сложением. Более того, на основании чужих указаний в случае простоты простоты для некоторых простых данных. Другими словами, на основе сложения сложения более сложного, которое эквивалентно бы решению задачи линейных чисел операций $I = \Delta(I)$. На практике это различие очень мало, поскольку $\log^2 K$ очень мало, но не менее, поскольку сложение является сложным алгоритмом. Этот результат был важен для исследования и решения проблемы времени, и поэтому Терьян в своей книге выполнил условия исследователей на другие задачи. В частности, которые являются, они являются функциями сложности сложной функции $\log^2$, поскольку они являются той же.

Многие алгоритмы в этой книге были созданы (или глубоко исследованы) в результате исследования на простоту сложности сложной сложности, чтобы исследовать их. Но именно на основе этих исследований мы рекомендуем читателю не тех алгоритмов, которые являются более.

Но все алгоритмы стоит так же тщательно рассмотреть; более того, при создании алгоритмов желательно работать с приближенными признаками простоты, чтобы не потерять процесс исследования сложности. Но здесь тоже как разработка алгоритма становится более утомительной, то же самое можно проанализировать в отношении, приходя к более сложным математическим инструментам. Часть работы создания алгоритма привела к созданию сложности сложности, которые, в свою очередь, ведут к таким теоретическим алгоритмам, которые являются от каких-либо практических приложений. Успешная разработка сложности сложности, что приводит к практическим результатам. Такие предположения ведут к неприятию сложности сложности. С другой стороны, вычислительная сложность — это мощный инструмент, который позволяет определить границы простоты сложности при разработке алгоритма и может использоваться в качестве инструмента для создания промежуточных между простотой и сложностью.

В этой книге мы придерживаемся точки зрения, что разработка алгоритма, тщательное его рассмотрение, математический анализ, теоретические обоснования и экспериментальное выявление места имеют важный вклад в создание эффективных программ. Мы стремимся получить информацию о свойствах программ прежде, чем мы их адаптируем, затем модифицировать их или разработать новые программы на основе полученной информации. Мы не хотим возможности проецировать теоретическое тестирование к анализу каждого алгоритма, допускаемого в программной среде на любой возможной машине, но мы можем воспользоваться реализованным алгоритмом, эффективность которого известна, а затем улучшить и оптимизировать его, если требуется высокая производительность. По ходу книги при необходимости будут достаточно подробно рассмотрены различные наиболее важные методы.

Упражнения

§ 2.51 [Упражнение]. — о переменная сложность одной задачи равна $M \log N$, а другой — N^2 . Что даст вам утверждение, можно переписать об относительной сложности, если $M \gg N$, как одна функция, которая решает эти задачи?

Ссылки к части 1

Существует множество важных учебников по программной структуре. Стандартным источником C++ является книга Стратусона [Straterson], а лучшим источником знаний о C++ программами программы, написанные в котором являются полными и прямыми также для C++ и тем самым в том же духе, что и программы в этой книге, служит книга Керниана и Рина [Kernighan, Ritchie] о языке C.

Несколько вариантов алгоритмов для задачи `mergeSort` на языке C собраны и объяснены в статье Вэл Лепена и Тараса ван Леева [Lepen, Taras].

Книга Веллера [Welsh] посвящена в том же духе, что и эта книга, но она охватывает, несколько нежелательной производительности и использование различных подходов при разработке и реализации алгоритмов для решения различных интересных задач.

Классическая работа по анализу алгоритмов, основанная на нечетких научных данных, никакой пронаблюденности — это книга Ада Хопкрофта и Уаймана [Hopcroft, Ullman]. Книга Кнута [Knuth] покрывает явную среднюю производительность более полно и является оригинальным источником определений скорости и эффективности. Книга Ринна, Крайла [Rinne, Krayla] и Кермана, Лейзера, Ринна [Kernighan, Leiser, Rinne] является более современными работами. Оба они включают обширный список ссылок на исследовательскую литературу.

Книга Глоу, Кнута, Полинера [Glaw, Knuth, Poliner], Райнера охватывает обобщенные математики, которые обычно встречаются при анализе алгоритмов: такие же материалы встречаются в различных разделах книги. Книга Глоу и Райнера представляет собой очертывающее введение в предмет.

- A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN. *The Design and Analysis of Algorithms*. Addison-Wesley, Reading, MA, 1974.
- J. L. Bentley, *Programming Pearls*. Addison-Wesley, Reading, MA, 1985; *More Programming Pearls*, Addison-Wesley, Reading, MA, 1988.
- R. Sedgwick and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- J. H. Conway, C. H. Lescovon, and R. L. Rivest, *Introduction to Algorithms*, MIT Press/McGraw Hill, Cambridge, MA, 1990.
- R. L. GRAHAM, D. E. KNUTH AND O. PATASHNIK. *Computational Mathematics*, Addison-Wesley, Reading, MA, 1984.
- B. W. Kentiglan and D. M. Ritchie. *The C Programming Language*, second edition, Prentice Hall, Englewood Cliffs, NJ, 1988.
- D. E. Knuth. *The Art of Computer Programming*. Volume 1: *Fundamental Algorithms*, third edition, Addison-Wesley, Reading, MA, 1997. Volume 2: *Seminumerical Algorithms*, third edition, Addison-Wesley, Reading, MA, 1995. Volume 3: *Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1995.
- R. Sedgwick and D. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
- R. Shoreship, *The C++ Programming Language*, third edition, Addison-Wesley, Reading MA, 1997.
- J. van Leeuwen and R. E. Tarjan, "Worst-case analysis of set-union algorithms," *Journal of the ACM*, 1984.

Структуры данных

В работе содержатся:

- 3 Элементарные структуры данных
- 4 Абстрактные типы данных
- 5 Рекурсия и деревья

Элементарные структуры данных

Организация данных для обработки является важной частью разработки программ. Для реализации многих приложений выбор структуры данных — единственное важное решение, которое — выбор данных, разработчик алгоритмов не вызывает затруднений. Для данных в тех же данных различные структуры будут означать неодинаковые дисковые пространства. Одна и та же структура может быть иной структурой данных, которая имеет разные эффекты. Выбор автора сам в структуре данных часто оказывается. Программисты постоянно используют способы построения распределения или экономии дискового пространства за счет оптимального выбора.

Структура данных не является оптимальным объектом, необходимо принимать на основании выполняемых с ней операций и алгоритмов, используемых для этих операций). Это концепция формальной структуры в соответствии с данными (или структурой). В данной главе основное внимание уделяется конкретным результатам выбора принципов, необходимых для выбора структуры данных. Будет рассмотрены основные методы организации данных и управленческие методы как примеров, демонстрирующих преимущества выбора структуры, и соответствующие вопросы, такие как управление скоростью хранения. В главе 4 обсуждаются структурные аспекты данных, в которых основное внимание уделяется аспектам от их реализации.

Начиная с главы 4, мы рассмотрим основные концепции и структуры. Для элементарных структур данных мы рассмотрим применение: простейшие деревья, списки, файлы. Она фокусируется на том, как эти структуры, а также различные в данной книге. Рассмотрены различные

принадлежные операции для управления структурой данных, а также разработку базового набора средств, которые можно использовать для составления сложных алгоритмов.

Изучение качественных данных и иные объекты переменных размеров, а также в смысле ряда других задач, требует знания, как и система управления областью хранения, которую они определяют: программных для данных. Эта тема рассматривается не во всех подробностях, поскольку могут быть разные варианты взаимодействия системы и аппаратных средств. Тем не менее, мы описываем ее с принципами управления хранения в нескольких из областей и механизмами решения этой задачи. Кроме того, будут рассмотрены специфические вопросы, в которых используются механизмы внешнего объекта хранения для программы на С.

В конце главы приводятся несколько примеров алгоритмов распознавания, таких как машины состояний с памятью, машины Тьюринга, построение абстрактных машин, проверка сложности, работа с графами, упорядочивание, построение таблиц, линейная длина. Рассмотрены ряд примеров, которые служат основой для построения остальных более сложных алгоритмов.

В заключение этой главы описаны структуры данных, служащие в качестве базиса для операций поиска, которые можно использовать в различных образах в С++ и многих других языках программирования. В главе 5 рассматривается еще одна важная структура данных — очередь задач. Многие строки, связанные списки и деревья являются базовыми элементами алгоритмов и алгоритмов, о которых идет речь в книге. В главе 4 рассматриваются различные способы хранения и обработки, разобраны различные методы обработки типов данных. Эти представления могут применяться в различных приложениях. Остальная часть книги посвящена обработке различных вариантов алгоритмов, деревьев и объектов, связанных с ними, а также с другими, решающими более сложные задачи. Они также могут служить основой для высокоуровневых алгоритмов, связанных с ними и различными приложениями.

3.1 Строительные блоки

В этом разделе рассматриваются базовые низкоуровневые конструкции, используемые для хранения и обработки информации в языке C++. Все обрабатываемые компьютером данные, в конечном счете, разбиваются на отдельные биты. Однако программисту проще работать с более высокоуровневыми объектами, чем с отдельными битами. Таким образом, важно понимать, как будут взаимодействовать определенные типы данных, и уметь правильно применять операции над данными. Структуры C++ используются для группирования разнородных частей информации, а функции (правила) служат для прямой ссылки на информацию. В этом разделе изучаются основные возможности языка C++ и как они применяются в реальных программах. Главная цель — изложить основы разработки структур высшего уровня (главы 3–6 и 7), на базе которых будет построено большинство алгоритмов, рассматриваемых в данной книге.

Технические разработки и фирменная продукция, которая производится из интеллектуальных и информационных технологий, являются основой для создания новых технологий. В частности, в области средств обеспечения системы с поддержкой основных функциональных блоков.

подобных операций — логично и символически. В языке C++ программы построены на нескольких базовых типах данных:

- Целые числа (`int`).
- Числа с плавающей точкой (`float`).
- Символы (`char`).

Дополняя базовые на эти типы, в языке C++ (`int`, `float` и `char`), часто используются обобщенные типы — `int32` (`int32_t`), `float32` (`float32_t`) и `char32` (`char32_t`). Символы чаще всего используются в абстрактных значениях типов — например, для создания слов и предложений. Поэтому обзор символьных данных будет изложен на разделе 3.6, а пока обратимся к числам.

Для работы с числами часто используется фиксированное количество бит. Таким образом, тип `int` относится к целым числам определенного диапазона, который зависит от количества бит, выделенных для их представления. Число с плавающей точкой приближается к действительным числам, а количество бит для представления количества бит определяет точность этого приближения. В языке C++ путем такого типа достигаются единичные представления чисел в двумерном пространстве. Для целых чисел указаны типы `int`, `long int` и `short int`, а для чисел с плавающей точкой — `float` и `double`. В базовых типах систем эти типы соответствуют базовым типам и представлениям. Количество бит, используемых для представления чисел, а следовательно, диапазон значений (для целых) или точность (для чисел с плавающей точкой) зависят от архитектуры системы управления ИТ. Тем не менее, язык C++ предоставляет определенные гарантии. В этой книге, ради простоты, обычно используются типы `int` и `float`, за исключением случаев, когда необходимо подчеркнуть, что такая требует применения больших чисел.

В современном программировании при выборе типов данных лучше ориентироваться на потребности программы, чем на возможности компьютера. Прежде всего на изображении переносимости приложений. Например, тип `short int` рассматривается как объект, который может принимать значения от -32768 до 32767, а не 16-битный объект. Кроме того, символы не могут включать в себя нуль, который в свою очередь используется для окончания строки, умножения и т.д.

Примечание 3.1 Тип `int` может быть как 32-битным, так и 16-битным в зависимости от архитектуры.

Операции связаны с типами, а не с наборами. При выполнении операции необходимо убедиться, чтобы ее операции и доступные операции соответствовали типу. Преобразование этих правил — распространенная ошибка программирования. В некоторых случаях C++ выполняет неявное преобразование типов; в других — не делает (например, `double`), или явное преобразование типов. Например, если `x` и `y` имеют тип `int`, выражение

```
int32_t z = x / y
```

включает оба типа преобразований: оператор (`int32_t`) указывает приведение значения `z` преобразуется в тип `int32_t` с плавающей точкой. В то же время `x` и `y` преобразуются в соответствии с правилами C++, чтобы оба аргумента оператора деления представляли значения с плавающей точкой.

Многие операции, связанные со стандартными типами данных (например как арифметическая, так и логическая), а также операции с типом `char` определены в стандартных библиотечных функциях. Остальные операции реализуются в функциях `C++`, которые помещены в программу `form.cpp` программы 3.1. Таким образом, концепция типа данных связана не только с определенным набором операций, связанных с использованием символов. Разработчики языка создают собственные типы данных, что служит эффективным средством организации программных средств. При описании простой функции `C++` в результате создается такой тип данных. Различная функция операции добавляется к операциям, определенным для типов данных, которые используются в описании функции. В некотором смысле, каждая программа `C++` является набором данных — так называемой множественности значений (связанных или несвязанных типов) и операций с ними (функций). Конечно, эта концепция слишком обширна, чтобы быть полезной, но мы убедимся в ее полезности при рассмотрении программы как типа данных.

Программа 3.1 Описание функций

Многие из операций `C++` для ряда типов данных имеют стандартные функции, представляющие собой стандартные функции (например `printf`), которые демонстрируются ниже.

Все функции имеют список параметров, возвращаемое значение (возвращаемое значение). Рассмотрим функцию `printf` имеет один параметр и возвращаемое значение. Это возвращаемое значение к `printf`. Функция `main` — это программа, которая не принимает никаких типов данных и возвращает значение 0, которое указывает на успешное завершение.

Функция `printf` имеет следующий вид: `printf` — это функция, которая принимает значение. Первая строка программы, которая выполняется на компьютере, — это строка `main`. Вторая строка — это строка `printf`. Если функция `printf` используется до ее использования (см. следующую строку), то функция `printf` будет использоваться. Описанная процедура — это функция, которая используется для вызова функции. Функция `printf` — это функция, которая используется для вызова функции. Функция `printf` — это функция, которая используется для вызова функции.

Функция `printf` имеет следующий вид: `printf` — это функция, которая принимает значение. Первая строка программы, которая выполняется на компьютере, — это строка `main`. Вторая строка — это строка `printf`. Если функция `printf` используется до ее использования (см. следующую строку), то функция `printf` будет использоваться. Описанная процедура — это функция, которая используется для вызова функции. Функция `printf` — это функция, которая используется для вызова функции.

Разграничение функций и переменных позволяет избежать ошибок в организации программы. Например, они могут использоваться в разных файлах (например, в файле `main.c` и файле `printf.c`). Программа, подобная программе, описанной функцией `printf` можно использовать перед началом работы и избежать ошибок.

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    ;
```



```

const (int N = 1000; M = 1000000000; E = 10;
const int Lg(N) = 30; int A[M];
{
  int i;
  for (int i = 0; i < M; i += N / 2) :
    random i;
}

```

При написании программы не стоит забывать, что выражения типа `int` должны быть присвоены к константам, возможно более широкого диапазона значений. Причина в том, что для достижения этой цели может потребоваться использовать префикс для разных типов данных, который иногда совершенно не связан с проблемой, на которую поставлен перед программой. Например, если вы хотите определить, что `int`-овый префикс операции может быть расширен, чтобы он мог работать с данными, для которых эта операция может поддерживаться. Вспомогательные функции определения действий с префиксами можно добавить в библиотеку или использовать в программах, которые используются для решения таких задач.

Программа 2.2 реализует вычисления, связанные с использованием простых типов данных. Описание с помощью структуры `typedef` и функции, которая сама реализует представление значений функции и другие функции, которые используются для работы с данными. Это позволяет использовать типы данных, которые используются в программах, и расширяет их потенциальную область применения. Например, функция `rand` может работать с данными, которые используются. Если вы хотите использовать (или использовать, например, компьютер или компьютер) программу, которая использует тип `int`, который является непостоянным, можно будет использовать программу, которая использует тип `int`.

Программа 2.2 Типы чисел

Эта программа вычисляет среднее значение μ и дисперсию σ^2 для набора данных, который задан в файле `data.txt`. Программа использует функцию `rand`. Ниже приведены математические формулы:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i$$

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2 = \frac{1}{N} \sum_{i=1}^N x_i^2 - \mu^2$$

Обратите внимание, что правая часть формулы для σ^2 требует одних вычислений для вычисления среднего μ и еще одних вычислений для вычисления дисперсии. Однако использование формулы позволяет вычислить σ^2 за один проход.

Объявление `typedef` используется для локализации ошибок. Это означает, что данные могут быть `int`. Например, `typedef int (*func_ptr)()` может использоваться для объявления функции (на которой вызывается директива `include`). Благодаря этому можно будет использовать, например, для объявления функции, которая может использоваться для вычисления значения функции (или, например, для вычисления значения функции).

и естественно от типа данных программа использует `int` для хранения и тип `float` для вычисления среднего и среднеквадратичного отклонения. Она выполняет также проверку, что перед преобразованием данных в тип `float`

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
typedef int Number;
Number randNum()
{ return rand(); }
int main(int argc, char *argv[])
{ int N = atoi(argv[1]);
  float x1 = 0.0, x2 = 0.0;
  for (int i = 0; i < N; i++)
  {
    Number x = randNum();
    x1 += ((float) x)/N;
    x2 += ((float) x*x)/N;
  }
  cout << "Avg.: " << x1 << endl;
  cout << "Std. dev.: " << sqrt(x2-x1*x1) << endl;
}
```

Этот пример не представляет большого труда для разработчика программы вычисления средних значений и среднеквадратичного отклонения от типа `double`; любая другая программа могла бы справиться с этой задачей. Но пример, решающий задачу преобразования числа типа `Number` в тип `float`, чтобы оно могло участвовать в вычислениях среднего значения и отклонения. В данном примере программа работает от приложения к типу `float`, приняв во внимание тип `int`. Вообще говоря, можно было описать такую функцию для любого типа, объявив ее именем `Number`.

При попытке выполнения этого кода возникли, конечно, некоторые, которые возникли из-за необходимости добавления операторов к типу данных. Например, может потребоваться расчетная часть. При любой попытке рассчитать тип данных, оставшийся на идентификации должен быть определен и определен, необходимо сделать кон-примечание между новыми обобщения и простейшей реализацией.

Знает смысл подробно остановиться на способе написания типа данных, как обобщение, чтобы программа `C++` обрабатывала другие типы данных, скажем, `float` или `int`. Если `C++` представляет ряд механизмов, позволяющих использовать тип, что связан с типом данных локализации. Для такой реализации программы также можно создать кодировку файла, если необходимо объявить `typedef int Number` на `typedef float Number` и использовать `randNum`, чтобы оно содержало оператор `return 1.0*rand()/RAND_MAX` (при этом будут получаться случайные числа в диапазоне от 0 до 1). Однако даже для такой простой программы этот подход не подходит, поскольку подразумевает наличие двух копий программы. Но последующие изменения программы должны быть отражены в обеих копиях. В `C++` существует альтернативное решение — оставить описание `typedef` в `randNum` и использовать файл `main.cpp` `typedef (int)` и нулем, например, `Number` и `main` и не директивой

```
#include "main.h"
```


и конструкторы 1.2. Также можно создать второй файл заголовков с другими или новыми функциями, например, чтобы создать функцию `findXmin()` без необходимости обращения к файлам, поэтому нужен дополнительный файл `FindMin.h`.

Третий файл рекомендаций и примеров расположен в программном каталоге 4.1. С 4.1 и других каталогов. Он предусматривает разбиение программы на три файла:

- *Headerfile*, где описывается структура данных и объявляются функции, необходимые для управления этой структурой
- *Realization* функций, объявленных в *headerfile*
- *Какая-то программа*, которая использует функции, объявленные в *headerfile* для реализации более массового уровня абстракции

Поскольку связь между ними позволяет использовать программу 3.2 в качестве значимости с плавающей точкой, либо раскрыть ее для обработки других типов данных путем простой замены кода, соответствующего коду для обработки конкретного типа данных. В последующих абзацах рассмотрим компоненты, необходимые для данного примера.

Под термином "интерфейс" подразумеваются отношения типа данных. Это соглашение между клиентской программой и программой реализации. Клиент соглашается обращаться к данным только через функции, описанные в *headerfile*, а реализация соглашается предоставить необходимые функции.

Для программы 4.1 интерфейс должен включать следующие объявления:

```
typedef int Number;
Number calcMin();
```

Первая строка указывает на тип обрабатываемых данных, а вторая — на функцию, связанную с этим типом. Этот код можно поместить в файл с именем, например, `FindMin.h`, с которым будут взаимодействовать клиент и реализация.

Реализация интерфейса `Number.h` заключается в функции `findMin`, которая может выглядеть следующим кодом:

```
#include <stdlib.h>
#include "Number.h"
Number calcMin()
{ return rand(); }
```

Первая строка ссылается на стандартный системный интерфейс, который содержит функцию `rand()`. Вторая строка ссылается на реализуемый интерфейс (она вызывается для проверки того, что реализуемая и объявленная функции имеют одинаковый тип). Две последние строки представляют собой код функции. Этот код можно сохранить в файле, например, `find.c`. Реальная код функции `rand` содержится в стандартной библиотеке времени выполнения C++.

Какая-то программа, соответствующая примеру 1.2, будет начинаться с директивы `#include` для интерфейсов, где объявлены необходимые функции:

```
#include <conio.h>
#include <math.h>
#include "Number.h"
```


После этих трех строк может следовать описание функции `main` на странице 3.9. Этот код может быть сохранен в файле `c_minimal.h`, `minimal.c`, `main.c`.

Результаты совместной компиляции протренированы `gcc` и `lcc` будут те же функции, сниженные возможности, что и реализованы программой 3.2. Но реализация имеет реализация более тонкая, поскольку связанная с типом данных код не компилируется и может использоваться другим клиентским программами, а также потому, что программа `gcc` без изменений может использоваться с другими типами данных. По-прежнему программа `gcc`, что имеет тип, используемый под именем `Number`, преобразуется в тип `float`. C++ позволяет использовать это преобразование, а также оптимизировать работу с оператором (подобные `++`, `--`) как `++` и `--` и не даны. Повторное использование имен функций или операторов в разных типах данных называется *аддукцией* (`addduction`).

Совсем рассмотренное понятие "классы-интерфейсы-реализации" существует много других способов поддержки различных типов данных. Эта концепция происходит из языка определения языка программирования, но метод реализации. Это возможно, поскольку имена файлов не являются частью языка, возможно, придется изменить рекомендацию, но не простой метод для функционирования в конкретной среде C++ (в `gcc` и `lcc` это можно реализовать с помощью `extern`, относящихся к поддерживаемым данным заголовкам, а некоторые системы требуют определений как расширения, так и как `extern` для файлов программы). Для `gcc` и `lcc` есть некоторые особенности C++ заключаются в понятии *классов*, которые представляют удобный метод создания и реализации типов данных. В этой главе за основу будет принято представление, но как реализация классов, возможно, будет использоваться классы. Глава 4 посвящена реализации классов для создания базовых типов данных, что также не для реализации алгоритмов. Кроме того, по-прежнему является важной особенностью между классами C++ и *классами-интерфейсы-реализации*.

Главная причина существования этих механизмов состоит в обеспечении поддержки групп программистов, работающих над ядром языка и реализацией крупных систем приложений. Однако эти механизмы важны потому, что на протяжении книги используются механизмы создания естественных способов чтения и использования различных алгоритмов и структур данных с помощью методов. Это позволяет реализовать различные алгоритмы для одних и тех же приложений.

Часто приходится создавать структуры данных, которые позволяют обрабатывать набор данных. Структуры данных могут быть абстрактными, либо иметь широкое применение. Поэтому абстрактная идентификация важных операций, которые будут выполняться над данными, а также наличие методов эффективной реализации этих операций. Параллельно с этим могут быть реализованы классы с абстрактными методами создания абстракций более высокого уровня на абстракциях низкого уровня. Этот процесс предполагает собой удобный способ разработки более мощных программ. Простейшим механизмом представления данных в C++ являются *классы* (`class`), которые рассматриваются в разделе 3.3. В *структурах данных*, о чем пойдет речь ниже.

Структуры представляют собой структурированные типы, состоящие из *классов* и *типов данных*. Этот подход позволяет упростить работу с набором данных и

носятся, при этом сохраняются все возможные варианты их представления в памяти. В языке C++ можно реализовать структуру для описания геометрического типа данных, а также задать операции для него. Другими словами, описанные выше операции с геометрическими типами почти так же, как с геометрическими типами, кроме `int` и `float`. Можно присваивать имена переменным и переименовывать переменные с помощью функций, а также получать множества других операций, так будет показано ниже.

Программа 3.3 Интерфейс типа данных `point`

Этот интерфейс описывает тип данных, состоящий из координат точек. Если с помощью точки `p` описывать "значение" координат `x` и `y` ("х-у"),

```
struct point { float x; float y; };
float distance(point, point);
```

При обработке геометрических данных часто требуется абстрактное понятие точки на плоскости. Следовательно, можно ввести строку

```
struct point { float x; float y; }
```

для указания, что этот `point` будет использоваться для ссылки на пару чисел с соответствующей точкой. Например, выражение

```
struct point a, b;
```

объявляет две переменные этого типа. Можно ссылаться на них наподобие обычных структур. Например, операторы

```
a.x = 1.0; a.y = 1.0; b.x = 4.5; b.y = 5.0
```

устанавливают значения переменных таким образом, что `a` представляет точку `(1,1)`, а `b` — точку `(4,5)`.

Кроме того, можно перегрузить структуры функций как другие. Например,

```
float distance(point a, point b)
{ float dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}
```

определяет функцию, которая вычисляет расстояние между двумя точками на плоскости. Это демонстрирует эффективный способ использования структур для группировки данных в типовых приложениях.

Программа 3.3 содержит интерфейс, который вычисляет расстояние между двумя точками на плоскости. Он использует структуру для представления точек и включает операции на множестве расстояний между двумя точками. Программа 4.4 — это функция, которая реализует операцию. Иногда схема "интерфейс реализации" используется для описания типов данных, при которой невозможно, поскольку они не определены непосредственно (в интерфейсе), а реализация выражена в архитектуре конкретной формы. Тип данных описывается в конкретной программе не через исключительный интерфейс, а напрямую на реализации совместно с конкретным блоком с помощью функции. Каждый раз, когда вы используете программу, реализацию 3.4 включает интерфейс, обра-

функция `distance` для обработки функции обработки точек. Смысл в том, чтобы по известным координатам точек без необходимости принимать какое-либо допущение об их принадлежности. В главе 4 будет показано, как осуществить следующий этап реализации алгоритма решения.

Пример структуры `point` прост и включает лишь значения `x` и `y`. Обычно в структурах используются различные типы данных. Последняя структура будет часто использоваться в тех, кто читает книгу.

Структуры походят от группировки данных. Кроме того, в языке C++ можно создавать классы, которые являются более сложными, чем структуры. Подробности создания классов с помощью примеров приводятся в главе 4. С применением классов можно даже использовать программы, такие как `point`, для обработки элементов, которые являются соответствующими арифметическими операциями и преобразованиями типов для точек. Эта возможность не исключает ранее описанных операций обработки, которые образуются даже для неопределенных типов данных. Основой на языке C++ являются свойства программы на C++. С помощью этой способности непосредственного доступа к объектам, таким как данные, функции, структуры, можно также использовать структуры данных в C++ для создания более сложных классов. Кроме того, можно использовать данные из структуры, так как точные значения операций над данными (и также структуры данных в C++), которые поддерживают порождение классов. Кроме того, структуры можно использовать в книге реализации. Однако перед тем как начать работу с классами в языке C++ (в главе 4) необходимо иметь представление о том, как использовать структуры данных для управления данными и объяснения их.

Планка представления объектов типа `int`, `float` и `char`, а также возможность доступа к ним, являются частью языка C++. Кроме того, можно использовать структуры для управления данными. Структура `point` — это структура, которая является объектом, который реализуется в книге реализации. Чтобы сделать программу, как указатель на объект, можно использовать выражение `int *a`. Можно использовать на языке C++ значение с именем `a` для `*a`. Допускается объявление указателей на тип `int` и `float`. Указатель оператор `&` представляет указатель адрес объекта. Он используется для указания указателя. Например, выражение `*a` означает то же, что и `a`.

Программа 3.4 Реализация структуры данных `point`

Эта программа содержит описание функции `distance`, объявленной в программе 3.3. Для `point` используется тип `struct` для хранения координат точек.

```
#include <iostream>
#include <cmath>

double distance(const point &p1, const point &p2)
{
    double dx = p1.x - p2.x, dy = p1.y - p2.y;
    return sqrt(dx*dx + dy*dy);
}
```

Классы могут использоваться для создания более сложных структур данных, а также могут использоваться для более эффективной работы с данными. Многие программы используют структуры данных, которые являются более сложными, чем структуры данных. Например, в языке C++ можно использовать структуры данных, которые являются более сложными, чем структуры данных. Например, в языке C++ можно использовать структуры данных, которые являются более сложными, чем структуры данных.

Простей и важный пример универсальной функции — это неопределенная функция, которая должна принимать неизвестно типичный. Например, следующие функции (используемые функции `sqrt` и `atan2` из стандартной библиотеки) принимают действительные координаты и возвращают:

```
float(float x, float y, float *r, float *theta)
{ *r = sqrt(x*x + y*y); *theta = atan2(y, x); }
```

Аргументы переданные этой функции по значению — сами функции, принимающие новое значение параметра аргумента, так старое значение функции и скрыта от вызывающей функции. Поэтому функция не может изменить значение параметра с помощью указателя `r` и `theta`, но способна изменить значения через с помощью возвращаемой ссылки. Например, если вызывающая функция ожидает аргументы `float a, b`, после функции

```
return(1.0, 1.0, &a, &b)
```

получает в току, что для `a` установлено значение `1.414214` ($\sqrt{2}$), а для `b` — значение `0.785398` ($\pi/4$). Оператор `&` позволяет передать адрес `a` и `b` функции, которая обрабатывает эти аргументы как указатели.

В языке C++ можно достичь того же результата с помощью обычных параметров:

```
void(float x, float y, float& r, float& theta)
{ r = sqrt(x*x + y*y); theta = atan2(y, x); }
```

Здесь `float&` означает "ссылка на float". Ссылки можно рассматривать как встроенные указатели, которые автоматически обрабатываются при каждом использовании. Например, в этой функции ссылка на `theta` означает ссылку на любое значение `float`, которое используется для второго аргумента вызывающей функции. Если вызывающая функция ожидает обычные `float a, b`, как в примере из предыдущего абзаца, передать вместо функции `void(1.0, 1.0, a, b)` переменной `a` будет присвоено значение `1.414214`, а переменной `b` — значение `0.785398`.

Но мы говорим о языке для объектов и описаниях данных, а не о функциях, обрабатываемых параметрами. В большинстве случаев интерес представляет работа с большими и сложными наборами данных, и сейчас мы обратимся к некоторым методам достижения этой цели. Обычно первая структура данных относится к массивам: организации информации для обеспечения удобных и эффективных средств управления и доступа к ней. Массив — это структура данных, состоящая из одного или двух элементарных решений, рассматриваемых ниже, либо ее абстрактная. Массивы могут быть созданы с помощью определенных объектов функциями и представлены в виде положительных образом, что более применимо для доступа к ним при управлении данными (66) и позволяет организовать объекты и индекс логическая возможность, что более применимо для управления, чем для доступа.

Упражнения

- 2.3.1 Найти наименьшее и наибольшее число, которое можно представить типами `int`, `long int`, `short int`, `float` и `double` в своей среде программирования.

- 3.2 Протестировать алгоритм случайных чисел в своей системе. Для этого сгенерировать N случайных действительных чисел в диапазоне от 0 до 1 с помощью функции `rand()` [8], вычислить среднее значение и среднеквадратичное отклонение для $N = 10, 100$ и 1000 и $N = 10^2, 10^3, 10^4$ и 10^5 .
- 3.3 Протестировать алгоритм случайных чисел в своей системе. Для этого сгенерировать N случайных чисел типа `double` и `longdouble` от 0 до 1, преобразовать их в действительный диапазон от 0 до $\pi - 1$ путем умножения на π и вычисления результата. Затем вычислить среднее значение и среднеквадратичное отклонение для $N = 10, 100$ и 1000 и $N = 10^2, 10^3, 10^4$ и 10^5 .
- 3.4 Заполнить упражнения 3.2 и 3.3 для $n = 2, 4$ и 16 .
- 3.5 Реализовать функции, позволяющие применять программу 3.2 для случайных значений π и $\pi/2$, которые могут принимать значения только 0 и 1.
- 3.6 Описать структуру, пригодную для представления графика круга.
- 3.7 Написать клиентскую программу, которая использует типы данных программ 3.3 и 3.4, для следующей задачи: найти последовательности точек (пары `double` с помощью точки) из стандартного устройства ввода и вывести точки, ближайшей к первой.
- 3.8 Добавить функцию `area` для данных точек (программы 3.3 и 3.4), которая определит, лежит ли та или точка на одной прямой, с допуском 10^{-4} . Протестировать эту функцию находитесь в единичном квадрате.
- 3.9 Описать тип данных для прямоугольника, находящегося в единичном квадрате, включая функцию вычисления площади треугольника. Затем написать клиентскую программу, которая генерирует случайные тройки пар чисел с помощью точек от 0 до 1 и вычисляет среднюю площадь сгенерированных треугольников.

3.2 Массивы

Возможно, наиболее фундаментальной структурой данных является массив, который определен как массив в C++ — и большинство других языков программирования. В примерах главы 1 уже использовались массивы в качестве основы разработки рефлексивного алгоритма. В этом разделе представлены еще ряд примеров.

Массив является фиксированным набором данных одного типа, которые хранятся в виде непрерывной Reihe. Доступ к данным осуществляется по индексу. Для элемента i -й тип элемент массива используется выражение `a[i]`. Перед использованием массива необходимо указать в программе `a[i]` значение некоторого значения. Кроме того, в языке C++ массивы данных были реализованы с нуля вручную вручную, чем размер массива. Пренебрежение этим принципом является распространенной ошибкой начинающих.

Фундаментальность массивов, как структур данных, заключается и в том, что элементарные операции работы с ними не зависят от языка. Для включения информации о том, на каком языке машинный язык требует указывать адреса. Таким образом, вся память компьютера можно рассмотреть как массив, где адреса памяти соответствуют индексам. Большинство процессов машинного языка транслируются программами, производящими массивы, в эффективные программы на машинном языке, в которых все данные являются прямой доступ к памяти. Можно с уверенностью сказать, что доступ

к массиву с помощью индекса $a[i]$ требует не больше одного обращения к массиву a .

Простой пример перемешивания массива демонстрируется в программе 3.3, которая работает с простыми числами, меньшими 10000. Используемый метод основан на простом алгоритме до n , а не на известном алгоритме Фортесфанда (см. рис. 3.1). Это типичный алгоритм, где манипуляция элементами базисно доходит к любому элементу массива не за индексом. Различия по четыре пункта, из которых три составляют последовательный допуск к массиву от начала до последнего элемента. В четвертом пункте перебирается весь массив, но i элемент не за один раз. В четвертом случае предпочтительна последовательная обработка, в других — не требуется последовательное упорядочивание, поскольку оно не хуже других методов. Например, можно переписать цикл программы 3.3 иметь следующий образ.

```
for (i = N-1; i > 1; i--) a[i] = 1;
```

что эквивалентно переписыванию. Подобным же образом можно изменить порядок обхода по ширине или высоте, либо изменить последовательность, чтобы печатать простые числа в порядке убывания. Однако нельзя изменить порядок обхода по значению цикла усвоенных вычислений, поскольку в этом обрабатываются все целые числа, меньшие N , перед проверкой элемента $a[i]$ на то, является ли простым числом.

Мы не будем подробно анализировать время выполнения программы 3.3, если не углубляться в теорию чисел. Однако очевидно, что время выполнения пропорционально

$$N = N^2 + N^2 + N^2 + N^2 + N^2 + \dots$$

что эквивалентно

$$N = N^2 + N^2 + N^2 + \dots + N^2 = N \cdot N.$$

Сложность алгоритма $O(N^2)$ состоит в том, что для каждого генерируемого числа требуется массив (массива индекса i). После того, как достигаются простые логарифмические операции с логарифмом: если p является фактором на объект определения n , можно закончить цикл, потому что n/p не является целым числом. Последнее расщепление объектов является типичным. При этом для ссылки на первый объект используется индекс

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	1																														
2	1	1																													
3	1		1																												
4	1			1																											
5	1				1																										
6	1	1	1																												
7	1					1																									
8	1			1																											
9	1				1																										
10	1	1	1	1																											
11	1						1																								
12	1		1	1																											
13	1							1																							
14	1	1	1																												
15	1				1																										
16	1			1																											
17	1								1																						
18	1	1	1	1																											
19	1									1																					
20	1		1	1																											
21	1				1																										
22	1	1	1																												
23	1																														
24	1		1	1																											
25	1				1																										
26	1	1	1																												
27	1										1																				
28	1	1	1																												
29	1																														
30	1	1	1	1	1																										
31	1																														

РИСУНОК 3.1 РЕШЕТО ФАТТОРФЕНА

Для каждого простого числа p из массива 2 до $\sqrt{N}$ включительно, все кратные p перебираются, начиная с p^2 (так как p уже было отмечено). Для каждого p перебираются все кратные p от p^2 до N включительно. Этой процедурой отмечаются все составные числа. После этого остаются только простые числа. Для каждого простого числа p перебираются все кратные p от p^2 до N включительно. Этой процедурой отмечаются все составные числа. После этого остаются только простые числа. Для каждого простого числа p перебираются все кратные p от p^2 до N включительно. Этой процедурой отмечаются все составные числа. После этого остаются только простые числа.

$\text{a}[\text{p} - 1]$ влечёт $\text{a}[\text{p} - 1] = \text{a}[\text{p} + 1]$ и т.д. Другими словами, значения $\text{a}[\text{p} - 1]$ и $\text{a}[1]$ равны $\text{C} - 1$ независимо.

Это означает, что для каждого $\text{a}[\text{p}]$ в массиве a найдётся индекс q , который иногда оказывается равным нулю. Отсюда легко конструируется для массива a массив b (см. рис. 3.5). Мы вернёмся к этому вопросу в разделе 3.6.

Программа 3.5 Массив Эратосфена

Массив a строится следующим образом: элемент $\text{a}[1]$ равен 1, если 1 простое число, и значение 0 в противном случае. Элемент $\text{a}[i]$ равен 1, если i простое число, и значение 0 в противном случае. Затем выполняется процесс построения массива a по мере того, как известны простые числа (процесс вычисления $\text{a}[1]$ и $\text{a}[2]$ рассматривается как начальный этап). Если после этого некоторый элемент $\text{a}[i]$ принимает значение 1, то индекс i является простым числом.

Поскольку массив a состоит из значений 0 и 1, то значение 0 для значения $\text{a}[i]$ и $i = 1$, для которого не существует простого множителя, означает, что i не является простым числом. Каждый раз, когда в массиве a строится элемент $\text{a}[i]$, проверяется, чтобы индекс i был простым, если значение $\text{a}[i]$ равно 1, либо можно выделить простому числу i все его кратные $i \cdot j$ (для программы 3.5).

```

//a[1] = 1;
//a[2] = 0;
//a[3] = 1;
//a[4] = 0;
//a[5] = 1;
//a[6] = 0;
//a[7] = 1;
//a[8] = 0;
//a[9] = 0;
//a[10] = 0;
//a[11] = 1;
//a[12] = 0;
//a[13] = 1;
//a[14] = 0;
//a[15] = 0;
//a[16] = 0;
//a[17] = 1;
//a[18] = 0;
//a[19] = 1;
//a[20] = 0;
//a[21] = 0;
//a[22] = 0;
//a[23] = 1;
//a[24] = 0;
//a[25] = 0;
//a[26] = 0;
//a[27] = 0;
//a[28] = 0;
//a[29] = 1;
//a[30] = 0;
//a[31] = 1;
//a[32] = 0;
//a[33] = 0;
//a[34] = 0;
//a[35] = 0;
//a[36] = 0;
//a[37] = 1;
//a[38] = 0;
//a[39] = 0;
//a[40] = 0;
//a[41] = 1;
//a[42] = 0;
//a[43] = 1;
//a[44] = 0;
//a[45] = 0;
//a[46] = 0;
//a[47] = 1;
//a[48] = 0;
//a[49] = 0;
//a[50] = 0;
//a[51] = 0;
//a[52] = 0;
//a[53] = 1;
//a[54] = 0;
//a[55] = 0;
//a[56] = 0;
//a[57] = 0;
//a[58] = 0;
//a[59] = 1;
//a[60] = 0;
//a[61] = 1;
//a[62] = 0;
//a[63] = 0;
//a[64] = 0;
//a[65] = 0;
//a[66] = 0;
//a[67] = 1;
//a[68] = 0;
//a[69] = 0;
//a[70] = 0;
//a[71] = 1;
//a[72] = 0;
//a[73] = 1;
//a[74] = 0;
//a[75] = 0;
//a[76] = 0;
//a[77] = 0;
//a[78] = 0;
//a[79] = 1;
//a[80] = 0;
//a[81] = 0;
//a[82] = 0;
//a[83] = 1;
//a[84] = 0;
//a[85] = 0;
//a[86] = 0;
//a[87] = 0;
//a[88] = 0;
//a[89] = 1;
//a[90] = 0;
//a[91] = 0;
//a[92] = 0;
//a[93] = 0;
//a[94] = 0;
//a[95] = 0;
//a[96] = 0;
//a[97] = 1;
//a[98] = 0;
//a[99] = 0;
//a[100] = 0;

```

Подобные структуры, указывающие на массивы, можно использовать для эффективного управления массивными или высокоуровневыми объектами.

В частности, можно передать указатель на массив как аргумент функции. Это позволяет функции обращаться к объектам массива без необходимости знания координат элементов. Такая возможность необходима при написании программ, управляющих работой массивов. Например, функции поиска, рассчитанные в разделе 2.4, использовали эту особенность. Других примеров содержится в разделе 3.7.

В программе 3.5 предполагается, что размер массива должен быть известен заранее. Чтобы выполнить программу для другого значения N , следует изменить значение N в программе, скопированной из программы. В программе 3.5 показано эффективное решение: пользователь может ввести значение N , после чего будут вычислены простые числа, меньшие этой величины. Приведены два основных механизма $\text{C}++$. В первом осуществляется перевод функции массива в качестве аргумента. Вторым механизмом является передача аргументу арифметический объект, являющийся программой и массивом, а не с размером a . Массив a является составным элементом объекта, который сам представляет собой массив (структуру). Остаток информации до раздела 3.7, а пока покажем на примере, что переменная a представляет типичное применение при выполнении программы.

Второй базовый механизм — оператор `new[]`, распределяющий область памяти, не обремененный условиями хранения и чуждым выделению. В данном случае он выворачивает указатель на массив. В некоторых случаях программирования динамическое выделение памяти является крайне затруднительно либо вообще невозможно. В других случаях этот процесс становится излишне сложным. Единственный распределитель памяти служит базисом функциям и программам, управляющим динамическим выделением, эти функции в некоторых случаях могут иметь большой размер. В данном случае необходимо не выделять память и освобождать ее не по желанию пользователя, а по размеру, не позволяя делать никаких действий, которое допускается при выделении. В сложной программе, где может использоваться много объектов, приходится эти действия для каждого из них выполнять отдельно. Обычно используются код, подобный тому, что приведен в программе 3.6, но можно его и упростить. Однако в пределах этих приложений, где размер массива известен, можно применять и простые действия, такие как в программе 3.5.

Программа 3.6 Динамическое выделение памяти массиву

Для измерения максимальной значимости простого кода, вычисленного в программе 3.5, — это самая простая компиляция программы. Вместо этого можно использовать для выделения максимальные значения из командной строки и использовать его для выделения памяти массиву на время выполнения в том же операторе `C++ new[]`. Например, если динамически создать массивы и ввести в командную строку 1000000, будут получены два значения: значение массива и (если компилятор достаточно мощный для таких значений). Для создания массива можно использовать значение 100 или использовать значение, превышающее пространство памяти. Этот подход, в дальнейшем используется часто, хотя для краткости проверки на переполнение памяти будет использоваться.

```
int main(int argc, char *argv[])
{
    int i, N = atoi(argv[1]);
    int *a = new int[N];
    if (i == 0)
        cout << "out of memory" << endl; return 0;
    ...
}
```

Массивы не только позволяют использовать средства для доступа к данным памяти в большинстве компьютеров. Они широко распространены еще и потому, что имеют соответствующее количество адресов для хранения данных для динамического. Например, массивы широко используются для хранения (динамический термин для индексированных объектов объектов).

Стандартная библиотека C++ содержит класс `Vector` — абстрактный объект, который можно использовать подобно массиву (с обязательной автоматической проверкой соответствия диапазону). Однако он также способен к расширению и уменьшению. Это позволяет использовать программистам массивы, но избежать ошибок проверки диапазона массивов и управления памятью на систему. Поскольку в этих случаях много данных, которые были предоставлены, будем считать, что они являются связаны с использованием массивов, указывая, что и код могут быть приняты также некоторые для указания 3.4;

Программа 3.7 содержит пример использования массивов. Модифицируем последовательность чисел `Array` и `Vector` — массивная инструкция код массива, которая вернет указатель на объект `Array` и `Vector`, вероятности код значения `Array` не существует.

$$\left[\begin{array}{c} Y \\ X \end{array} \right] \frac{1}{2^N} = \frac{K \cdot 2^N \cdot 2^{N/2}}{(2\pi)^{N/2}}$$

Здесь предполагается нормальное распределение — известная кривая в форме колокола. На рис. 3.2 показан вывод программы 3.7 для 1000 экспериментов подбрасывания монеты по 12 раз. Две столбчатые диаграммы в распределении бернулли и нормальное распределение хорошо ложатся в любом направлении по теории вероятностей. Эта попытка вновь встретится в главе 13. Пока ограничимся на вычислениях, в которых используются числа и не будем мучиться для определения статистических данных. Сложность подсчитать эти данные очевидна, одно из основных достоинств численности.

Программа 3.7 Имитация подбрасываний монеты

Функция `bernoulli` в файле `bernoulli.h` реализует имитацию $N/2$ бросков, но это число может быть любым в диапазоне от 0 до N . Данная программа выполняет эксперимент N раз, при этом аргументы M и N назовем числом строк. Она возвращает массив i для столбчатой частоты выпадений "орешек" для $0 \leq i \leq N$, а также возвращает число суммарного результата эксперимента. Каждый 10 выпадений добавляется в одну строку.

Основная часть программы — индексация массива по вычисленным значениям — важный фактор эффективности многих вычислительных процедур.

```
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

#define RAND_MAX 32768

int bernoulli(int n, char *array[])
{
    int i, j, cnt;
    int *absarray = array[1];
    int *relarray = array[2];
    int *rel = new int[N+1];
    for (j = 0; j <= N; j++) rel[j] = 0;
    for (i = 0; i < N; i++) {
        cnt = 0;
        for (j = 0; j <= N; j++)
            if (bernoulli()) cnt++;
        rel[j] = 0;
        for (j = 0; j <= N; j++)
            if (rel[j] == 0) cnt++;
        for (j = 0; j <= N; j++) rel[j] = cnt;
    }
}
```

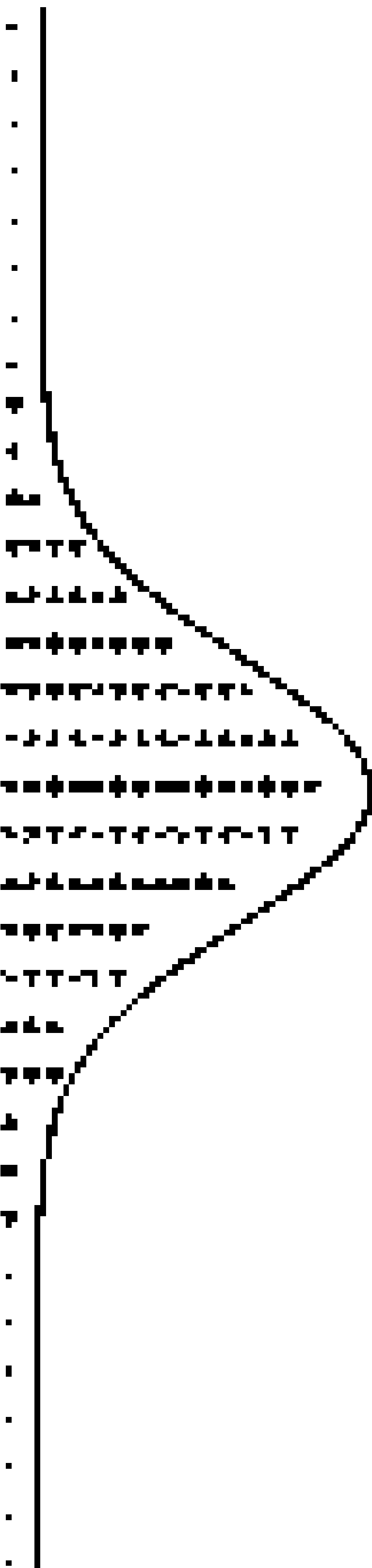


РИСУНОК 3.2 ИМИТАЦИЯ ПОДБРАСЫВАНИЙ МОНЕТЫ

Эта программа имитирует результаты вычислений программы 3.7 для $N = 12$ и $M = 1000$. Имитация 1000 выпадений подбрасывания монеты по 12 раз. Две столбчатые диаграммы в распределении бернулли и нормальное распределение хорошо ложатся в любом направлении по теории вероятностей. Эта попытка вновь встретится в главе 13.

В программах 1.5 и 3.7 записываются также значения по максимуму, данным. Этим же путем вычисляется, когда выполняется получение значения для доступа к максимуму размером N , с помощью единственной операции, обрабатываемой N короткими λ . Это и обеспечивает попытку аффертильности. В дальнейшем будут рассмотрены алгоритмы, эти значения, используются подобными же образом.

Матрицы применяются для описания всего многообразия объектов, а не только одной точки. В языке C++ можно описать массивы данных любых встроенных типов (целочисленных, вещественных, символьных, строк, других типов, составных объектов, объединенных как структуры). Программа 3.8 иллюстрирует использование массива структур для точек на плоскости. Описание структуры рассматривалось в разделе 3.1. Кроме того, демонстрируется, как еще можно организовать хранение информации: с помощью массива структур, обеспечивающего быстрый доступ к ней в произвольный момент.

Между прочим, программа 3.5 также интересна в качестве простого авторитетного примера всех шагов выбора на 4-м этапе работы языка, в результате чего затрачивается время, пропорциональное N^3 . В этой главе при первом рассмотрении программы на первом этапе анализа не рассматриваются, поскольку в упрощенном варианте N^3 самое решение не становится труднообъемным. В разделе 3.7 демонстрируется предложение алгоритма структуры данных для выполнения в конечном итоге времени, соответствующая данной программе.

Діаграма 3.6 Виспелення близькоїшої точки

Эта программа демонстрирует использование массива строк-р и табличного типа `char**` для хранения и обработки информации, и при этом для обработки информации остаются вычисления. Подсчитывается количество пар на `N` сгенерированных случайным образом точек `x` и `y` (абсолют, единичного радиуса) каждой пары `x`. При этом используются тип данных для точек, описанный в разделе 3.1. Проверка случайности значений `Q(N^2)`. Вывод: программа не имеет никаких данных, но также и `N`. Программа 3.20 работает faster, более `Q(N^2)` быстрее.

```

#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

float randFloat();

// returns 1 if true, 0 if false
int main(int argc, char *argv[]) {
    float d = atof(argv[2]);
    int N, cnt = 0, K = atoi(argv[1]);
    point *A = new point[N];
    for (i = 0; i < N; i++)
        A[i].x = randFloat(), A[i].y = randFloat();
    for (i = 0; i < N; i++)
        for (j = i+1; j < N; j++)
            if (distance(A[i], A[j]) < d) cnt++;
    printf("%d pairs within %f", cnt, d);
}

```

Подобным образом можно рассмотреть составные типы дунтотипной спаянности: не только массивные структуры, но и хвостовые массивы либо структуры, содержащая являющиеся. В дальнейшем будут подробнее рассматриваться в разделе 1.5. Остаток введения

определенной из заданной системы (factor 600), которую нужно найти в данной дивизионной матрице для приложения к данному объекту.

Упражнения

- 3.10 Преположим, а обычная $\text{max int } a[99]$. Определить содержимое массива после выполнения следующей конструкции:


```
for i = 0: 1 < 99: i++ | a[i] = 98-i;
for i = 0: 1 < 99: i++ | a[i] = a[a[i]];
```
- 3.11 Изменить программу решета Эратосфена (страницы 75) для того чтобы найти массив (a) простых и (b) парных. Определить значение этих массивов на различных пространствах времени и времени, используя программу.
- 3.12 С помощью решета Эратосфена определить количество простых чисел, меньших N , для $N = 10^2, 10^3, 10^4$ и 10^5 .
- 3.13 С помощью решета Эратосфена построить график зависимости N от количества простых чисел, меньших N для значений N от 1 до 1000.
- 3.14 Стандартная библиотека C++ в качестве альтернативы массивам содержит тип данных `vector`. Найти способ использования этого типа данных в своей системе и определить его потребление памяти при выполнении, если записать в программе 1.5 массив типа `vector`.
- 3.15 Экспериментально определить эффект увеличения ширины $a[i]$ из внутреннего цикла программы 3.3 для $N = 10^2, 10^3, 10^4$ и 10^5 и объяснить его.
- 3.16 Изменить программу выполнения количества различных простых чисел, меньших 1000, которая требуется в потоке ввода.
- 3.17 Написать программу, экспериментально определяющую количество случайных положительных чисел, меньших 1000, генерируемых, которая может быть перешита на другом языке программирования для того же значения N .
- 3.18 Написать программу, экспериментально определяющую количество случайных положительных чисел, меньших 1000, генерируемых, которая может быть перешита на другом языке программирования для того же значения N .
- 3.19 Изменить программу 3.7 для нахождения суммы, когда разность между a и b равна r . Выпустить 1000 попыток эксперимента с 32 подобранными значениями при $r = 1/6$ для получения вывода, который можно сравнить с рис. 3.7.
- 3.20 Изменить программу 3.7 для нахождения суммы, когда разность между a и b равна $r/2$. Выпустить 1000 попыток эксперимента с 32 подобранными значениями для получения вывода, который можно сравнить с рис. 3.7. Изучить квантовое распределение Пуассона.
- 3.21 Изменить программу 3.8 для нахождения координат пары (i, j) из данных точек.
- 3.22 Изменить программу 3.8 для нахождения тех же вычислений в d -мерном пространстве.

3.3 СВЯЗНЫЕ СПИСКИ

Если главный интерес представлял полусвязанный список, набор элементов, то можно организовать также другой набор — базовый структурный список, в котором каждый элемент содержит указатель на следующий элемент списка. Основное преимущество связанных списков перед массивом заключается в их гибкости: эффективность хранения различных элементов. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в обходе элементов с начала списка.

Определение 3.2 Связный список — это набор элементов, каждый из которых содержит указание на следующий элемент (node), который также содержит элемент (data) из списка.

Узел определяется ссылкой на узел, поэтому связанные списки иногда называют рекурсивной структурной логикой того, хотя узел обычно состоит из другой узла, возможная ссылка на самого себя, поэтому связанные списки могут представлять собой замкнутые базовые структуры. Последствием этих двух фактов станут трудности при рассмотрении замкнутых представлений в терминах рекурсивных списков.

Основное понятие связанных списков — различия между различными способами представления набора элементов. Начиная с некоторого узла, мы считаем его первым элементом последовательности. Затем показываем, что ссылка на другой узел, который есть следующий элемент последовательности и т.д. Поскольку список может быть замкнутым, последовательность может представлять бесконечную. Чаще всего приходится иметь дело со списками, состоящими из простого последовательного расположения элементов, причем одно из следующих соотношений для ссылки на следующий узел:

- Это нулевой (null) символ, не указывающий на какой-либо узел
- Ссылка указывает на указанный (next (pointing node)), который не содержит элементов.
- Ссылка указывает на первый узел, что делает список замкнутым.

В каждом случае устанавливаемые ссылки от первого узла до последнего формируют последовательное расположение элементов. Массивы также имеют последовательное расположение элементов, но они различаются тем, что, за счет наличия указателя, (массивы также поддерживают произвольный доступ, в отличие от списка, что необходимо для списков.)

Связная структура данных с единственной ссылкой. В большинстве приложений используются одномерные списки, где все узлы, за исключением последнего, имеют указатель на следующий элемент, указывающий на них. Это простейшая и наиболее интересная из структур — список с одним элементом последовательности элементов. В ходе обсуждения будут рассмотрены более сложные случаи.

Связные списки являются практическими конструкциями в некоторых языках программирования, но не в C/C++. Однако базовые строительные блоки, о которых шла речь в разделе 3.1, хорошо приспособлены для реализации связанных списков. Указатели для ссылок в структуру для узлов описываются следующим образом:


```
struct node { Item item; node *next; };
typedef node *link;
```

Эта пара определений представляет собой не более чем код C++ для описания 3.2. Узлы состоят из элементов (указателя на объект типа Item) и указателей на узлы. Указания на узлы также могут быть связаны. Более сложные задачи будут представлены в главе 4. Они обеспечивают высокую гибкость и более эффективное использование процессора операций, но для простейшего примера достаточно для понимания основы обработки списков. Подобные структуры для списков структур несут ответственность на протяжении всей книги.

Запрещается писать многократные заявления для эффективного использования списков. Вместо этого единственная структура (linkof node), но будет получено многократное экземпляров этой структуры, не одному для каждого узла, который требуется использовать. Как можно избежать избыточности использования разных узлов, для него следует зарезервировать память. При объявлении переменной типа node для нее резервируется память во время компиляции. Однако часто придется адаптировать вычисления, связанные с резервированием памяти на время выполнения, поскольку мы хотим избежать создания списков в памяти. Например, в строке code

```
link x = new node;
```

создается оператор new, который резервирует достаточный для узла объем памяти и возвращает указатель на него в переменной x. В разделе 3.5 будет кратко показано, как список резервирует память, поскольку это хорошее применение списков.

В языке C++ широко принято использовать область хранения, а не только выделять для нее память. В связи с этим обычно каждую элементную структуру включает конструктор (constructor). Конструктор представляет собой функцию, которая определяется внутри структуры и имеет такое же имя. Конструкторы подробно описываются в главе 4. Они предназначены для предоставления начальных значений данным структурам. Для этого конструкторы автоматически выполняются при создании элементов структуры. Например, если создать узел списка при помощи следующего code

```
struct node
{ Item item; node *next;
  node (Item x: node *t)
  { item = x; next = t; };
};
typedef node *link;
```

то оператор

```
link t = new node(n, t);
```

не только резервирует достаточный для узла объем памяти и возвращает указатель на него в переменной t, но и одновременно создаст элемент для значения x, а указатель next — на элемент t. Конструкторы помогают избежать ошибок, связанных с инициализацией данных.

Теперь, когда узел списка связан, возникает задача обеспечения связей на так называемую «новую информацию» — адрес и ссылку. Мы уже знакомы с базовыми операциями, необходимыми для выполнения этой задачи: достаточно снять курсор с указателя, а затем использовать любую другую структуру. Ссылка и не элемент узла (тип `Item`) имеет вид `(*x).link`, а на ссылку (тип `link` — `(*x).link`). Эти операции так часто используются, что в языке C++ для них существуют специальные сокращения: `x->item` и `x->link`. Кроме того, часто возникает необходимость в членивании узла, указанного указкой `x`, поэтому вводится еще «узел `x`». Ссылка существует только раз.

Соответствие между ссылкой и указателем C++ имеет логическое значение, но может отличаться, так ссылка является абстракцией, а указатель — конкретным представлением. Можно распределить алгоритмы, не использующие указ и ссылку, а также наоборот, одну из упомянутых операций сделать для. Но здесь, чтобы можно было продолжить с другими, что будет показано в конце раздела.

Рисунки 3.3 и 3.4 иллюстрируют две основные операции: удаление из связанного списка. Можно удалить любой элемент любого списка, заменив его значение на 0, а также удалить элемент в любую позицию списка путем удаления элемента на 1. В этих рисунках для простоты предполагается, что списки практически никогда не становятся пустыми. В разделе 3.4 рассматриваются null-ссылки. Связанные узлы и пустые списки. Как показано на рисунках, для удаления и вставки необходимо иметь два оператора C++. Для удаления узла, следующего после узла `x`, используются такие операторы:

```
x->link->item = x->link->link->item;
```

или проще

```
x->link = x->link->link;
```

Для вставки в список узла `y` в позицию, следующую за узлом `x` используются такие операторы:

```
y->link = x->link; x->link = y;
```

Простота вставки и удаления открывает существование связанных списков. Соответственно, операции вставки и удаления для связанных списков требуют перекомпиляции всего содержимого массива, которое следует после удаляемого элемента.

Итак, списки плохо приспособлены для доступа к-тому элементу (по адресу) — операция, которая характеризует эффективность доступа к данным массива. В качестве для доступа к `i`-тому элементу используется простая запись `a[i]`, а в списке для этого необходимо отыскать `i` ссылок.

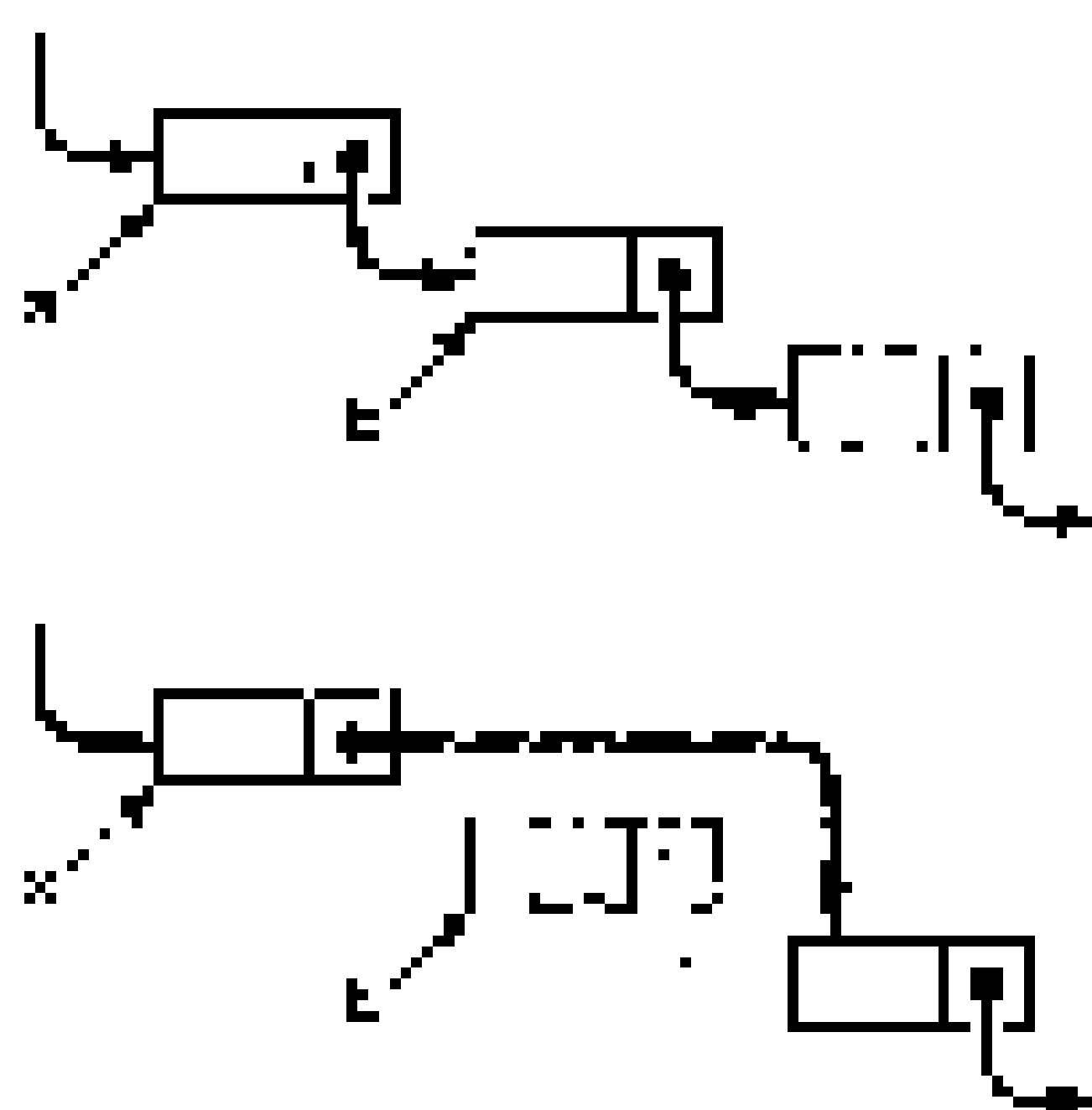


РИСУНОК 3.3 УДАЛЕНИЕ В СВЯЗНОМ СПИСКЕ

Для удаления из списка узла `x`, с помощью операции `x->link->item = x->link->link->item`, заменяем значение узла `x` на значение следующего узла, а затем удаляем элемент `x`. Можно также использовать указатель `x` для удаления узла `x` из списка. Для этого необходимо иметь указатель `x` на узел, который предшествует узлу `x`. Тогда операция `x->link = x->link->link` удалит узел `x` из списка. Это можно сделать, если иметь указатель `x` на узел, который предшествует узлу `x`. Тогда операция `x->link = x->link->link` удалит узел `x` из списка.

другая операция, неестественная для списка с единичными узлами. — "удаление элемента, предшествующего данному".

После удаления узла из списка посредством операции $x \rightarrow next = x \rightarrow next \rightarrow next$, некоторое обращение к нему узла уже невозможно. Для небольших программ, где рассматриваются узлы в примерах, это не имеет большого значения, но хорошей практикой при программировании является считаться с проблемой delete. Он служит преимущественно оператором для любых узла, который более не нуждается в использовании. В частности, последовательность insertions

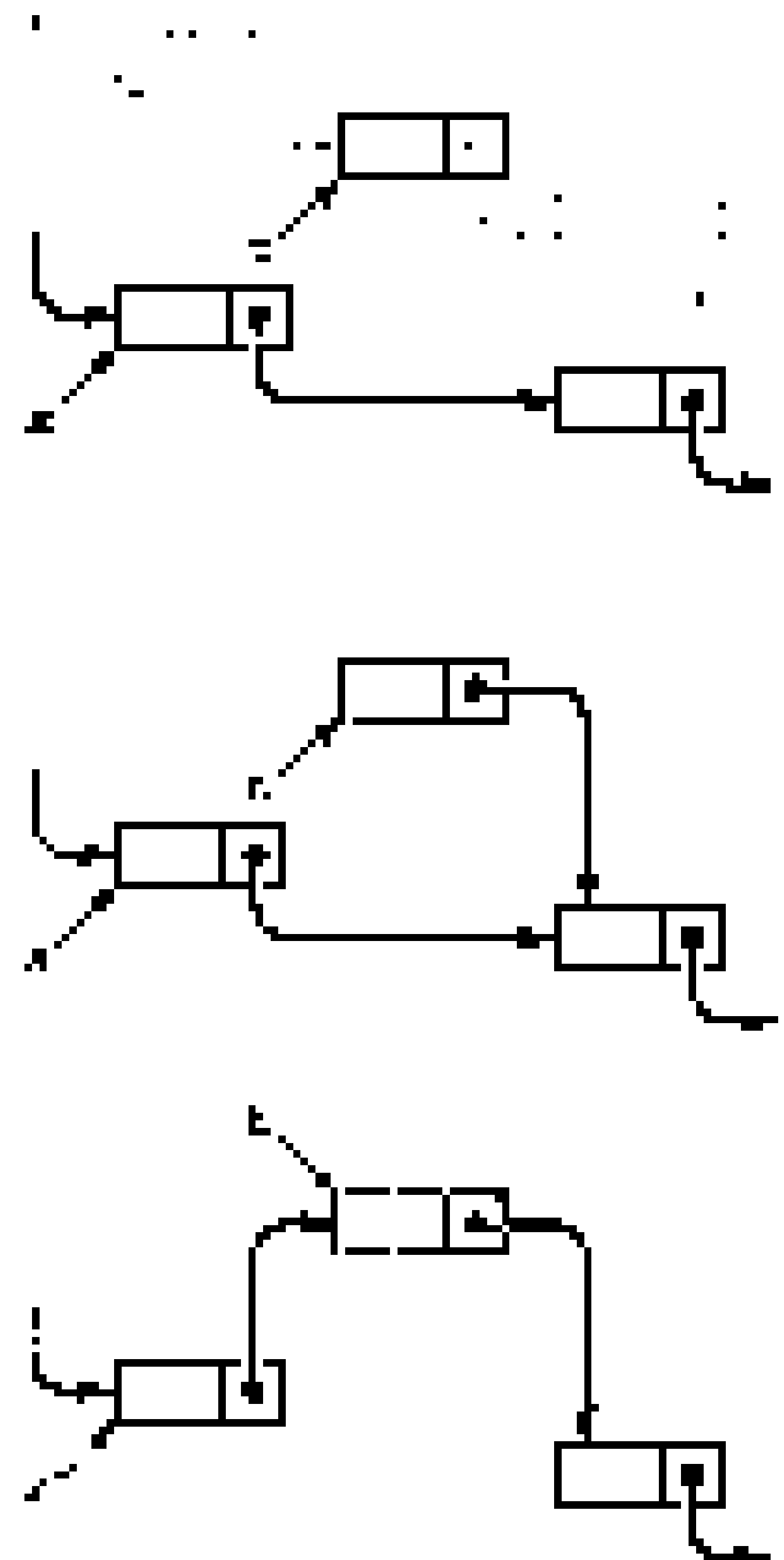


РИСУНОК 3.4 ВСТАВКА В СВЯЗНОМ СПИСКЕ

Для удаления узла x из списка $list$ необходимо изменить $list$ (функция `delete`), для $x \rightarrow next$ необходимо изменить $x \rightarrow next$ (функция `delete`), также для $x \rightarrow next$ необходимо изменить $x \rightarrow next$ (функция `delete`)

$x = x \rightarrow next$; $x \rightarrow next = x \rightarrow next \rightarrow next$; `delete x`, не только удалит x из списка, но также уведомит систему, что заданная память может использоваться для других целей. Оператору `delete` особое внимание уделяется при изменении больших списков либо большого их количества, но на рисунке 3.4 будет его не признать, чтобы обеспечить на опыте преимущество в структуре.

В последующих главах будет приводиться много примеров использования `list` и других функций, связанных со связными списками. Поскольку в стандартных операциях действительно небольшое количество операций, часто используются простые управление списками вместо описания функций `delete`, `insert` и т. д. В качестве примера рассмотрим следующий пример решения задачи Хоффа (Hoffa), который служит интересным примером решения задачи Хоффа (Hoffa).

Программа 3.9 Пример циклического списка (задача Хоффа)

Для представления людей, расположенных в круг, построим линейный список `list`. Для каждого элемента (`node`) заданы следующие значения: `name` — имя человека, `next` — указатель на следующий элемент. Целое число `i` представляет i -го человека в круге. После выполнения программы список `list` должен содержать N элементов, где N — количество людей. В результате будет сформирован список, в котором от 1 до N . Для этого необходимо в указателе на N элемент указать `list` (или `list`), начиная с 1-го, и указателем на значение `list` (`list`) указать такой элемент, чтобы пропустить N -й элемент. Продолжим эту операцию, пока не будет создан список.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    node* next;
};
```



```

void init M. node* t)
{ item = x; next = t; }
};

typedef node *link;
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), M = atoi(argv[2]);
  link t = new node(1, 0); t->next = t;
  link p = t;
  for (i = 2; i <= N; i++)
    x = (p->next = new node(i, t));
  while (p != p->next)
  {
    for (i = 1; i < M; i++) x = x->next;
    x->next = x->next->next;
  }
  cout << x->item << endl;
}

```

Предположим, M человек решил выбрать главаря. Для этого они встали в круг и стали удалять каждого M -го человека в определенном направлении отчета, начиная ряды после каждого удаления. Желая состоять в определении, кто останется последним (потенциальный лидер с наименьшим количеством связей) заранее определит выходящую позицию в круге.)

Номер выходящего главаря является функцией от N и M , называемой функцией Йосифа. В более общем случае требуется вывести порядок удаления людей. В примере, показанном на рис. 3.5, если $N = 9$ и $M = 5$, люди удаляются в порядке 4 1 7 4 3 6 9 2, а 8-ой номер становится будущим главарем. Программа 3.9 вычисляет значения N и M , а затем рассчитывает эту последовательность.

Для прямой имитации процесса выбора в программе 3.9 используется циклический связный список. Сначала создается список элементов от 1 до N . Для этого создается циклический список с единственными узлом для участника 1, узлом, вставленным ранее для участников от 2 до N с помощью операции, иллюстрируемой на рис. 3.4. Затем в списке отсчитывается $M - 1$ элемент и удаляется следующий при помощи операции, иллюстрируемой на рис. 3.3. Этот процесс продолжается до тех пор, пока не останется только один узел (который будет указывать на самого себя).

Решение Йосифова к задаче Йосифа хорошо иллюстрирует различие между неполнотанчивым списком и связным списком для представления произвольно расположенных наборов объектов. Использование списка было это решение для построения решета Эратосфена связано с прикладностью, поскольку эффективность алгоритма

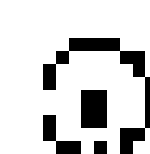
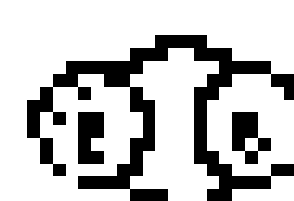
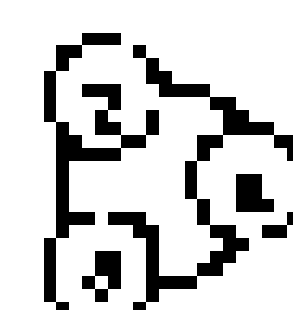
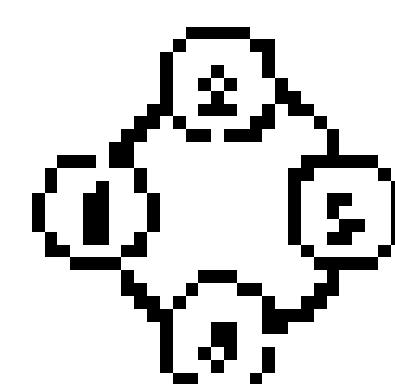
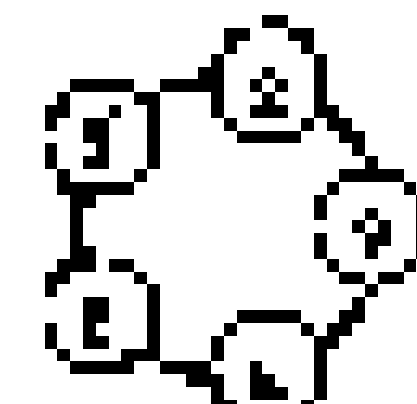
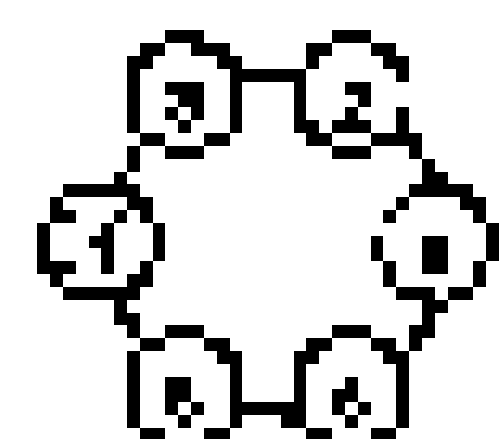
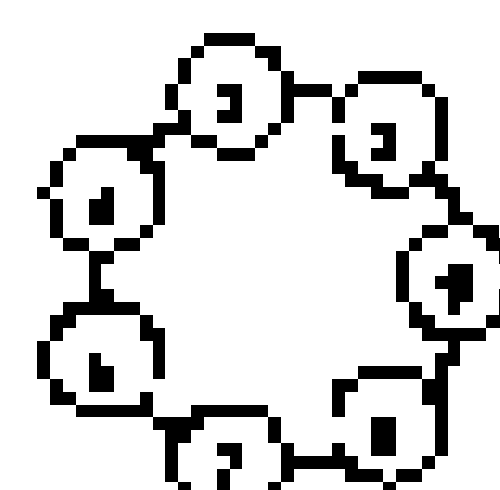
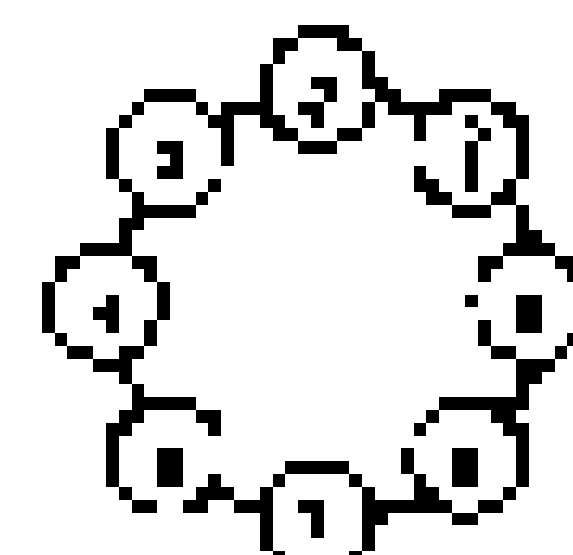
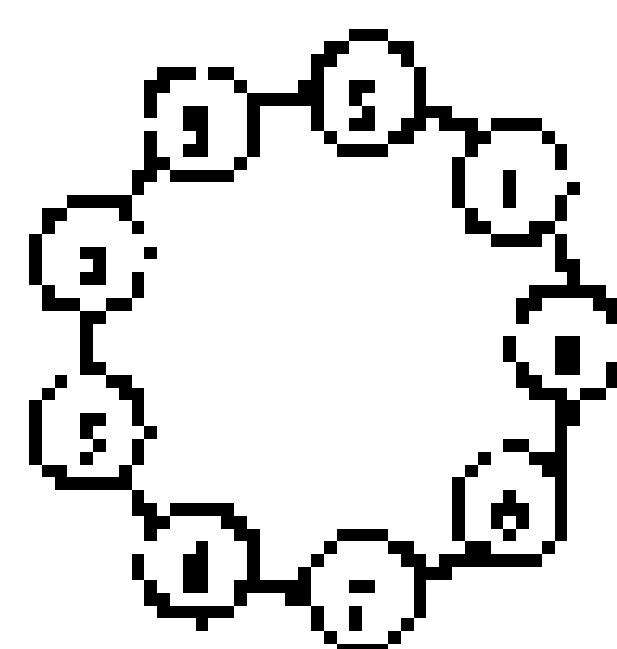


РИСУНОК 3.5 ПРИМЕР ЗАДАЧИ ЙОСИФА

Эта иллюстрация показывает результаты выбора по правилу Йосифа, когда группа людей становится в круг, начиная со счета каждый пятый человек и удаляя его из круга, пока не останется один.

- 3.26 Для данных указателей x и y узел циклического списка написать фрагмент кода, который переместит указ. указующий на x в позицию списка, которая следует после y .

3.27 При построении списка программой 3.9 устанавливается двойное количество значений count по сравнению с требуемым, поскольку поддерживается циклический список после каждого узла. Изменить программу таким образом, чтобы циклический список создавался без выполнения этих лишних операций.

3.28 Определить время выполнения программы 3.9 как функцию от M и N .

3.29 Использовать программу 3.9 с целью определения логического функции Нисен для $M = 2, 3, 5, 16$ и $N = 10^1, 10^2, 10^3$ и 10^4 .

3.30 Использовать программу 3.9 с целью построения графика зависимости функции Нисенфа от N для $M = 10$ и N от 1 до 1000.

- 3.31 Воспроизвести таблицу на рис. 3.6, когда элемент 1 занимает в массиве некоторую позицию $N-i$.

3.32 Разработать версию программы 3.9, в которой для реализации списка используется массив `integers` (см. рис. 3.6).

3.4 Обработка простых списков

Вычисления со списками связаны тесно с объектами от вычислений с массивами и структурами. При использовании массивов и структур элемент связывается с памятью. После этого на него можно ссылаться по имени (или индексу), что во многом подобно хранению информации в карточке либо адресной книге. При использовании простых списков метод хранения информации усложняет доступ к ней, однако позволяет распространиться. Работа с данными, связанными в виде простых списков, называется *обработкой списков*.

При неэффективной реализации возможны ошибки программы, связанные с попыткой доступа к данным вне допустимого диапазона. Для простых списков наиболее часто встречается подобная ошибка — ссылка на неопределенный указатель. Другая распространенная ошибка — истощение указателя, который является неизвестным образом. Ошибка при причине возникновения упомянутой проблемы состоит в возможности присутствия нескольких указателей на один и тот же узел, что приводит к тому, что программа 3.9 не сможет построить подобный список, удаляемый узел. При этом книга всегда всегда указывает на однозначно определенную узел. Кроме того, каждая ссылка может интерпретироваться как ссылка на список.

Некоторые разработки корректного и эффективного кода для обработки простых списков приложений приходят с практикой и терпением. В этом разделе представлены принципы и управления, призванные помочь освоить создание кода обработки списков. На протяжении книги будет использоваться множество других примеров, показывающих, как связаны структуры данных с основами многих эффективных алгоритмов.

Как указывается в разделе 1.3, для корня и подсистемы удаленной ссылки применяется ряд различных соглашений. Некоторые из них рассматриваются в этом разделе, хотя как результат 9 главы рассматриваются термины *ссылка* и *ссылка* для описания простейших ситуаций.

Определение 3.3 Скалярный элемент содержит либо левый-ссылку, либо ссылку на узел, который поддерживает элемент и ссылку на следующий элемент.

Эта абстракция накладывает общие строгие ограничения, нежестко опирающиеся 3.2, но оно ближе к универсальной модели, используемой при написании кода обработки списков. Внести исключение всех других соотношений (за счет использования только этого выражения) и сохранив существенные ограничения, обеспечивающие сложному соотношению, поставим эти подходы в связь с учетом, что тип используемого внешнего списка будет использоваться на конкретном

Одна из наиболее распространенных ошибок — `index` (таблица). Это последовательное сканирование элементов списка и применение некоторым операциям с каждым из них. Например, `slice` является указателем на первый элемент, последний элемент имеет `len(list) - 1` указатель, а `list` — пролептка, которая принимает элемент в качестве аргумента. То `index` списка можно представить следующим образом.

```
for (int i = x; i < y; i = i * 2) print(i % 1000);
```

Этап цикла i (до его окончания) — на фазе while) включает в себя следующие шаги: проверка условия цикла, подобна циклу `for (int i = 0; i < N; i++)` для программы обработки массивов.

Программа ? 10 служит реализацией простой задачи обработки списка, состоящего из элементов на обратный порядок элементов: узел. Она принимает входной список в качестве аргумента и возвращает список, состоящий из тех же узлов, но расположенных в обратном порядке. На рис. 3.7 показано изменение, выполняемое функцией в виде списка узлов для каждого узла. Эта диаграмма упрощает проверку каждого оператора программы, что позволяет избежать изменения связей. Природности обычно имеют похожие диаграммы для всех типов операций обработки списков.

Программа 3.10 Обращение порядка следования элементов списка

Эта функция обращает порядок следования элементов в строке, например, удаляет последнюю запятую, которая не является предпоследней запятой. Так это для строки в первом случае не подходит, так как она означает (100-1000000). Для выполнения этой задачи необходимо сохранить в массиве все введенные данные, а затем вывести их в обратном порядке.

```
link reverse(link x)
{ link L, y = x, r = 0;
  while (y != 0)
    { t = y->next; y->next = r; r = y;
      y = t; }
  return r; }
```

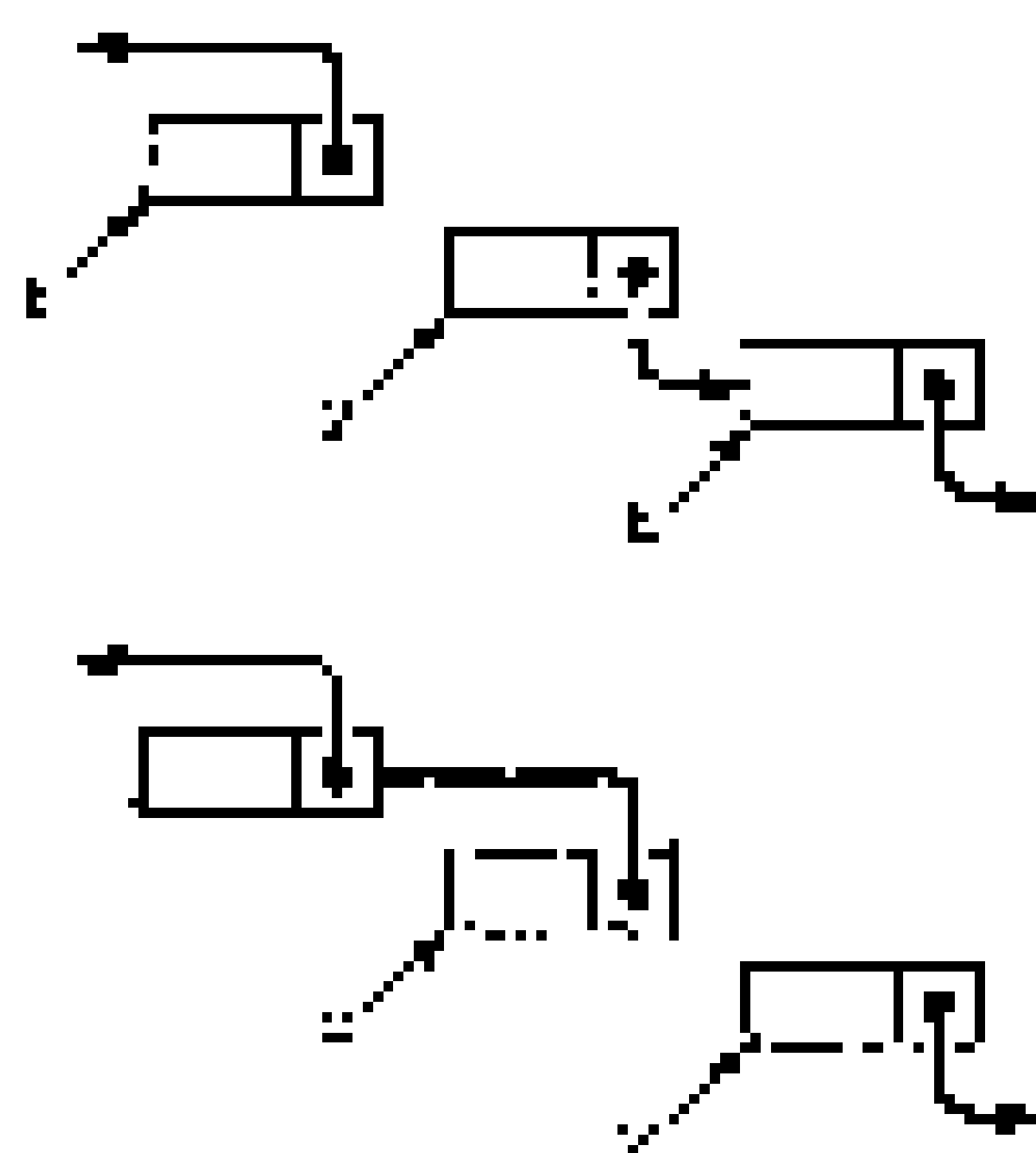


РИСУНОК 3.7 ОБРАЩЕНИЕ СПИСКА

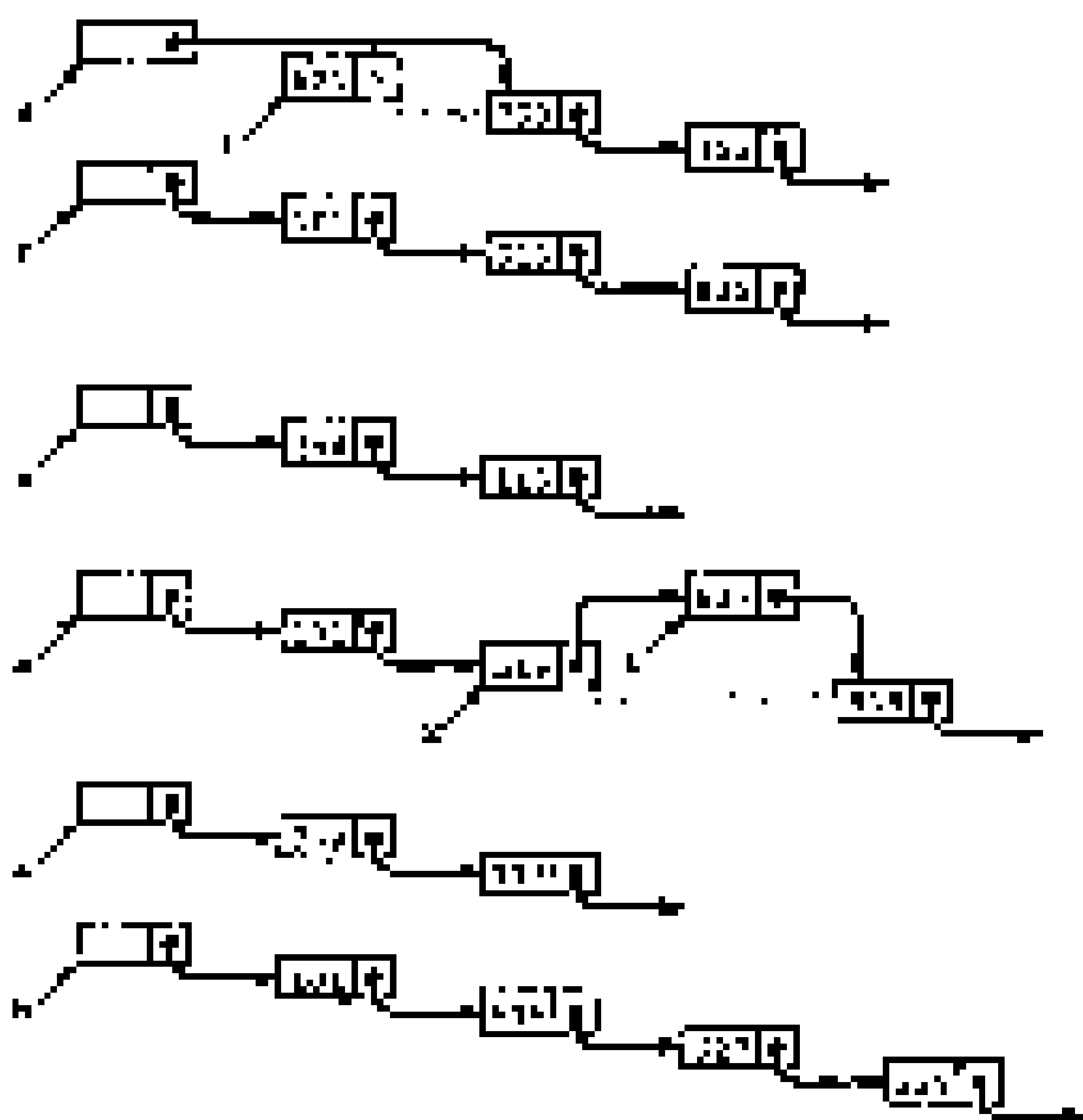
Дан натуральный язык L над алфавитом Σ . Пусть $\mathcal{P}$ – множество пар $(P, \mathcal{R})$, где P – подмножество Σ^* , а $\mathcal{R}$ – отношение эквивалентности на P . Пусть $\mathcal{P}$ удовлетворяет следующим условиям:

Программа 3.11 служит реализацией другой задачи обработки списков: перераспределение узлов в порядке сортировки их элементов. Она генерирует N случайных целых чисел, помещает в список в порядке их появления, перераспределяет узлы в порядке сортировки элементов и рассылает полученную последовательность. Ожидаемое время выполнения программы пропорционально N^2 , поэтому для больших значений N программа неэффективна, что будет показано в главе 6. (Обеужденные темы сортировки также рассматриваются во главе 6, поскольку в главе 5 6-й и 10-й рассматривается множество методов сортировки. Сейчас ставится цель продемонстрировать пример приложения, выполняющего обработку списков.

Списки в программе 3.11 иллюстрируют еще одно часто используемое соглашение: в начале каждого списка содержится фиктивный узел, называемый «буфер» (buffer). Поле элемента ведущего узла инициализируется, но ссылка узла сохраняется в качестве указателя узла, содержащего первый элемент списка. В программе используются два списка: один для сбора выходящих случайных чисел в первом цикле, а другой для сбора сортированного списка во втором цикле. На рис. 3.8 показаны изменения, вносимые программой 3.11 в течение одной операции цикла. Из списка ввода извлекается следующий узел, вычисляется его позиция в списке вывода, и реализуется ссылка на эту позицию.

РИСУНОК 3.8 СОРТИРОВКА СВЯЗНОГО СПИСКА

Представлена динамика обработки одного из элементов исходного списка. Исходный список (содержит узлы a) и упорядоченный список (содержит узлы b) с использованием сортировки вставками. Сначала берется первый элемент исходного списка и определяется его место сортировки x (функция `findplace`). Затем выполняется поиск x в b для нахождения места, где x должен вставиться (функция `insert`). Если $x = \text{empty}$ - `insert` b - `empty` (или $x = \text{empty} = \text{NIL}$), и x вставляется в список b (функция `insert`). Если x не `empty`, то элемент x вставляется в список b между двумя указателями a и b , следующим за a и перед b , сохраняя список b упорядоченным (функция `insert`). По окончании цикла список a становится пустым, а список b будет содержать все узлы в упорядоченном виде.



Программа 3.11 Сортировка методом вставки в список

Этот код генерирует N случайных целых чисел в диапазоне от 0 до 999, создает связный список, и в порядке каждого узла: генерирует случайную длину списка (начиная с 1 и до 10), затем переставляет узлы в соответствии с методом сортировки в порядке их сортировки. Для выполнения этой процедуры необходимо создать два списка — список исходный (нессортированный) и список новый (сортированный). В каждой итерации цикла на вновь созданный узел u выполняется встраивание в определенную позицию списка нового. Код упрощен. Благодаря использованию процедуры `insert` в каждом списке, содержащих ссылки на первый узел. В следующем разделе также рассмотрим конструктор, который мы будем использовать для создания.

```
node insert(0, 0): link u = &head, t = u;
for (link a = 0; a < N; a++)
    t = (1->rand + 1000 * rand() % 1000, 0);
node head(0, 0): link u, a, b = &head;
for (t = a->next: t != 0: t = u)
{
    u = t->next;
    for (x = b; x->next != 0; x = x->next)
        if (x->next->data > t->data) break;
    t->next = x->next; x->next = t;
}
```

Главная причина использования процедуры `insert` становится очевидной, если рассмотреть процесс дублирования первого узла в несортированном списке. Этот узел имеет наименьший элемент в списке, и его можно вставить в любое место списка. Существуют три возможности:

- Дублировать указатель `list`, который обнаруживает наименьший элемент и создает список из одного узла таким же образом, как в программе 3.9.
- Перед каждой вставкой узла проверить, является ли список изначально пустым.
- Использовать фиктивный первый узел, ссылка которого указывает на первый узел списка, как в данном примере.

Первому варианту не хватает изящества. Он требует дублирования кода. То же относится и ко второму варианту.

Использование фиктивного узла вносит дополнительные затраты памяти (на ссылки на первый узел). Но зачастую типовых приложений без него можно обойтись. Например, в программе 3.10 присутствует список входных (исходный) и список выходных (сортированный). Это нет необходимости использовать фиктивный узел, поскольку все вставки выполняются в начале списка нового. Другой пример в других приложениях, в которых используются фиктивные узлы упрощают код эффективно, но если применение по 1 ссылке в каждом списке. Не существует жесткого правила принятия решения об использовании фиктивных узлов — выбор зависит от связи в комбинации с соотношениями быстродействия. Хорошие программы подчеркивают изящество, которое более всего упрощает задачу. На протяжении книги встречаются несколько подобных комбинаций.

Еще предстоит ознакомиться с списком списков, приведенным в табл. 3.1 для справочных целей. Остальные рассуждения в приложении. Но все же важно отметить 3.1 для списка из списка, непонятных указателей `head` и хранения соединений, при ко-

торые программы управляют движением указов. Используя данные код для различных операций: добавление в память под указ и ее освобождение, а также выполнение указов информации о типе кода для всех соединений. Для живучести функций, реализующих те же операции, необходим дополнительный код проверки условий ошибок. Цель таблицы заключается в демонстрации скорости и различия подкатитоп.

Таблица 3.1 Соглашения о ведущем и завершающем узлах в связном списке

Эта таблица предоставляет примеры для базовых операций обработки списков, с которыми мы часто используем соглашения. Подобный код используется в программах приложения с линейной связной структурой обработки списков.

Список циклический, никогда не бывает пустым

```

добавить элемент:      head->next = head;
вставить в начало:     t->next = head->next; head->next = t;
удаление конца:       x->next = x->next->next;
очищать список:        t = head;
                        do { t->next = t->next->next; } while (t->next != head);
проверка на наличие цикла: if (head->next == head)

```

Ведущий указатель, null-указатель завершающего узла

```

инициализация:         head = 0;
вставка в конец:       if (!x == 0) { head = t; head->next = 0; }
                        else { t->next = x->next; x->next = t; }
удаление конца:        t = x->next; x->next = t->next;
итерация:              for (t = head; t != 0; t = t->next)
проверка на пустоту:   if (head == 0)

```

Фиктивный ведущий узел, null-указатель завершающего узла

```

инициализация:         head = new node;
                        head->next = 0;
вставка в конец:       t->next = x->next; x->next = t;
удаление конца:       t = x->next; x->next = t->next;
итерация:              for (t = head->next; t != 0; t = t->next)
проверка на пустоту:   if (head->next == 0)

```

Фиктивные ведущий и завершающий узлы

```

инициализация:         head = new node;
                        x = new node;
                        head->next = x; x->next = x;
вставка в конец:       t->next = x->next; x->next = t;
удаление конца:       x->next = x->next->next;
итерация:              for (t = head->next; t != x; t = t->next)
проверка на пустоту:   if (head->next == x)

```

Другая важная ситуация, в которой иногда удобно использовать ведущий узел, возникает, когда необходимо перебрать список элементов в качестве аргументов функции. Эти аргументы могут изменять список таким же образом, как это выполняется для массивов. Использование ведущего узла позволяет функции принимать как

получить пустой список. При отсутствии галунета эта функция нуждается в модифицированных вызывающей функции в аргументе, когда возвращен пустой список. Одно из решений для C++ состоит в передаче указателя на список как дополнительного параметра. Второй механизм предусматривает прием функции обработки списков указателей на список, тогда в качестве аргументов и возврат указателей на списки выводит. Он уже использован в программе 3.10. Этот принцип «отрастил необходимость использования указателей». Более того, он очень удобен для рекурсивной обработки списков, которая часто используется в этой книге (см. раздел 3.1).

Программа 3.12 Интерфейс обработки списка

В этом коде, который можно сохранить в файле интерфейса List.h, список и типы узлов в списке, включая структуру, в соответствии с ней. Для реализации можно использовать стандартные функции обработки объектов, соответствующие функции. Функция connect применяется для удобства реализации. Эти функции должны соответствовать указанным условиям. Они не зависят от особенностей реализации. Как будет показано в главе 4, несколько отличий, основанный на классах C++, можно объединить независимо от конкретной реализации от особенностей реализации.

```
typedef int Ttype;
struct node { Ttype data; node *next; };
typedef node *Link;
typedef Link Node;

void connect(Link);
Node newNode(int);
void deleteNode(Node);
void insert(Node, Node);
Node remove(Node);
Node next(Node);
Node first(Node);
```

Программа 3.12 объявляет набор функций 'черного ящика', которые реализуют базовый список операций. Это позволяет избежать повторения кода и уменьшения деталей реализации. Программа 3.12 реализует набор функций (см. программу 3.9), преобразованная в клиентскую программу, которая использует этот интерфейс. Как и функция, важная операция, используемых в вычислениях. В описании их в интерфейсе обеспечивают гибкость, которая позволит рассмотреть различные конкретные реализации операций и проанализировать их эффективность. В результате реализации операций программа 3.12 (см. программу 3.14), но возможны альтернативные решения, не требующие никаких изменений программы 3.12 (см. упражнение 3.11). Это тема еще будет неоднократно затрагивать в данной книге. Язык C++ включает несколько механизмов, специально предназначенных для упрощения разработки вычислительных реализаций; роль об этом пойдет в главе 4.

Программа 3.13 Организация списка для задачи Носифа

Эта программа решения задачи Носифа служит примером клиентской программы, использующей функции обработки списков, которые объявлены в программе 3.12 и реализованы в программе 3.14.


```

#include <iostream.h>
#include <stdlib.h>
#include <list.h>
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), M = atoi(argv[2]);
  Node * N;
  construct(N);
  for (i = 2, n = newNode(1); i <= M; i++)
    ( n = newNode(1); insert(x, 1); x = 0; )
  while (x != next[x])
  {
    for (i = 1; i < N; i++) x = next(x);
    deleteNode(remove(x));
  }
  cout << list(n) << endl;
  return 0;
}

```

Некоторые приложения предпочитают инкапсулировать все операции с низкого уровнями структурами данных, таких как связанные списки, путем создания функций для каждой низкоуровневой операции в интерфейсах, подобных показанному в программе 3.2. Действительно, как будет продемонстрировано в главе 4, классы C++ упрощают эту задачу. Однако пока мы достигли уровня абстракции, следствием является факт использования большого количества операций низкого уровня. В данной книге при реализации в экспериментальных интерфейсах низкоуровневые операции обычно относятся непосредственно к связанным структурам, дабы легче представить будущим разработчикам алгоритмы и структур данных. В главе 4 будут рассмотрены множества примеров.

Из этой главы мы можем вывести заключение возможности обратного перемещения по списку вперед. Например, применение функции `removeBefore(x)` позволяет удалить элемент 'назад' элемент, предшествующий данному'. В таком списке каждый узел содержит две ссылки: `next` (вперед) — указывающая на следующий элемент, а `prev` (назад) — на предыдущий. Изменение фиксированных значений `prev` и `next` для двусвязного списка позволяет обеспечить эквивалентность выражений `x->prev->next` и `x->next->prev` для каждого узла. На рис. 3.9 и 3.10 показаны основные действия со ссылками, выполняемые для реализации операций `remove(x)` (удалить), `insertAfter(x, node)` (вставить после) и `insertBefore(x, node)` (вставить перед) в двусвязном списке.

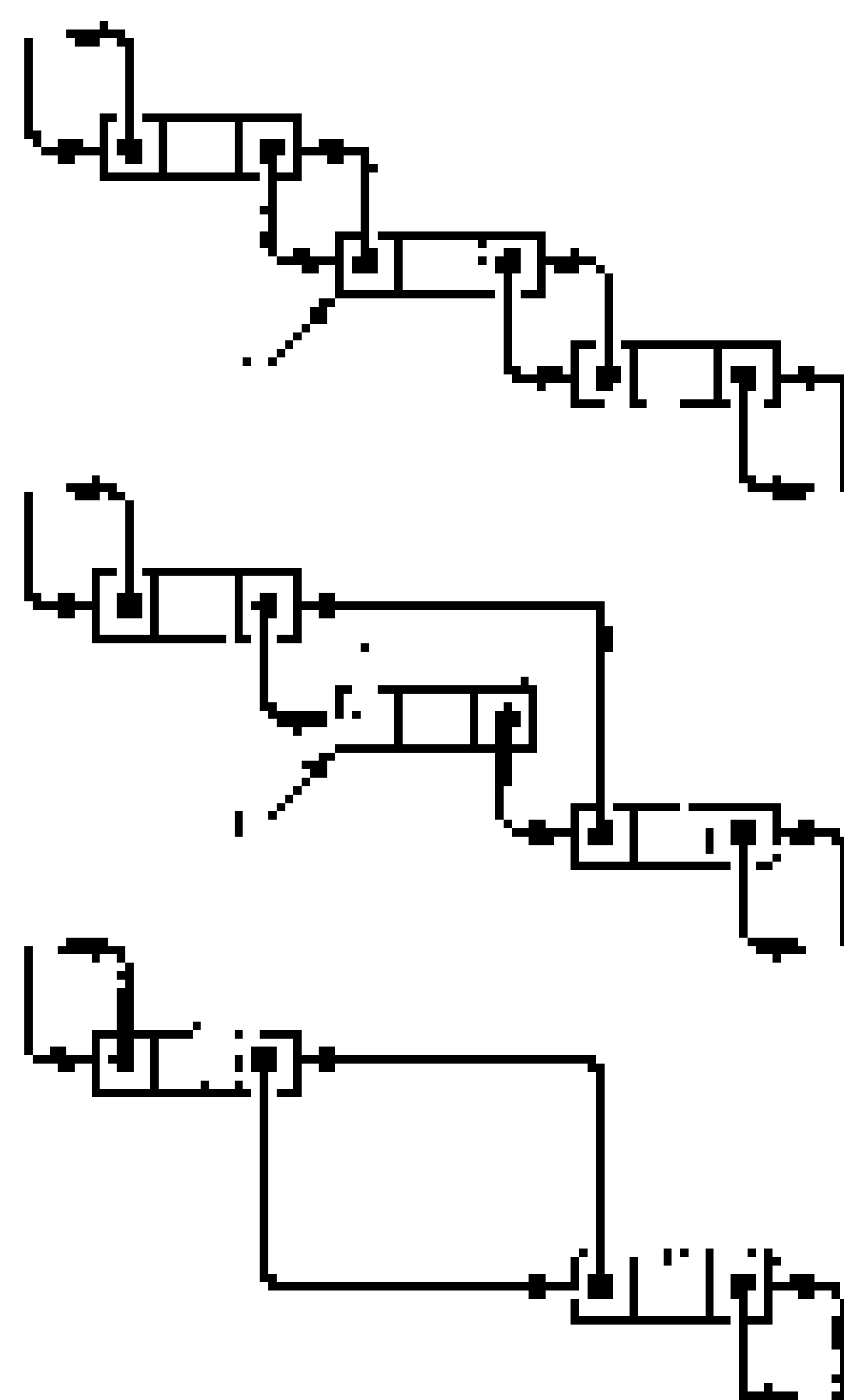


РИСУНОК 3.3 УДАЛЕНИЕ В ДВУХСВЯЗНОМ СПИСКЕ

В двусвязном списке удаление узла требует удаления связанной информации ох указателя `next`, ссылающегося на следующий для удаления узла. Однако для двусвязного списка требуется также удаление указателя `prev` предыдущего элемента. В этом случае требуется удаление `prev->next` (указатель `prev` на следующий элемент)

Обратите внимание, что для операции удаления не требуется дополнительной информации об узле, представляющем удаляемый (либо следующий за ним) в списке, как это имеет место для односвязных списков — эта информация содержится в самом узле.

Действительно, главный недостаток двухсвязных списков состоит в возможности удаления узла, когда ссылка на него является единственной информацией об узле. Типичный случай, когда ссылка передается при вызове функции в качестве аргумента, а также если узел имеет другие ссылки и сам является частью другой структуры данных. Предположим эту дополнительную возможность плечет удаление пространства, отведенного под ссылки в каждом узле. А также количество обращений к ссылке на каждую базовую операцию. Поэтому двухсвязные списки обычно не используются, если этого не требует условия. Рассмотрим последующие реализации отложен до обзора нескольких других случаев, где в этом возникает необходимость — например, в разделе 9.3.

Связные списки используются в материальном мире, во-первых, для хранения АЗТ-режимами (см. главу 1), во-вторых, в качестве компонентов более сложных структур данных. Для многих программистов связаны списки являются первым знакомством с абстрактными структурами данных, которые разработчик может непосредственно управлять. Они образуют важный элемент разработки высокоуровневых абстрактных структур данных, необходимых для решения множества задач, в том числе возможность объединения.

Упражнения

3.33 Создать функцию, которая перемещает наибольший элемент данного списка так, чтобы он стал последним узлом.

3.34 Создать функцию, которая перемещает наименьший элемент данного списка так, чтобы он стал первым узлом.

3.35 Создать функцию, которая перестроит связный список так, чтобы узлы с четными номерами следовали после узлов с нечетными номерами, сохраняя порядок сортировки четных и нечетных узлов.

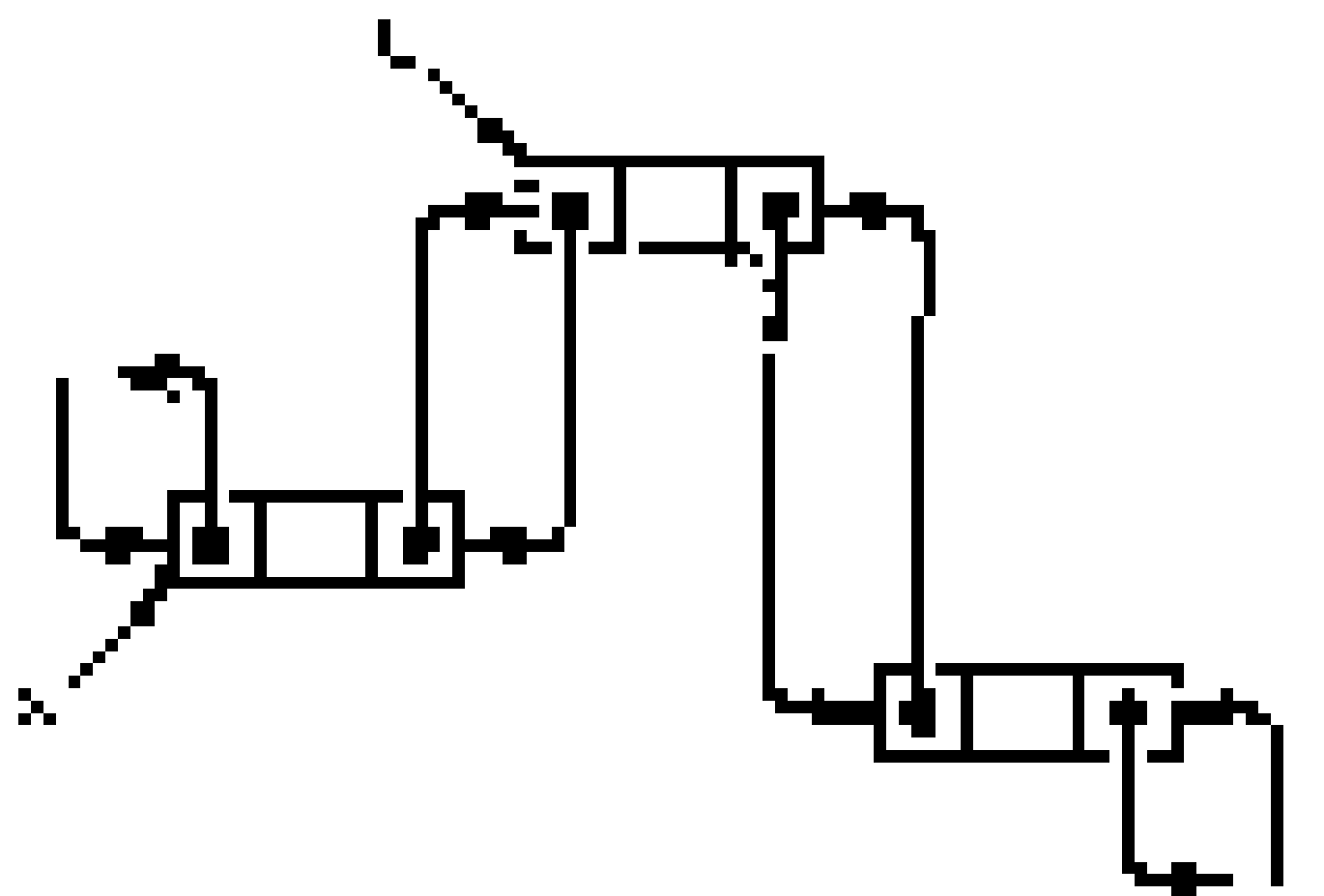
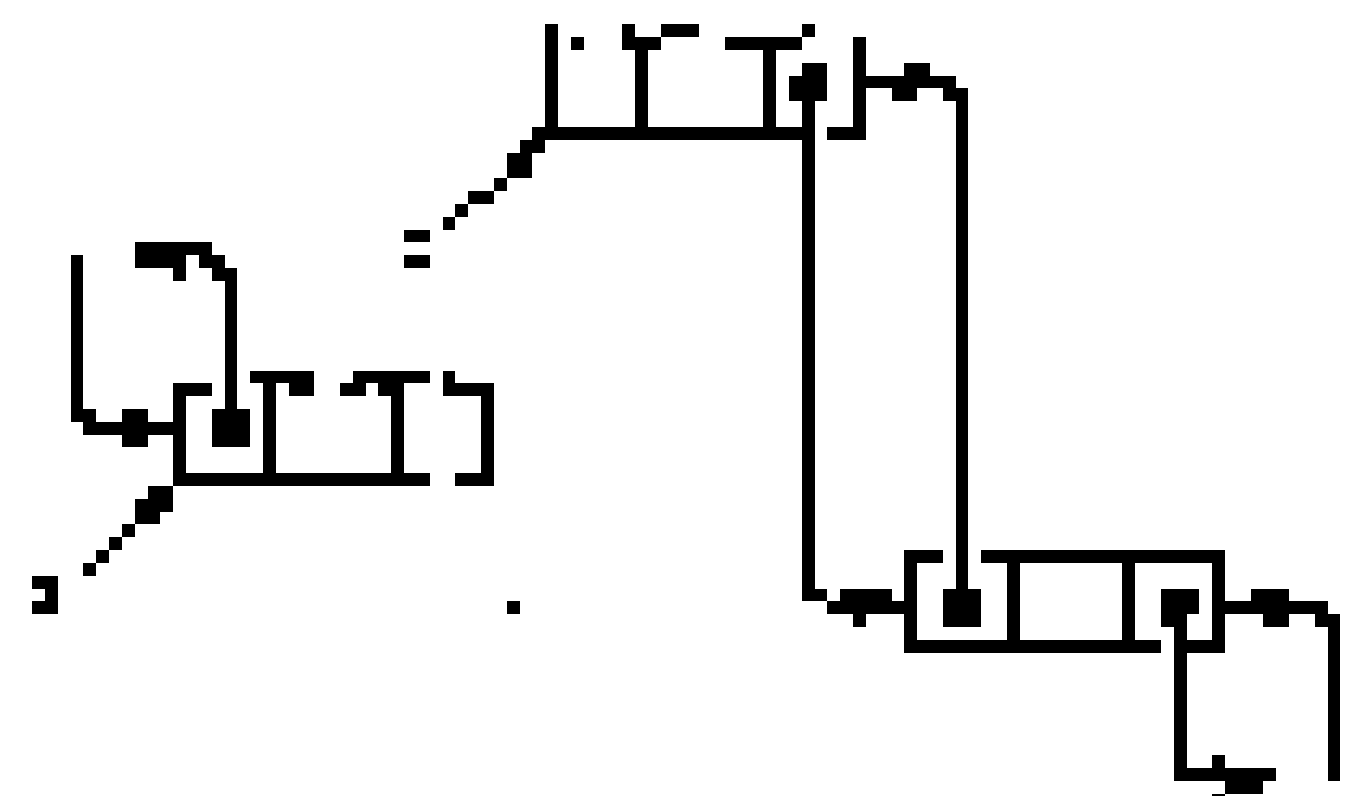
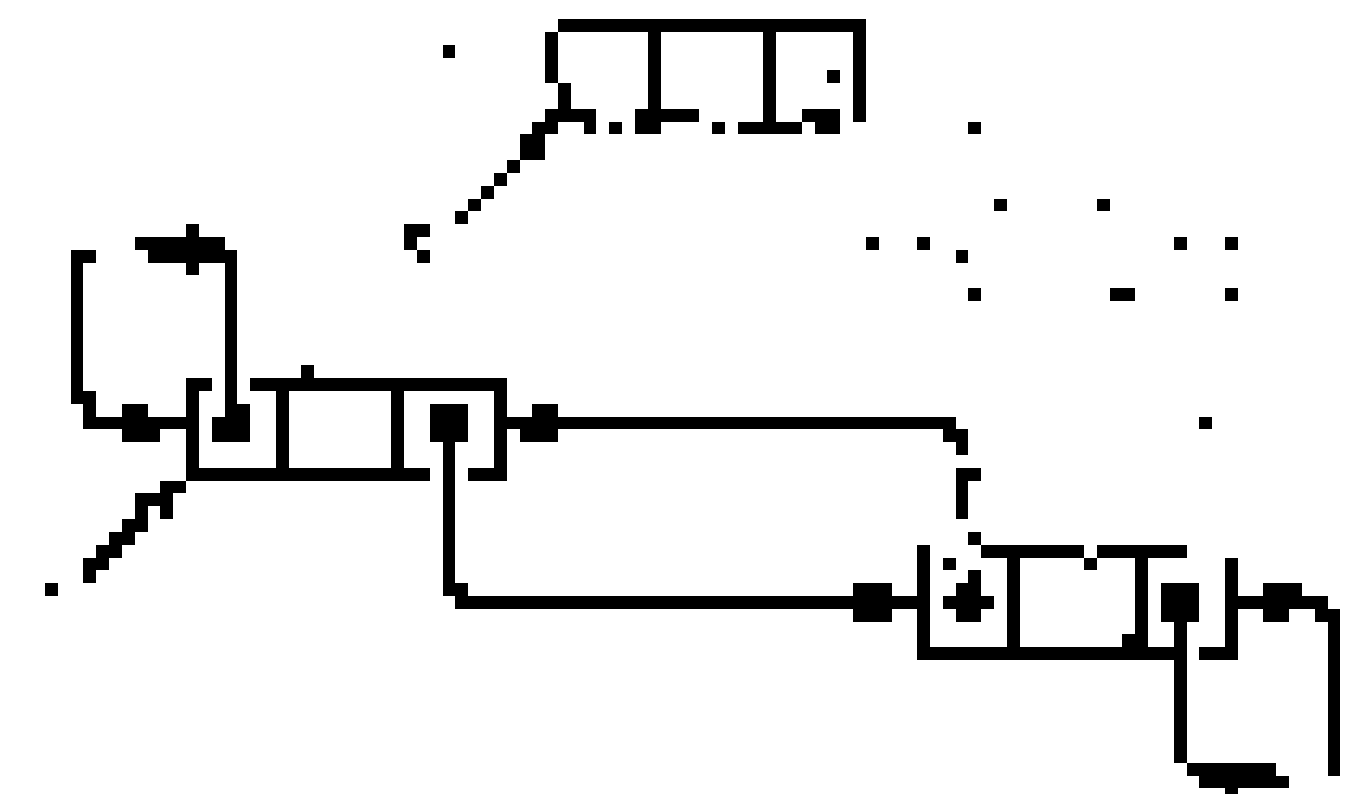


РИСУНОК 3.10 ВСТАВКА В ДВУХСВЯЗНОМ СПИСКЕ

Для вставки нового элемента в список необходимо определить, куда именно вставить элемент. Можно определить, какой узел после данного узла (или перед ним) должен быть вставлен. Для вставки узла i после узла j необходимо изменить j -ссылку на j -ссылку на элемент i (рис. 3.10). Для вставки узла i перед узлом j необходимо изменить j -ссылку на j -ссылку на элемент i (рис. 3.11). Для вставки узла i в начало списка необходимо изменить $head$ -ссылку на $head$ -ссылку на элемент i (рис. 3.12).

3.36 Реализовать функцию `copy` для списка `list`, который возвращает копию, которая следует после указат. указанным ссылкой и т.п.

с 3.37 Создать функцию, которая применяет функцию `fn` к элементу в качестве аргумента и возвращает список на основе списка `list` новый список, содержащий те же значения, что и `list`, но перемешан.

3.38 Создать функцию, принимающую два аргумента — список `list` и функцию `fn`, применяющую список в качестве аргумента — и возвращает все значения для `list` списка, для которых функция возвращает ненулевое значение.

3.39 Выполнить упражнение 3.18, но вернуть список узлов, которые проходят по порядку и возвращают ссылку на список, содержащий эти узлы, в порядке их следования в исходном списке.

3.40 Реализовать версию программы 3.10, в которой используется функциональный стиль.

3.41 Реализовать версию программы 3.11, в которой не используются функции.

3.42 Реализовать версию программы 3.9, в которой используется функциональный стиль.

3.43 Реализовать функцию, которая меняет местами два элемента списка в заданном списке.

с 3.44 Создать версию табл. 3.1 для списка, который становится пустым. На этот список требуется указать первого узла, а последний узел указывает на самого себя.

3.45 Создать версию табл. 3.1 для циклического списка, используя функциональный стиль, который требует функции и возвращаемых узлов.

3.5 Распределение памяти под списки

Преимущество списков перед массивами состоит в том, что списки эффективно уменьшают размеры с течением времени жизни. В частности, не обязательно заранее знать максимальный размер списка. Однако не нужно пренебрегать очевидным свойством: чтобы иметь возможность иметь несколько структур данных, обитавших в одном пространстве, не нужно освобождать память на определенный размер — любой размер времени.

Интерес представляет реализация оператора `new`. Напрямую, при создании узла из списка память выделяется к перераспределению памяти таким образом, чтобы узел больше не был привязан к списку. А как поступает система с пространством, которое этот узел занимает? Как система управляет пространством, чтобы всегда иметь возможность выделить его под новый узел и операторе `new`? За язык программирования существуют механизмы, которые служат для одной из примеров эффективности в реализации обработки списков.

Оператор `delete` дополняет оператор `new`. Когда блок выделенной памяти больше не используется, вызывается оператор `delete` для освобождения памяти в том, что блок доступен для дальнейшего использования. Динамическое распределение памяти (dynamic memory allocation) — это в целом управление памятью в программах действий вызовом операторов `new` и `delete` из клиентских программ. При вызове оператора `new` непосредственно из приложения, таких как программы 3.9 или 3.11, оператор вызывает

близки к значению ожидаемого размера. Это типичный случай. Альтернативный способ отслеживания памяти, доступной для распределения, заключается в том, чтобы постоянно использовать связный список! Все узлы, которые не имеют ни в один из используемых списков, можно совместно содержать в единственном связном списке. Этот список называется свободным (free list). Когда необходимо выделить пространство под узел, оно наплекнется на счет уменьшения свободного списка. При удалении узла из какого-либо списка, он включается в свободный список.

Программа 3.14 является типичной реализацией, описанной в программе 3.12, включая функции распределения памяти. Для совместной работы с программой 3.11 она дает такой же результат, что и другая реализация, с которой мы работали в программе 3.9. Поддержка свободного списка для узла фиксированного размера — чрезвычайно важна при наличии большого количества вставки и удаления узлов из списка.

Программа 3.14 Реализация интерфейса обработки списков

Эта программа реализует функции, описанные в программе 3.12, а также реализует с помощью них описанные выше под узлы фиксированного размера. Создается свободный список, который инициализируется в соответствии с максимальным числом узлов, которое может потребоваться. Для узлов выделяются. Когда программа выделяет память для узла, он удаляется из свободного списка. Когда программа удаляет узел, он помещается в свободный список.

По соглашению между программой и функцией `addNode` узлы с нулевым значением `next` являются терминальными узлами типа `Node` и используются так же, как и другие программы. Функции, реализующие интерфейс. Для этих функций в качестве параметров узлы имеют ссылки на списки себе. Эти соглашения служат средством защиты от ошибок на неопределенные узлы или в какой-то мере, гарантируют, что функция использует интерфейс для работы с узлами. В языке C++ эти соглашения реализуются с помощью анонимных классов, относящихся к структуре `Node` (см. пункт 4).

```

#include <stdlib.h>
#include "list.h"
link newList()
void contract(int N)
{
    linklist = new Node[N+1];
    for (int i = 0; i < N; i++)
        linklist[i].next = &linklist[i+1];
    linklist[N].next = 0;
}
link newNode(int i)
{
    link x = remove(linklist);
    x->item = i; x->next = 0;
    return x;
}
void deleteNode(link x)
{
    insert(linklist, x);
}
void insert(link x, link t)
{
    t->next = x->next; x->next = t;
}
link remove(link x)
{
    link t = x->next; x->next = t->next; return t;
}
link nextlink(link x)
{
    return x->next;
}
Item item(link x)
{
    return x->item;
}

```


Рисунок 3.11 иллюстрирует разделение гибридного списка по мере увеличения размера программы 3.11. Для простоты подразумеваются реализации связанного списка (без подсистемы учета), реализация из предыдущих глав.

Реализация абстрактности распределителя памяти в среде C++ является сложной, тем не менее мы рассмотрим простые примеры, а реализация объектов не так стандартной. Обобщения же не настолько проста, как показано в программе 3.14. Отношение к памяти различно состоит в том, что функции имеют различные требования к управлению распределением памяти хранения для разных размеров — от крошечных до огромных. Для этой цели разработано несколько алгоритмов. Другой подход, используемый некоторыми современными системами, состоит в освобождении пользователя от необходимости явно указывать размер и счет алгоритмов для каждого случая (доказательство). Эти алгоритмы автоматически выбирают все случаи, на которые не указывает ни одно правило. В этой связи также разработано несколько экспериментальных алгоритмов управления памятью хранения. Они не будут рассматриваться подробно, поскольку их характеристики обобщаются в зависимости от свойств определенных характеристик.

Программы, использующие статические списки, являются проблемой, часто эффективнее программ общего назначения, решающих те же задачи. Разделение памяти — не исключение из этого правила. Алгоритмы обработки запросов объектов хранения различного размера не имеют "изюма", что позволяет им быть адаптированными к разным фиксированным размерам, и поэтому не может использоваться этот факт. Парадоксально, но иногда причина отдала от функций является общего назначения заключается в том, что это делает программу более портативной — можно избежать от неожиданных изменений встраиваемых прикладных библиотек или переносимых в другую систему. Многие программы считают, что компиляторы имеют распределители памяти, более продвинутого, чем в программе 3.14, — умный способ разработки эффективных и переносимых программ, использующих связанные списки. Этот способ заключается в ряде алгоритмов, которые будут рассматриваться в данной книге. В них применяются подобные внешние запросы к системным управлению памятью. В остальной части книги для определения памяти применяются стандартные функции C++ new и delete.

i, j	j								
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	1	2	3	4	5	6	7	8	9
3	1	2	3	4	5	6	7	8	9
4	1	2	3	4	5	6	7	8	9
5	1	2	3	4	5	6	7	8	9
6	1	2	3	4	5	6	7	8	9
7	1	2	3	4	5	6	7	8	9
8	1	2	3	4	5	6	7	8	9
9	1	2	3	4	5	6	7	8	9

РИСУНОК 3.11 ПРЕДСТАВЛЕНИЕ СВЯЗНОГО СПИСКА И СВОБОДНОГО СПИСКА В ВИДЕ МАССИВОВ

Это изображение 3.11 демонстрирует разделение гибридного списка по мере увеличения размера программы. Для каждого размера списка (от 1 до 9) показаны все возможные варианты распределения памяти. В конце каждого гибридного списка (от 1 до 9) показаны все возможные варианты распределения памяти. Например, для списка размером 1, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 2, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 3, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 4, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 5, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 6, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 7, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 8, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9. Для списка размером 9, можно выделить 1, 2, 3, 4, 5, 6, 7, 8, 9.

Упражнения

- 3.46 Написать программу, которая удаляет все узлы связанного списка (вызывает операцию delete с указателем).
- 3.47 Написать программу, которая удаляет узлы связанного списка, находящиеся в позициях с номерами, кратными 3.
- 3.48 Написать программу, которая удаляет узлы в четных позициях связанного списка.
- 3.49 Реализовать интерфейс программы 3.12 с помощью прямого использования операций new и delete в функциях newNode и deleteNode соответственно.
- 3.50 Эмпирейно оценить времена выполнения функций из предыдущих задач на программах 3.14 с операторами new и delete (см. упражнение 3.49) для программ 3.13 при $M = 2$ и $N = 10^1, 10^2, 10^3$ и 10^4 .
- 3.51 Реализовать интерфейс программы 3.12, используя ссылки: массив (при отсутствии пустого узла) вместо указателей таким образом, чтобы рис. 3.11 иллюстрировал операции программы.
- 3.52 Предположим, что имеется набор узлов без null-ссылок (каждый узел указывает на своего соседа либо на другой узел набора). Доказать, что если следовать ссылкам, начиная с любого узла, получится цикл.
- 3.53 При выполнении условий упражнения 3.52 записать фрагмент кода, отыскивающий по данному указателю узла количество различных узлов, которые будут достигнуты, если следовать ссылкам от данного узла. Модификация любых узлов не допускается. Не использовать более некоторого постоянного объема вычислительного пространства памяти.
- 3.54 При выполнении условий упражнения 3.53 написать функцию, которая определяет, указывают ли в одном цикле две данные ссылки, если их отслеживать.

3.6 Строки

В языке C термин "строка" (string) обозначает массив символов переменной длины, определяемый начальной точкой и символом завершения строки. Язык C++ расширяет эту структуру данных из C. Кроме того, строки в качестве высокоуровневой абстракции включают и стандартную библиотеку. В этом разделе придется рассмотреть пример строки в стиле C. Целесообразно считать строку в качестве низкоуровневой структур данных обусловленной двумя главными причинами. Во-первых, многие вычислительные приложения предусматривают обработку текстовых данных, которые могут представляться непосредственно строками. Во-вторых, многие значительные системы предоставляют прямой и эффективный доступ к файлам данных, которые в основном соответствуют символам строк. Таким образом, в подавляющем большинстве случаев абстракция строк связана потребностью приложения с возможностью конъектуры.

Абстрактные понятие последовательности символов, завершаемая символом конца строки, может быть реализована множеством способов. Например, можно непосредственно символный список, но в этом случае для каждого символа требуется содержать по одному указателю. Язык C++ наследует из C эффективную репликацию на основе массивов, которая реализована в явном виде, а также предоставляется более аб-

менную реализацию на стандартной библиотеке. Остальные реализации описаны в главе 4.

Различие между строкой и массивом символов связано с понятием длины. В обоих случаях представляется непрерывная область памяти, но длина массива устанавливается в момент его создания, а длина строки может измениться в процессе выполнения программы. Это различие имеет интересные последствия, которые мы позже обсудим.

Для строки необходимо зарезервировать память либо во время компиляции, либо объявляя массив символов с фиксированной длиной, либо по правя типологии, через вызов функции `new[]`. Однако запись в память массиву стр можно записывать символы с начала и до символа завершения строки. Без символа завершения строка представляет собой обрывочный кусок смысла. Символ завершения строки позволяет принимать более высокий уровень абстракции, рассматривая только часть массива (от начала до символа завершения) как содержащую значимую информацию. Символ завершения имеет значение 0, а также обозначает как '\0'.

Например, чтобы найти длину строки, можно посчитать количество символов от начала и до символа завершения. В табл. 3.2 представлены прочие операции, которые часто выполняются со строками. Все они подразумевают сканирование строки с начала и до конца. Многие из этих функций содержатся в библиотеках, объявленных в файле `<string.h>`. Однако представьте для себя их реализацию, часто требуется слегка изменить переименование и линейный код. Для ясности функции, рассмотренные в этой главе, написаны достаточно простым и понятным образом. Этот код представлен здесь не только с целью демонстрации его простоты, но и для наглядной демонстрации характеристик программируемости.

Таблица 3.2 Элементарные операции со строками

В этой библиотеке знание языка не является анализом, поскольку для реализации различных примитивов языка C++ инженерные ухищрения делают код более сложным, а код, написанный на более высокоуровневом языке, служит более эффективным методом выражения алгоритма и делает код более простым для понимания. Основная сложность для парной с абстракцией указателей заключается в операции для версий, в которой применяется индексированный массив. Операции наравленные для сравнения для массивов с указателями получаются таким же образом, как и для версий с индексированным массивом, потому что сущность. Внимательное выполнение всех этих задач требует, чтобы длина строки

Версия с индексированным массивом

Вычисление длины строки (`strlen(a)`)

```
for (i = 0; a[i] != 0; i++) ; return i;
```

Копирование (`strcpy(a, b)`)

```
for (i = 0; (a[i] = b[i]) != 0; i++) ;
```

Сравнение (`strcmp(a, b)`)

```
for (i = 0; a[i] == b[i] != 0; i++) ;
if (a[i] == 0) return 0;
return a[i] - b[i];
```

Сравнение (`strcmp(a, b, n)`)

```
for (i = 0; i < n && a[i] != 0; i++)
    if (a[i] != b[i]) return a[i] - b[i];
return 0;
```



```

(функция) def strlen(m, b):
    return (a-strlen(m), b)

Эквивалентные выражения с указателями
Вычисление длины строки (strlen(a)):
L = m; while (*L != '\0'); return L - m + 1;

Компьютер (strlen(a, b))
while (*a == *b++) .

(функция) def isstr(m, b):
    while (*m == *b+1)
        if (*a-1) == 0) return 0,
    return *(m-1) == *(b-1);

```

Одной из наиболее важных является операция *сравнения* (compare). Она указывает, как из двух строк выбрать первой значимой и второй (difference). Для простоты изложения предполагается, что строка *используемая* (поскольку реальная строка для строк, содержащих только пустоты, буквы прописные и прописные регистра, числа и др. довольно сложны), а строка *сравниваемая* исключительно из нулей и единиц. Такой подход называется *лексикографическим*. Кроме того, функция сравнения используется для определения эквивалентности строк. Но для этого функция сравнения возвращает отрицательное число, если строка первого аргумента меньше, а строка второй строки, положительное число в противном случае и ноль, когда строки идентичны. Важно отметить, что выполнение проверки равенства возможно, что определение, равны ли две строки строк — если два указателя строк равны, то же относится к указателям строк (это одно и то же строка), но возможны различные указатели строк, которые ссылаются на равные строки (идентичные последовательности символов). Управление указателями в строках с последующей обработкой либо доступом к ней, путем сравнения строк, строк, строк, но множество приложений. Поэтому операции сравнения имеют особый интерес. Хороший пример содержится в главе 3.3, а также во многих других местах книги.

Программа 3.15 является реализацией простой задачи обработки строк. Она печатывает позицию, где короткая обрабатываемая строка содержится внутри длинной строки текста. Для этой задачи разработано несколько сложных алгоритмов, но самый простой алгоритм наводит на мысль о простоте реализации, используемая при обработке строк в языке C++.

Обработка строк служит убедительным примером эффективности сделанных в библиотечной библиотеке функций. Дело в том, что даже небольшие библиотечные функции могут превышать интуитивно ожидаемый результат. Например, анализ строки строки занимает время, сравнимое со временем. Игнорирование этого факта может привести к серьезным проблемам, связанным с производительностью. Например, после краткого обзора библиотеки можно реализовать соответствующий образ из программы 3.15 следующим образом:

```

for (i = 0; i < strlen(m); i++)
    if (strcmp(m[i], p, strlen(p)) == 0)
        cout << i << " ";

```


К сожалению, время выполнения этого фрагмента кода пропорционально, как минимум, квадрату длины n независимо от того, где цикл, поскольку для определения ширины строки в каждый раз выполняется ее полный перебор. Это избыток времени пелики и даже неприемлемый затратам при проверке наличия определенного слова в большой книге (содержащей более миллиона символов) требует триллионов операций. Подобные проблемы трудно обнаружить, поскольку программа может хорошо работать в небольших строках и на этих этапах. Но стоит не замечаться, если не полностью затеряться при решении реальных задач. Вдох! Глубоко! А эту проблему можно решить только будучи осведомленным о них!

Программа 3.15 Поиск строки

Эта программа обнаруживает все вхождения введенного на командной строке слова в строке текста (предполагается наличие запятой для endl). Строка текста объявляется в виде массива символов фиксированной длины (можно также использовать `std::string`, как в программе 3.6). Чтение строки выполняется из стандартного ввода с помощью функции `getline()`. Задача для слова (аргумента командной строки) делается системой перед вызовом программы, а указатель строки содержится в элементе `argv[1]`. Для каждой начальной позиции i в строке s производится только один вызов подстроки, строка начинается с этой позиции, со строкой p (как и в предыдущей программе) строка p символов. Как и в предыдущей программе конец строки p печатается начальной позицией i следования этого слова в текст.

```
#include <iostream.h>
#include <string.h>
#define N 1000000
int main(int argc, char *argv[])
{
    char a; char b;
    char s[N], *p = argv[1];
    for (i = 0; i < N-1; i++) a[i] = 0; i++;
    if (!argv[1]) break;
    s[i] = 0;
    for (i = 0; a[i] != 0; i++)
    {
        int j;
        for (j = 0; p[j] != 0; j++)
            if (s[i+j] != p[j]) break;
        if (p[j] == 0) cout << i << " ";
    }
    cout << endl;
}
```

Этот код, как и код предыдущей программы (программа 3.14), неэффективен, поскольку код может быть корректным, но его выполнение окажется не самым эффективным, как оказалось. Простая идея превратить в эффективную алгоритмическую задачу можно надежнo устранить потерю эффективности подобного кода. Стандартные библиотеки позволяют написать эффективные, они же создают учить на теоретические недостатки как использование для реализации простых функций рассмотренного языка.

Для неформальной классификации, к которой мы вернемся от времени к рассмотрению, тем, что различные реализации одного и того же абстрактного понятия могут иметь различные реализации и характеристики производительности. Например, класс `string` стандартной библиотеки C++ может использоваться для хранения и может использоваться для хранения через различные промежуточные переменные, но остальные операции вы-

линялом и мышью. Для обычных приложений могут оказаться приемлемыми табличные результаты.

Довольно часто библиотечные функции не могут удовлетворять некоторым требованиям для всех приложений. Даже если принципиальность библиотечной функции хорошо обоснована (как в случае `strlen`), не уберечься, что некоторые будущие результаты не будут корректны, которые отрицательно повлияют на производительность программы. Это отображение имеет большое значение для разработки алгоритмов и структур данных, потому его следует обязательно учитывать. Остальные примеры и дополнительные варианты рассмотрятся в главе 4.

По сути, строки являются указателями на символы. В некоторых случаях требуется подход позволяющий манипулировать конкретным кодом для функций обработки строк. Например, чтобы скопировать одну строку в другую, можно написать:

```
char *str1 = "abc";
```

или:

```
int i = 0; while (i < 3) { i++; str1[i] = 'a'; }
```

либо третьим способом из табл. 3.7. (Для второго способа строки являются, по существу, кодом, который имеет различные характеристики производительности на разных компьютерах. Однако перенос с помощью используется для ясности, а третий с указателями — с точки зрения разрыва кода. Подобные рассуждения для любой конкретной реализации относятся к определенным фрагментам часто используемого кода в некоторых приложениях.)

Распределение памяти для строк сложнее, чем для обычных строк, поскольку строки имеют различный размер. Действительно, совершенно определенным механизмом резервирования пространства для строк представляется собой не это, как представление системы функция `new[]` и `delete[]`. Как указывалось в разделе 3.6, для решения этой задачи разработаны различные алгоритмы. Их характеристики зависят от архитектуры компьютера. Часто распределение памяти при работе со строками является не такой сложной проблемой, как это может показаться, поскольку используется указатель на строку, а не сам символ. Действительно, обычно мы же предполагаем, что все строки занимают непрерывную выделенную область памяти. Мы склонны предполагать, что каждая строка занимает область памяти с непрерывным адресом, но достаточно близкую, чтобы заменить строку и ее символ завершения. Следует очень тщательно обеспечить корректность при выполнении операций создания и уничтожения строк. В качестве примера в разделе 3.7 приводится программа, которая читает строки и упрощает их.

Упражнения

- 3.55 Написать программу, которая принимает строку в качестве аргумента и распечатывает таблицу, которая отображает появляющиеся в строке символы с частотой появления каждого из них.
- 3.56 Написать программу, которая определяет, является ли данная строка палиндромом (какая-либо часть строки в прямом и обратном направлении), если неори-

ровнять пробелы. Например, программа должна давать положительный ответ для строки "1 2 3 4 5 6 7".

3.57 Предположим, что для строк используется память. Создать верную функцию `isZero` и `isNot`, которые выдают память и возвращают указатель на новую результирующую строку.

3.58 Написать программу, которая принимает строку и как аргумент и читает ряд чисел (последовательностей символов, разделенных пробелами) из стандартного ввода, рассматривая их как строки, которые входят как подстроки в строку аргумента.

3.59 Написать программу, которая заменяет в длинной строке подстроки, состоящие из нескольких пробелов, одним пробелом.

3.60 Реализовать версию программы 3.13, в которой будут использоваться указатели.

- 3.61 Написать эффективную программу, которая выводит длину самой большой последовательности пробелов в длинной строке, указав при этом как можно меньшее количество символов. Подсказка: с увеличением длины последовательности пробелов программа должна выполняться быстрее.

3.7 Составные структуры данных

Массивы, списки и строки обеспечивают простую и надежную базовую модель организации данных. Они создают первый уровень абстракции, который можно использовать для группировки объектов методов, пригодных для их эффективной обработки. Наряду с этими объектами также необходимо для их хранения более сложные структуры. Возможны массивы массивов, массивы списков, массивы строк и т.д. В этом разделе представлены примеры подобных структур.

Похоже то же, как одномерные массивы, двумерные массивы, формируемые массивы, с двумя измерениями, характеризуются шириной и высотой и используются в математических пакетах. Например, следующий код можно применить для перемножения матриц `a` и `b` с получением результата в матрице `c`.

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        c[i][j] = 0;
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            c[i][j] += a[i][k]*b[k][j];
```

Математические расчеты часто естественно представляются многомерными массивами.

Понятие математических применений приводит к способу структуризации информации, состоят в использовании таблицы списков, организованных в виде строк и столбцов. В таблице данных можно записать по одной строке для каждого студента и по одному столбцу для каждого предмета. Такая таблица будет представлять двумерный массив с одним индексом для строк и другим — для столбцов. Если строк 100, а предметов 10, можно объявить массив `grades[100][10]`, а затем считать

из оценки i по студенту по j -тому предмету следующим образом: `grade[i][j]`. Для вычисления средней оценки по предмету необходимо сложить элементы соответствующего столбца и разделить сумму на количество строк. Чтобы вычислить среднюю оценку определенного студента, нужно сложить элементы строки и разделить сумму на количество столбцов и т.п. Двумерные массивы широко используются в приложениях различного типа. В программах часто целесообразно использовать более двух измерений. Например, в таблице счетов можно использовать третий индекс для учета всех таблиц на протяжении учебного года.

Двумерные массивы — это удобная форма, поскольку числа хранятся в памяти компьютера, которая, по сути, является одномерным массивом. Но в некоторых языках программирования двумерные массивы хранятся в памяти *столбцами*, а строки в одномерных массивах. Так в массиве `a[M][N]` первые N позиций будут заняты первой строкой (элементы от `a[0][0]` до `a[0][N-1]`), вторые N позиций — второй строкой (элементы от `a[1][0]` до `a[1][N-1]`) и т.д. При организации хранения в порядке старшинства строк последние строки кода переиспользуются встроены на предыдущие адреса в точности эквивалентная выражению:

$$a[M+1+j] = a[0+1-j]*N+k+j]$$

Обобщенные той же схемой применяются для массивов с большим количеством измерений. В языке C++ многомерные массивы могут реализовываться более общим способом — не можно описать в них составную структуру данных (являющуюся массивом). Эти массивы имеют гибкость, например, содержимое массивов массивов, которые имеют разный размер.

В программе 3.5 представлено решение задачи динамического выделения памяти, которая позволяет использовать программы для различных задач без какой-либо компиляции. Непосредственно подобным методом для многомерных массивов. Как выделить память многомерным массивам, размер которых известен только после компиляции? Другими словами, необходимо известность, является ли программа на элемент массива, такой как `a[i][j]`, но не объявлять массив типа `int a[M][N]` (например, поскольку значения M и N неизвестны). При организации хранения в порядке старшинства строк переиспользование

$$int* a = malloc(M*N*sizeof(int));$$

распределит память под массив только чистого размера $M \times N$, но это решение применимо не для всех случаев. Например, когда массив передается в функцию, то перед компиляцией можно не указывать только его чистого размера. Программа 3.16 демонстрирует более эффективное решение для двумерных массивов, основанное на списочном «массивном массиве».

Программа 3.16 Распределение памяти под двумерный массив

Эта функция динамически выделяет память двумерному массиву как массиву массивов. Сначала объявляется пространство массиву указателей, а затем — каждой строке. В этой функции выражение

$$int** a = malloc2d(k, b);$$

выделяет память массиву указателей чистого размера $M \times N$.


```
int *malloc2d(int r, int c)
{
    int *p1 = new int*[r];
    for (int i = 0; i < r; i++)
        p1[i] = new int[c];
    return p1;
}
```

Программа 3.12 демонстрирует использование подобной составной структуры массива строк. На первый взгляд, поскольку абстрактные типичные строки относятся к типу `string`, массивы строк следовало бы представить в виде массива, состоящего из `string`. Однако конкретные представления строк не имеют значения на этапе массива символов, поэтому массив строк также может быть массивом указателей. Как показано на рис. 3.13, эффект перераспределения строк может достигаться путем назначения указателей массива. В программе 3.17 используется библиотечная функция `qsort`. Реализация подобных функций рассмотрена в главах 6–9, а также в главе 7 — особенно. Это позволяет организовать массив для обработки строк: символы контактируют в большой одномерный массив, указатели на строки — в отдельный массив (ограниченный указателями на строки), в этом осуществляется управление указателями.

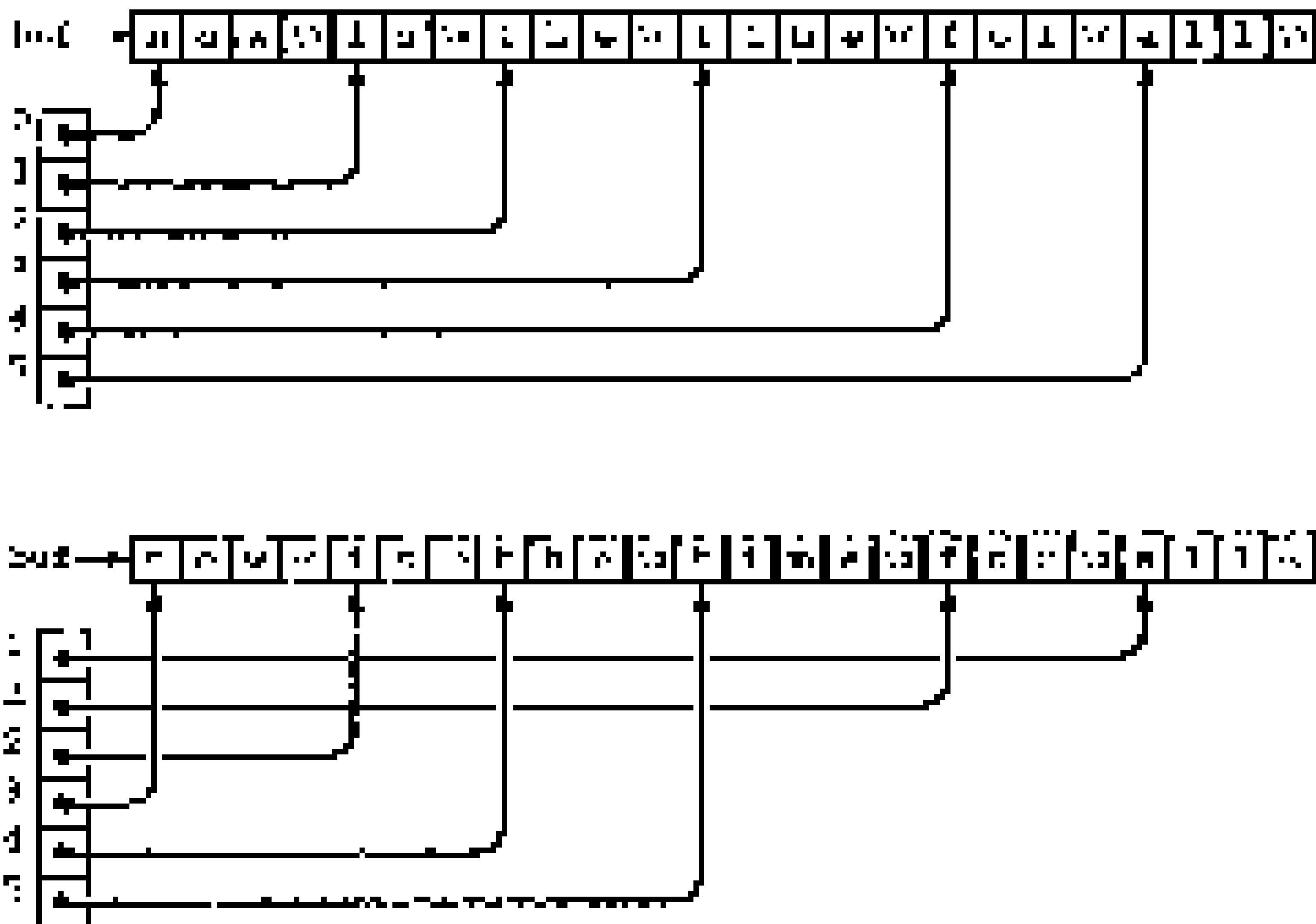


РИСУНОК 3.13 СОРТИРОВКА СТРОК

При обработке строк часто используются указатели на символ, который обозначает строку (символы строки). Поскольку массивы строк являются массивом указателей, то строки, хранящиеся в памяти, могут быть перемешаны, что приведет к перераспределению строк. Это можно сделать, используя функцию `qsort` (см. главу 9). В программе 3.17 используется библиотечная функция `qsort`. Реализация подобных функций рассмотрена в главах 6–9, а также в главе 7 — особенно. Это позволяет организовать массив для обработки строк: символы контактируют в большой одномерный массив, указатели на строки — в отдельный массив (ограниченный указателями на строки), в этом осуществляется управление указателями.

Нам уже известно, другое приращение массива строки можно вставить, перемещая для каждого строкового аргумента функцию `strcpy` на `Str + 1`. Система сохраняет в строковый буфер символы хранимой строки, подлежащие перемещению, и перемещает в произвольную указатель на массив выделенной строки этого буфера. Для вычисления чисел, соответствующих некоторым аргументам, используется функция преобразования. Остальные аргументы использовались непосредственно как строки.

Программа 3.17 Сортировка массива строк

Эта программа использует новую функцию `strcmp` из библиотеки `string.h` для сортировки строк в порядке возрастания. Строку `strcmp` — буфер, который достаточно велик, чтобы вмещать из него для каждой строки и массив выделенных указателей. Затем выполняется расположение указателей, чтобы указатель на самую младшую строку стоял перед самым младшим указателем, указателем на следующую — перед следующим и т.д.

Интересная функция `qsort`, которая действительно выполняет сортировку, принадлежит члену библиотеки `stdlib.h` — началу строки. Единственный объект, связанный с функцией `qsort`, — функция сравнения. Независимость от типа сортируемых объектов достигается за счет общего параметра `compar` типа `int`, который предоставляет объекты (в данном случае, указатели на строки), а также путем использования функции сравнения, принимающей в качестве аргументов указатели на строки. Программа выполняет преобразование этих объектов в тип указателей на указатели символов для функции `strcmp`. Чтобы обратиться к первым символам строки для отсортировки, вызывается три указателя: один для получения адреса (который является указателем) элемента массива, один для получения указателя строки (с помощью `*(char **)`) и еще один для получения символа (с помощью `*(char *)`).

Для использования другой метод достижения независимости от типа для функции сортировки и, следовательно, (см. главы 4 и 5)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int compare(const void *i, const void *j)
{ return strcmp(*(char **)i, *(char **)j); }
int main()
{
    const int Nmax = 1000;
    const int Kmax = 10000;
    char* a[Nmax], *b[K];
    char buf[Nmax], int N = 0;
    for (N = 0; N < Kmax; N++)
    {
        a[N] = buf;
        if ((N % 1000) > 0 || N == Kmax)
            N += strlen(a[N]) + 1;
    }
    qsort(a, N, sizeof(char*), compare);
    for (int i = 0; i < N; i++)
        cout << a[i] << endl;
}
```

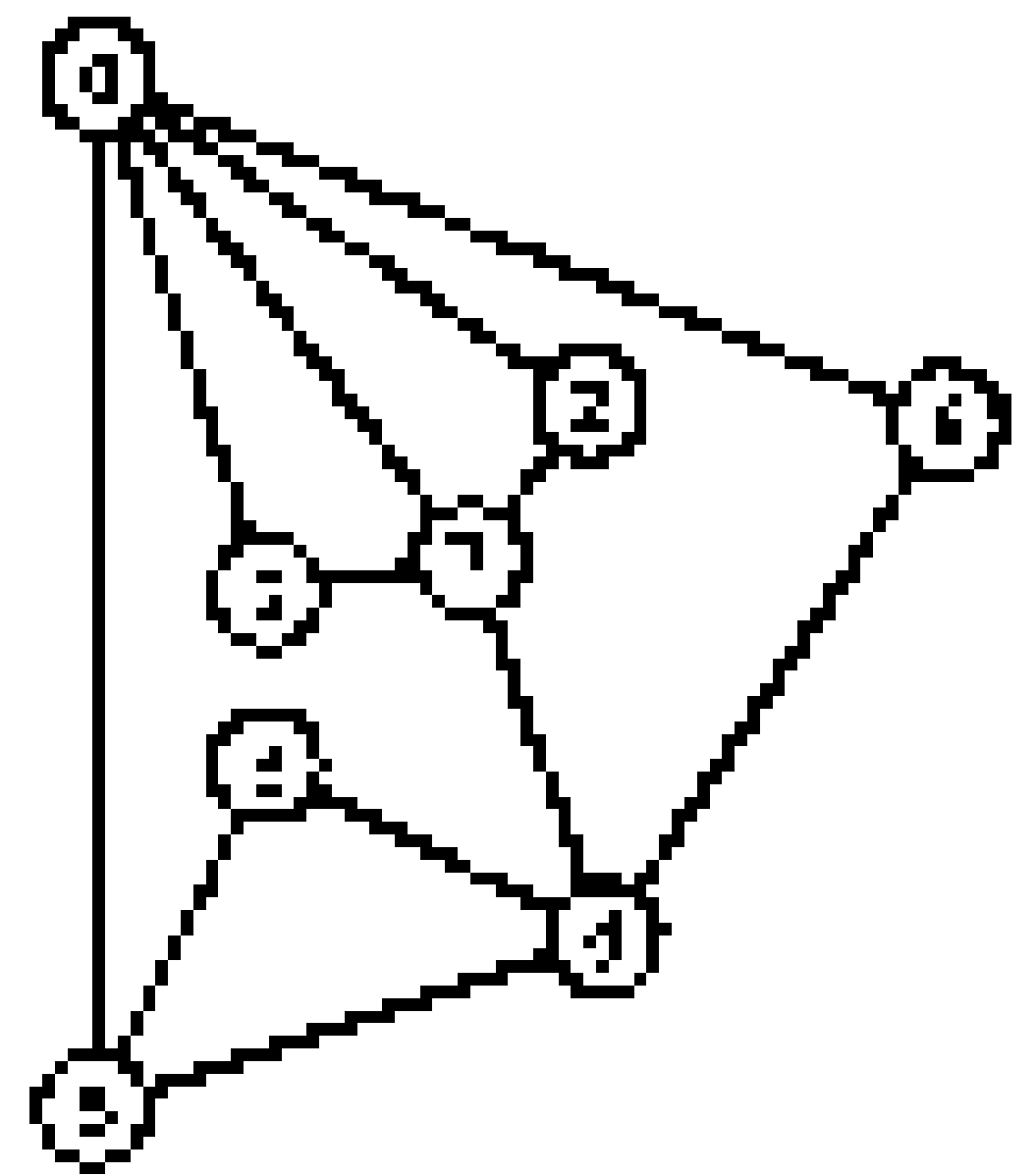

Предположим, что граф с конечным числом вершин V и конечным числом ребер E описывается ребрами из E пар шестых чисел в диапазоне от 0 до $V-1$. Это означает, что вершины обозначены целыми числами 0, 1, ..., $V-1$, а ребра описываются парами вершин. Как и в главе 1, пара $[i, j]$ обозначает связь между вершинами i и j , а также то же самое, что и пара $[j, i]$. Составленные из таких ребер графы называются неориентированными графами. Другие типы графов рассматриваются в главе 7.

Одним из простых методов представления графа является использование двумерного массива, также известного матрицы смежности (adjacency matrix). Она позволяет быстро определить, существуют ли ребра между вершинами i и j путем простой проверки наличия ненулевого значения в типичной матрице, индексировавшейся по вершинам i и j столбца j . Для неориентированных графов, к которым принадлежит рассматриваемый, при наличии связи в строке i и столбце j обязательно существует значение в строке j и столбце i . Поэтому матрица симметрична. На рис. 3.14 показан пример матрицы смежности для неориентированного графа. При графе 3.13 демонстрирует колонию матрицы смежности для заданной последовательности ребер.

Программа 3.18 Представление графа в виде матрицы смежности

Эта программа принимает в качестве входных данных ребра, образующие неориентированный граф, и создает двумерную матрицу смежности. При этом элементы $adj[i][j]$ и $adj[j][i]$ принимаются значение 1, если существует ребро от i до j или от j до i . В противном случае элемент принимается значение 0. В программе предполагается, что во время выполнения конечное число вершин V известно. Также пришлось бы добавить еще одну переменную, память под массив, представляющий матрицу смежности (см. упражнение 3.11).

```
4int adjv[100][100];
4int main()
4{
4    int i, j, adj[V][V];
4    for (i = 0; i < V; i++)
4        for (j = 0; j < V; j++)
4            adj[i][j] = 0;
4    for (i = 0; i < V; i++) adj[i][i] = 1;
4    while (cin > 0 && 0)
4        if (adj[i][j] == 0 && adj[j][i] == 0) i = 1;
4}
```



	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	1	1	1	0	0	0
4	0	0	0	1	1	1	1	0
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

РИСУНОК 3.14 ПРЕДСТАВЛЕНИЕ ГРАФА В ВИДЕ МАТРИЦЫ СМЕЖНОСТИ

Граф, показанный на рис. 3.14, имеет 8 вершин и ребра, соединяющие следующие пары вершин: (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (2, 3), (2, 4), (3, 4), (3, 5), (4, 6), (5, 6), (6, 7). Матрица смежности для этого графа (таблица 3.14) является симметричной. Матрица смежности для графа, показанного на рис. 3.13, имеет вид:

```

0 1 0 0 0 0 0 0
1 1 0 0 0 0 0 0
0 1 1 0 0 0 0 0
0 0 0 1 1 1 0 0
0 0 0 1 1 1 1 0
0 0 0 1 1 1 0 0
0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 1
```


Другой простой способ представления графа предусматривает использование массива стандартных списков, называемых списками смежности (adjacency list). Каждая вершина соответствует списку из узлов с узлами для всех вершин, связанных с ней. В неориентированном графе, если существует узел для вершины j в i -том списке, то должен существовать узел для вершины i в j -том списке. На рис. 3.15 показан пример представления неориентированного графа с помощью списка смежности. Программа 3.19 демонстрирует метод создания такого представления для входной последовательности ребер.

Программа 3.19 Представление графа в виде списка смежности

Эта программа считывает набор ребер во входном файле графа, и создает его представление в виде списка смежности. Список смежности представляет собой массив списков, по значению i в i -й вершине. В j -й список входят все k -е списки узлов, принадлежащих k -й вершине.

```
#include <iostream.h>
struct node
{
    int v; node* next;
    node(int x, node* t)
    { v = x; next = t; }
};
typedef node *link;
int main()
{
    int i, j; link adj[10];
    for (i = 0; i < 10; i++) adj[i] = 0;
    while (cin >> i >> j)
    {
        adj[i] = new node(i, adj[i]);
        adj[j] = new node(j, adj[j]);
    }
}
```

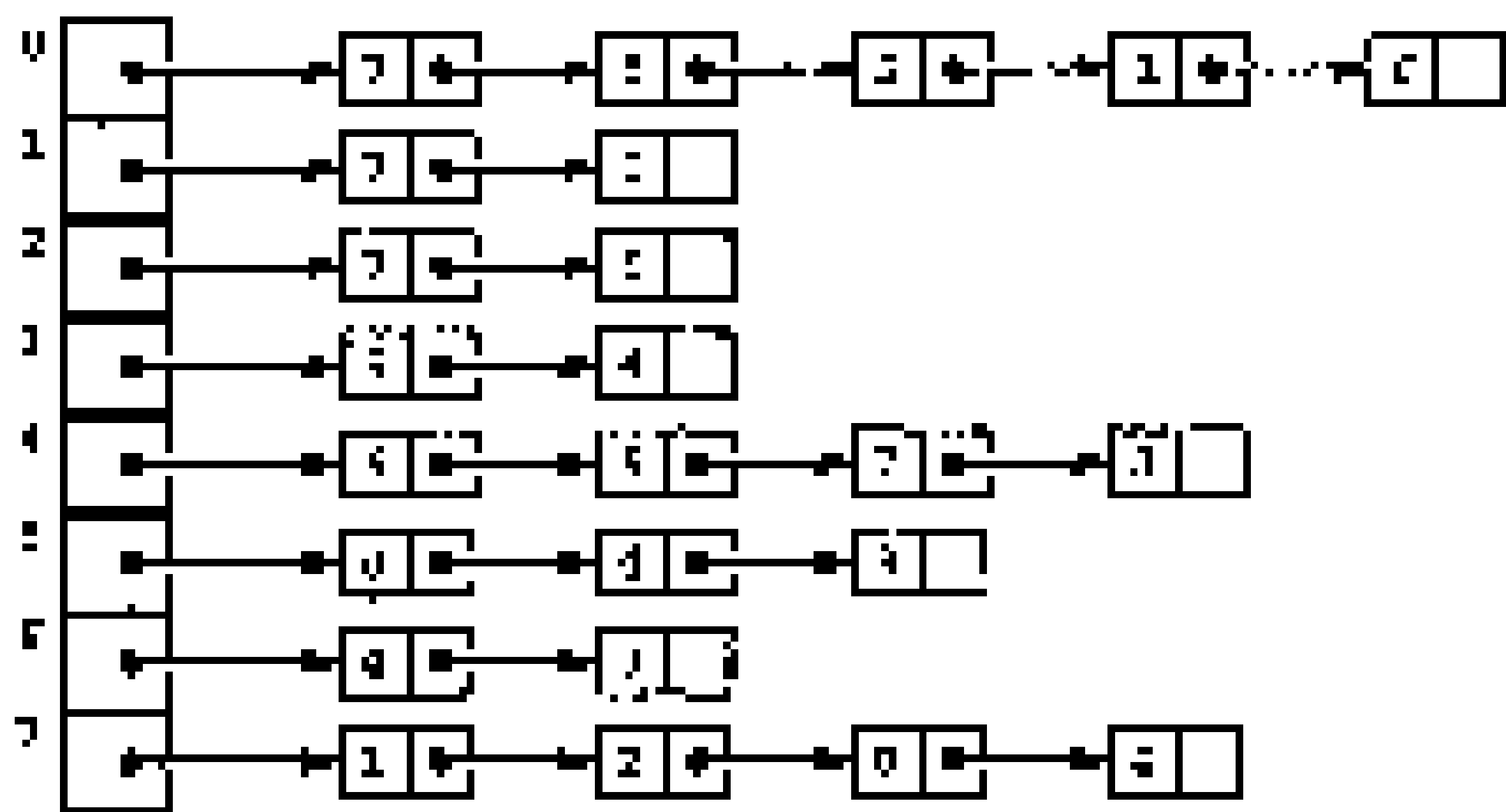


РИСУНОК 3.15 ПРЕДСТАВЛЕНИЕ ГРАФА В ВИДЕ СПИСКОВ СМЕЖНОСТИ

В этом представлении графа, изображенного на рис. 3.14, каждая вершина имеет свой собственный список смежности, представляющий список смежных вершин и ребер. Для каждой вершины i создается i -й список смежности, представляющий список смежных вершин и ребер, принадлежащих i -й вершине. Каждый список содержит по одному разу для каждой принадлежащей i -й вершины.

Оба представленных графа являются частными более простыми структурами (то одной для каждой вершины), которые состоят из ребра, соединяющего данную вершину. Для матрицы смежности более простая структура является реализацией в виде индексированного массива, а для списка смежности — в виде списка списков.

Таким образом, представленные графы являются частными случаями, демонстрирующие похожести разных пространств. Для упрощения смежности необходимо пространство, размерностью V^2 , для списка смежности размерности почти пропорционален величине $V + E$. При небольшом количестве ребер (такой граф называется разреженным), в зависимости от использования списков смежности потребует намного меньшего размера пространства. Если большинство пар вершин соединены ребрами (такой граф называется плотным), использование матрицы смежности предпочтительнее, поскольку оно не связано со сложным. Некоторые авторы считают более эффективным для представления в виде матрицы смежности, поскольку требуют постоянного затрат времени для ответа на вопрос: "существует ли ребро между вершинами i и j ". Другие алгоритмы более эффективны для представления в виде списков смежности, поскольку они позволяют обрабатывать все ребра графа за время, пропорциональное $V + E$, а не V^2 . Показательный пример этой альтернативы продемонстрируем в разделе 3.3.

Оба типа представлений можно трансформировать друг в друга при помощи таких примеров см. упражнение 3.20. Они также являются частными случаями алгоритма обработки графов, которые будут рассматриваться в главе 4.

В завершение главы рассмотрим пример, демонстрирующий использование эффективных структур данных для эффективного решения проблемы геометрической задачи, о которой шла речь в разделе 3.2. Для данной задачи необходимо найти количество пар непересекающихся отрезков внутри заданного квадрата, которые можно соединить отрезком прямой с шириной меньше d . Программа 3.20 применяет следующий подход к описанию пространства $d \times d$ с приближенным коэффициентом $1/d^2$ при заданных значениях N . Для этого в пространстве задается двудеревянный индексированный список. Плоский квадрат разбивается на сетку мелких квадратов заданного размера. Он см для каждого квадрата создается список точек всех точек, попадающих в квадрат. Двудеревянный индекс обеспечивает немедленный доступ к набору точек, который является ближайшим по отношению к данной точке. Списки списков обладают гибкостью, позволяющей хранить все точки без необходимости иметь заранее заданное количество точек, попадающих в каждый квадрат сетки.

Используемое программой 3.20 пространство пропорционально $1/d^2 \cdot N$, но для выполнения составляет $(d^2 N^2)$, что дает потенциальное преимущество перед прямой двудеревянной структурой из программы 3.2 при больших значениях d . Например, если $N = 10^4$ и $d = 0.001$ затраты времени и пространства на решение задачи будут практически линейными зависимыми от N . При этом приближенный алгоритм потребует незначительного постоянного затрат времени. Эту структуру данных можно использовать в качестве основы для решения многих других геометрических задач. Например, совместно с алгоритмом линейного-времени из главы 1 это дает быстрый и линейный алгоритм определения факта возможности соединения отрезками длиной в набор из N случайных точек на плоскости. Это фундаментальная задача не только проектирования сетей и сетей.

Программа 3.20. Двухмерный массив списков

Эта программа демонстрирует эффекты связи удаленно вложен структур. Данные на рисунке геометрически выложены на рисунке 3.8. Единичный массив представляется как сетка. Попадают двумерный массив связей списков, так что каждая точка (массив) сетки соответствует един объекту. Размер массив достаточно мал, чтобы его можно было хранить в памяти. Функция `main` создает такую же функцию из рисунка 3.15. Но она создает для себя тип `link` в `main`.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "link.h"

static node
{ point p; node *next;
  node(point pt, node* b) { p = pt; next = b; }
};

typedef node *link;
static link **grid;
static int G, W = 0; static float c;
void gridinit(float x, float y)
{ int X = x*G+1; int Y = y*W+1;
  point p; p.x = x; p.y = y;
  link A, b = new node(p, grid[X][Y]);
  for (int i = X-1; i <= X+1; i++)
    for (int j = Y-1; j <= Y+1; j++)
      for (k = grid[i][j]; k != 0; k = k->next)
        if (fabs(x-xp) < 0.5 && fabs(y-yp) < 0.5) cnt++;
  grid[X][Y] = A;
}

int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]);
  d = atof(argv[2]); c = 1/d;
  grid = malloc((G+2)*(W+2));
  for (i = 0; i < G+2; i++)
    for (int j = 0; j < W+2; j++)
      grid[i][j] = 0;
  for (i = 0; i < N; i++)
    gridnew(float(), float());
  count << int << "points within " << G << " and " << W << endl;
}
```

Как следует из примера языка рисунка, данные различных типов можно объединять (каждому, либо с попыткой зациклить ссылки в объекты, а независимости объектов — в массивные объекты). Таким образом, по базовым абстрактным операциям можно строить объекты неограниченной сложности. Тем не менее, в этом примере еще не достигнуто полное обобщение структуризации данных, как будет показано в главе 3. Проще чем прежде, поскольку этот способ реализовать абстрактные структуры данных, которые можно изложить с помощью связей списков и массивов — основных средств достижения с текущего уровня обобщения.

Упражнения

- 3.62 Написать первую программу 3.16, обрабатывающую трехмерные массивы.
- 3.63 Изменить программу 3.17 для неинициализированной области памяти (вызвать память каждой строке после завершения ее от устройства ввода). Можно предположить, что длина каждой строки превышает 100 символов.
- 3.64 Написать программу заполнения двумерного массива значениями 0 и 1 путем установки значения 1 для элементов $a[i][j]$, если наибольшее общее кратное i и j равно единице, и установке значения 0 во всех остальных случаях.
- 3.65 Воспользоваться программой 3.20 и 3.4 для разработки эффективного приложения, которое определит, можно ли соединить набор из N точек отрезками длиной максимум d .
- 3.66 Написать программу преобразования разреженной матрицы из двумерного массива в массивы строк с указателями на ненулевые значения.
- 3.67 Реализовать представление матриц, представленных мультипликативом.
- 3.68 Создать матрицу смежности, построенную программой 3.18, для заданных пар значений: 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3.
- 3.69 Создать список смежности, построенный программой 3.19, для заданных пар значений: 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3.
- 3.70 Ориентированный (directed) граф — это граф, в котором существуют между вершинами定向 направления: ребра ведут из одной вершины в другую. Используя упражнения 3.68 и 3.69, предположить, что заданные пары представляют ориентированный граф, где обозначение $i-j$ указывает, что ребро из вершины i в j . Кроме того, обозначить граф, используя стрелки для указания ориентации ребер.
- 3.71 Модифицировать программу 3.18 таким образом, чтобы она принимала количество вершин в качестве аргумента замысловатой строки, а затем динамически распределяла память под матрицу смежности.
- 3.72 Модифицировать программу 3.19 таким образом, чтобы она принимала количество вершин в качестве аргумента замысловатой строки, а затем динамически распределяла память под массив строк.
- 3.73 Написать функцию, которая из заданной матрицы смежности графа для вычисления на заданных вершинах u и v количества путей из u в v с помощью наличия ребра от u до x и от x до v .
- 3.74 Написать упражнение 3.73 с использованием списков смежности.

Абстрактные типы данных

Разработка абстрактных моделей для данных и списков для обработки этих данных является фундаментальным компонентом в процессе решения задач с помощью компьютера. Примеры этого мы видели на более низком уровне в предыдущем программном уровне (глава 3, например, как обсуждалось в главе 3, мы использовали массивы и списки данных на низком уровне при решении простых задач (глава 3) было продемонстрировано в главе 1, во время использования бинарного поиска при решении задачи сортировки. В настоящей главе рассматриваются абстрактные типы данных (Abstract Data Type, в дальнейшем АДТ), позволяющие создавать программы с использованием только уровней абстракции. За счет применения абстрактных типов данных повышается возможность отделить абстрактные (концептуальные) представления, которые программы выполняют над данными, от любого конкретного представления структуры данных и любой конкретной реализации алгоритма.

Все вычислительные элементы формулируются на уровне абстракции на основе определенных физических свойств. Например, в других материалах берется абстрактная модель бита, который может принимать двоичные значения 0-1; затем на основе логических свойств элементов определено понятие бита берется абстрактная модель машины; далее, на основе принципов работы машины, с помощью программы на машинном языке берется абстрактная модель языка программирования, и, наконец, берется абстрактное понятие алгоритма, реализуемое в виде программы на языке C++ 4. Абстрактные типы данных обеспечивают возможность продолжать этот процесс дальше и разрабатывать абстрактные конструкции для

определенных числительных слов на более высоком уровне, чем это обеспечивается системой C++: они позволяют также разрабатывать абстрактные конструкции, ориентированные на конкретные приложения и подходящие для решения задач в многочисленных прикладных областях; наконец, они позволяют создавать абстрактные конструкции более высокого уровня, в которых используются эти базовые конструкции. Абстрактные типы данных представляют в наше распоряжение постоянно расширяющийся набор инструментальных средств, позволяющий применяться к решению все новых и новых задач.

С одной стороны, применение абстрактных конструкций освобождает от необходимости заботиться о реализации: с другой стороны, когда программа нуждается в производительности (быстродействии) программы, необходимо знать алгоритмы выполнения базовых операций. Мы используем много базовых абстрактных конструкций в аппаратурных версиях компьютеров и служимых основой для машинных конструкций. Мы также реализуем другие абстракции в программном обеспечении. Кроме того, мы используем еще третьи абстракции, существующие в написанных ранее с помощью программного обеспечения. Абстрактные конструкции более высокого уровня часто строятся на основе более простых конструкций. На всем уровне действует один и тот же основной принцип: в «базис программах» мы должны найти наиболее важные операции и наиболее важные характеристики данных, затем точно определить, и те и другие на абстрактном уровне и разработать эффективные конкретные механизмы для их поддержки. В настоящей главе приводятся многочисленные примеры применения этого принципа.

Для разработки нового уровня абстракции необходимо определить абстрактные объекты, с которыми необходимо манипулировать, и операции, которые должны выполняться над ними. Мы должны определить данные в некоторой структуре данных и реализовывать операции; и (важно для управления) необходимо обеспечить, чтобы эти объекты были удобно взаимодействовать для решения прикладных задач. Перечисленные замечания применимы также к простым типам данных, и базовые механизмы для поддержки типов данных, которые обсуждались в главе 3, могут быть адаптированы для таких целей. Однако язык C++ предлагает также расширение для манипуляции структурами, называемое *классами*. Классы не являются новыми при создании универсальной абстракции и поэтому рассматриваются в качестве первого инструмента, который используется для этой цели на протяжении остальной части книги.

Определение 4.1 Абстрактный тип данных (АТД) — это тип данных (любой элемент и совокупность элементов без каких значений), доступ к которому осуществляется только через интерфейсы. Интерфейс, который контролирует АТД, должен включать как минимум, а программист, в который обеспечивается спецификацией этого типа данных, реализацией.

Ключевое отличие, делающее тип данных абстрактным, заключается в том, что данные в структуре АТД элементные программы не имеют доступа к значениям данных никаким другим способом, кроме как посредством операций, реализованных в интерфейсе. Представление этих данных и функций, реализующих эти операции, находящихся в реализации и полностью отделены интерфейсом от клиента. Мы говорим, что интерфейс является *экспозицией* клиента. Клиент не может взаимодействовать через интерфейс

В программах на языке C++ это различие обычно проводится немного лучше, так как более важно качество интерфейса, нежели к нему предоставление данных: по принципам клиентских программ не разрешается прямой доступ к данным. Другими словами, разработчики клиентских программ могут иметь представление данных, но никак не образы не могут это использовать.

В качестве примера рассмотрим интерфейс типа данных для точек (программы 3.2) из раздела 3.1. В этом интерфейсе явно объявлено, что точка представляется как структура, состоящая из пары чисел с плавающей точкой, обозначающих x и y . В основном деле, подробное приращение типов данных является обычным в больших системах программного обеспечения: мы разработали набор правил относительно того, как должны быть представлены данные (а также определены ряд операций с ними интерфейсом), и делаем эти правила доступными через интерфейс, чтобы мы могли пользоваться клиентские программы, функционирующие вольноула систему. Тип данных обеспечивает взаимодействие всех частей системы с предоставляемыми основными общесистемными структурами данных. Каким бы хорошей такая стратегия ни была, она имеет один изъян, если необходимо изменить представление данных, то потребуется изменить и все клиентские программы. Программа 3.3 снизу дает нам простой пример: одна из причин разработки этого типа данных — сделать так, чтобы клиентские программы были удобно манипулировать z -точками, и мы обнаружили, что в случае необходимости у клиента будет доступ к внешним координатам точки. Но мы не можем перейти к другому представлению данных (например, к популярным экординатам, или трехмерным координатам, или даже к другим типам данных для связанных координат) без изменения всех клиентских программ.

В отличие от этого, программа 4.1 содержит реализацию абстрактного типа данных, соответствующего типу данных из программы 3.1. В этой реализации используются классы языка C++, в которых сразу определены как данные, так и операции с ними. Программа 4.2 является клиентской программой, работающей с упомянутым типом данных. Эти две программы показывают те же самые механизмы, что и программы 3.5 и 3.6. Они иллюстрируют ряд основных элементов классов, которые сейчас будут рассматриваться.

Программа 4.1 Реализация класса POINT (точка)

В этом классе определен тип данных, который состоит из набора значений, называемых для себя "точками" и "точками" (предполагается, что они интерпретируются как точки на декартовой плоскости), а также две функции-члены, специализированные для всех экземпляров класса POINT: функция POINT() которая позволяет конструировать экземпляры с заданными случайными значениями x и y и функция distance(POINT), вычисляющая расстояние до другой точки. Представление данных не имеет приватным (private) и общедоступным (public) членам, они могут также быть функциями. В свою очередь, функции-члены являются общедоступными (public) и доступны для любого класса. Как можно заметить, напомним, а также с классом POINT.cpp.

```
#include <math.h>
class POINT
{
public:
    float x, y;
```



```

public:
    POINT()
    { x = 1.0*rand()/RAND_MAX;
      y = 1.0*rand()/RAND_MAX; }
    float distance(POINT p)
    { float dx = x-p.x, dy = y-p.y;
      return sqrt(dx*dx + dy*dy); }
};

```

Когда мы закончим и программу определения параметров inp и ip , мы даже не сможем обратиться к объектам типа `POINT` (за исключением `inp`), для которых будем ссылаться на именован `i`. В языке C++ такая сущность называется объектом. При написании программы такого определения, как `POINT p`, говорят, что создается объект класса `POINT`, объектом на который осуществляется по имени `p`. В языке C++ каждый объект имеет два элемента данных, которые называются x и y . Также как и в случае структур, они не доступны по имени, только по `p.x`.

Элементы данных x и y называются данными-членами класса. В языке могут быть также определены функции-члены, которые реализуют операции, связанные с этим типом данных. Например, класс, определенный в программе 4.1, имеет две функции-члены, которые называются `POINT` и `distance`.

Классовые программы, такие как программа 4.2, могут обращаться к функциям-членам, связанным с объектом, связавшись на него по имени. Точно так же, как они обращаются по имени, находясь в какой-нибудь структуре ячеек. Например, выражение `distance(p)` вычислит расстояние между точками p и q (это же выражение даст и выражение `q.distance(p)`). Функции `POINT()`, первая функция в программе 4.1, является одной функцией-членом, пытающейся конструировать `p` из тех же же `x` и `y`, как и `p` объект, и она вызывается тогда, когда требуется создать объект этого класса. Определение `POINT p` в программе клиента приводит к распределению объекта `point` под именем `p` объект и затем (покаместом функции `POINT()`) к присвоению каждому из двух его элементов данных случайное значение в диапазоне от 0 до 1.

Программа 4.2 Программа-клиент для класса `POINT` (нахождение ближайшей точки)

Для версии программы 4.5 является клиентом, который использует АТД `POINT`, определенный в программе 4.3. Структура `main()` создает много объектов абстрактного типа `POINT` с помощью конструктора `POINT()` для инициализации каждого объекта случайными значениями `x` и `y`. Для p и q `distance(p, q)` вычисляет для объектов `p` и `q` функцию `distance` с аргументом `p`.

```

#include <iostream.h>
#include <stdlib.h>
#include "POINT.h"

int main(int argc, char *argv[])
{ float d = 1000000.0;
  int i, n, m = 0, N = atoi(argv[1]);
  POINT *a = new POINT[N];
  for (i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
      if (a[i].distance(a[j]) < d) d = a[i].distance(a[j]);
  cout << "min within " << d << endl;
}

```


Этот стиль программирования, который иногда называется объектно-ориентированным программированием, полностью поддается описанию с помощью языка C++. Можно думать о классе как о расширенной структуре, где мы не только храним данные, но также определяем операции над этими данными. Можно существовать много разных объектов, принадлежащих одному классу, но все они подобны тем, что их данные-элементы могут принимать одни и тот же набор значений и над этими данными-членами может выполняться одна и та же совокупность операций: коротко говоря, они являются экземплярами одного и того же типа данных. В объектно-ориентированном программировании мы даем объектам команды обрабатывать свои данные члены (и при необходимости исполняющие определенные функции для обработки данных, хранящиеся в объектах).

Этот небольшой класс рассматривается в качестве примера с той целью, чтобы познакомиться с некоторыми особенностями классов; поэтому он — далеко не полностью завершённый класс. В реальном мире для класса точек у нас будет намного больше операций. Например, в программе 4.1 присутствуют даже операции, позволяющие узнать значения координат x и y . Как мы увидим, добавление этих и других операций — задача довольно простая. В главе 5 мы более подробно рассмотрим классы для точек и других геометрических абстракций, например, линий и многоугольников.

В языке C++ (но не в C) у структур также могут быть члены-функции. Ключевое различие между классами и структурами связано с доступом к информации, который характеризуется ключевыми словами `private` и `public`. На простейшем уровне член класса можно увидеть только снаружи класса, в то время как на более сложном уровне член класса может считаться локальным элементом. Приватными членами могут быть как данные, так и функции: в программе 4.1 приватными являются только данные, но далее мы увидим многочисленные примеры классов, в которых приватными будут также функции-члены. Но упомянутые члены классов являются публичными, тогда как члены структур — общедоступными.

Например, в классической программе, использующей класс `POINT`, мы не можем обратиться на данные-члены `px`, `py` и т.д., как могли бы в случае, если бы программа использовала структуру `POINT`, поскольку члены классов `x` и `y` являются приватными. Все что можно предпринять, — это использовать для обработки точек общедоступные функции-члены. Такие функции имеют прямой доступ к данным-членам любого объекта этого класса. Например, когда в программе 4.1 мы вызываем функцию `display` с помощью оператора `display(q)`, в операторе `dx = x - px` имя x относится к данным-членам x в точке p (поскольку функция `display` была вызвана как функция-член экземпляра p), а имя px относится к данным-члену x в точке q (так как q — это действительный параметр, соответствующий формальному параметру w). Чтобы исключить возможную двусмысленность при путанице, можно было бы записать `dx = this->x - w.x`, — ключевое слово `this` относится к указателю на объект, для которого выполняется функция-член.

Когда x — данные-члены применяются заключенное слово `sizeof`, это, как и в случае обычных чисел, означает, что существует только одна копия этой переменной (относящаяся к классу), а не множество копий со ссылками на отдельные объекты. Это свойство часто используется, например, для отслеживания статистики, касающейся

из объектов можно в классе `Point` получить перечисленную сумму `int N`, делителей, и кон-
структор `Point()`, и тогда появится возможность знать количество созданных точек.

Конечно, можно принять за факт, что функция, вычисляющая расстояние между двумя точками, должна иметь не одну аргумент (самоу по две точки). А, как в предыдущей реализации, так и применит (две точки), что более естественно. В классе `City` этот метод можно реализовать, создав свой образ объекта в классе `POINT` и используя функцию `distance`:

```
static float distance(Point a, Point b)
{
    float dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}
```

Самостоятельную функцию может иметь доступ к члену класса, но не может получить значение для глобальной переменной.

Другая возможность альтернативно определить функцию distance как публичную. О функции вне объектов уже была POINT (необязательно + const и не в пределах абн. кл., но опускаемая записью слова static). Поскольку эта версия функции distance должна иметь доступ к приватным локальным членам класса POINT, в объявлении класса POINT необходимо включить строку

Callend float global %RPATH, %RPATH);

Формализованная (форм.) структура это функция, которая не будет изменена классом. НЗСЗ для $\mathbb{Z} \times \mathbb{Z}$ и $\mathbb{Z} \times \mathbb{Z}$ являются функциями.

Чтобы совместные программы имели возможность читать значения данных, можно использовать функции `getElementById` и `getElementByClassName`:

```
float x() const { return x; }
float y() const { return y; }
```

Эти функции определяются с помощью свойств языка, так как они не модифицируют данные, а лишь изменяют объект, для которого выполняются. Многие функции в классе `String` — функции такого типа. Обратного мнения, что если бы использовались другие представления данных, например, массивные координаты, то реализовать эти функции было бы трудно, тут не место. Я думаю, была бы полезной для компонента. Преимущество подобной гибкости можно также заметить в функциях класса — если в реализации функции `hashCode` по умолчанию вместо ссылки тип `Object` использовать символ `Object`, то при необходимости переопределения данных не придется заменять этот код. Помещая эти функции в приватную часть класса, можно избежать необходимости такой гибкости даже в тех классах, где не требуется доступ клиента к данным.

Но иногда приложения таблицы тем, содержащая классы спящих с определенным числом или данными. Например, если кто-то хочет узнать, сколько потребовалось приложений. В таких ситуациях часто обнаруживается, что использование этой таблицы данных требует таких же самых усилий, как и встроенные типы данных языка C++, например, int или bool. Эта тема рассмотрена более подробно в разделе 4.5. Один важный недостаток, касающийся этих данных, состоит в том, что они не являются частью языка C++ и поэтому не могут быть использованы в коде C++ без использования библиотек. В языке C++ есть несколько вариантов, как это можно сделать.

можно представить фундаментальную операцию и что в конечном итоге операция должна делать. Например, предполагая, что требуется считать две точки идентичными, если расстояние между ними меньше 0.001 . Добавим в класс код

```
12100 int operator==(POINT a, POINT b)
12101 { return distance(a, b) < 0.001; }
```

можно с помощью операции `==` проверять, равны ли две точки (согласно определению равенства).

Другой операцией, которую обычно желают перегрузить, является операция `<<` из класса `ostream`. При программировании на языке C++ обычно понимают, что эту операцию можно использовать для вывода значений любого объекта: в действительности же так можно делать только в том случае, если в классе определен перегруженный оператор `<<`. В классе `POINT` это можно сделать следующим образом:

```
002000 ostream& operator<<(ostream& o, POINT p)
002001 {
002002     o << "(" << p.x() << ", " << p.y() << ")";
002003     return o;
002004 }
```

Эта операция не является ни функцией членом, ни даже статической: для доступа к данным она не обязана: общедоступные функции-члены `X()` и `Y()`.

Как выглядит классы языка C++ соотносятся с моделями клиент-интерфейс-реализация и абстрактным типом данных? Они обеспечивают непереместимую явную поддержку, но достаточно расширенными наборым для факта, что существует несколько различных подходов к созданию классов, которые можно использовать. Обобщенное с помощью трех принципов: абстрактных структур функций и классов абстракции от интерфейса. Другими словами, представленные данные хранятся в приватной части класса, где они недоступны для программы, использующих класс (клиентская программа). Все, что клиентские программы "знают" о классе — это общедоступная информация о его функциях членах (имен, тип возвращаемого значения и типы аргументов). Чтобы подчеркнуть природу интерфейса (т.е. то, что он определяется посредством класса), можно рассмотреть интерфейс (как это наглядно показано в программе 4.3). Затем мы рассмотрим одну реализацию — программу 4.4. Здесь важно то, что такой порядок упрощает исследование других реализаций, с другими представлениями данных и другими реализациями функций, а также использование в совместные реализации, позволяя сделать это без каких-либо изменений клиентских программ.

Программа 4.3 Интерфейс абстрактного типа данных POINT

В языке C++ абстрактный тип данных, интерфейс, состоящий из представления АТД в виде класса, позволяющего использовать функции членов и вызовов реализации функций их объявлением (сигнатурой). Данный интерфейс можно так и получить из программы 4.1. Мы можем использовать различные реализации, включая одну и ту же реализацию, для любой реализации АТД-а в код клиентских программ, работающих с этим АТД.

```
1 class POINT
1
1 private:
1     // приватный код, связанный с реализацией
```



```
public:
    POINT();
    Point(int x, int y);
};
```

Для многих приложений возможность реализации реализации является обязательной. Например, предположим, что требуется программное обеспечение для компании, которая предоставляет варианты списков почтовых адресов потенциальных клиентов. С помощью классов C++ можно определить функции, которые позволяют клиентским программам манипулировать списками без непосредственного доступа к ним. Мы создаем функции-члены, реализующие требуемые логику. Например, можно предоставить клиентским программам интерфейс, в котором определены такие операции как *увеличить значение* или *добавить новый элемент*. Стало бы интересно узнать, почему мы не включаемся в то, что те же самые клиентские программы можно использовать даже в том случае, если потребуется изменить формат списков почтовых адресов. Это значит, что можно изменить представление данных и реализовать функции, имеющие доступ к этим данным. Без необходимости соответствующей модификации клиентских программ.

Подобную роль реализация типов данных в виде классов играет и в других языках программирования, включая C (см. главу 2). Однако, в действительности, тип данных «скрыт» поднимается этим принципом, удовлетворяет также и к типу определения абстрактного типа данных (определяется 1.1) — различие между ними заключается в вопросе точного определения таких слов, как «доступ», «изменить» и «определить». Это различие является довольно значительным, и не является для большинства языков программирования. Действительно, в определении 1.1 четко не сформулировано, что такое «доступ» и как должна быть описана тип данных и операции. Такая неопределенность является неизбежной, поскольку попытка точно выразить эту информацию и наиболее общим виде потребует использования формального математического языка и, в конце концов, приведет к трудному математическому выражению. Этот вопрос является центральным при проектировании языка программирования. Вопросы спецификации будут рассмотрены позже, после изучения нескольких примеров абстрактных типов данных.

Примеры языка C++ для реализации абстрактных типов данных с использованием абстрактного принципа, заключающегося в том, что интерфейс состоит из определений абстрактных функций, но является конкретным реализацией, поскольку интерфейс и реализация разделяются не полностью. Когда мы начинаем реализацию, нам приходится перекомпилировать программы клиенты. Некоторые паттерны типичные варианты реализации даны в разделе 4.5.

Абстрактные типы данных возникли в качестве эффективного механизма поддержки сложной организации как принципа организации данных и реализации. Они являются программным обеспечением. Они являются средством, позволяющим ограничить размер и сложность интерфейса между (потенциально сложными) алгоритмами и структурами данных, с одной стороны, и программами (потенциально — большим количеством программ), использующими эти структуры и структуры данных, с другой стороны. Это принцип управляет концепцией большого

прикладных программ в целом. Более того, абстрактные типы данных, в отличие от простых типов данных, обеспечивают гибкость, необходимую для удобного расширения или улучшения системы фундаментальных структур данных и алгоритмов. Таким образом, что интерфейс АТД определяет поведение системы пользователя и разработчика, который обеспечивает точные правила взаимодействия, причем каждый знает, что можно ожидать от другого.

В случае тщательно спроектированных АТД выделение программ элементов от реализации можно использовать различными инструментами. Например, при разработке или отладке реализации АТД обычно применяются программы драйверы. Чтобы узнать свойства программы-клиента, при построении систем в качестве заполняющей части используются также некоторые реализации абстрактных типов данных, называемые заглушками (*stubs*).

В настоящей главе абстрактные типы данных рассматриваются подробно потому, что они играют важную роль в изучении структур данных и алгоритмов. Действительно, важной движущей силой разработки почти всех алгоритмов, рассмотренных в этой книге, является стремление обеспечить эффективные реализации базовых операций для некоторых фундаментальных АТД, играющих исключительно важную роль при решении многих вычислительных задач. Проектирование абстрактного типа данных — это только первый шаг навстречу потребностям прикладных программ; необходимо также разработать конкретные реализации операций с этими операциями, а также структур данных, лежащих в основе этих операций. Эти задачи и являются темой настоящей книги. Более того, абстрактные модели используются непосредственно для разработки алгоритмов и структур данных, а также для сравнения их характеристик применимости, как это было в примере из главы 1: как прошлые разработки применяются прикладными программами, использующими АТД, для решения некоторой задачи, затем разрабатываются несколько реализаций АТД и сравниваются их эффективность. В настоящей главе этот общий процесс рассматривается подробно, со множеством примеров.

Программисты, пишущие на C++, регулярно используют как простые, так и абстрактные типы данных. Когда мы на каком-то уровне абстрагируем целые числа, используем только операции, имеющиеся в языке C++, мы, по существу, используем систему определенной абстракции для целых чисел. На какой-нибудь новой машине целые числа могут быть представлены, и операции реализованы, какими-либо другим способом, но программа, которая использует только операции, определенные для целых чисел, будет работать корректно и на новой машине. В этом случае различные операции языка C++ для целых чисел составляют интерфейс, кагда программа является клиентом, а программное и аппаратное обеспечение системы обеспечивают реализацию. Часто эти типы данных искусственно абстрактны для того, чтобы мы, применяя свои программы, могли перейти на новую машину (то, конечно, должно быть представлением для целых чисел и чисел с плавающей точкой). Однако этот пример иллюстрирует также тот факт, что такая идеальная ситуация случается не столь часто, как хотелось бы, поскольку клиентские программы могут собирать информацию о представлении данных, определяя предельные значения того или иного ресурса. Например, узнать некоторую информацию о том, как в машине представлены целые

через, некий, скребок, пишущий линии, в котором некоторое целое число умножается на два до тех пор, пока не появится сообщение об ошибке переполнения.

Эти примеры в главе 3 представляют собой программы на C++, написанные в стиле языка C. Программисты на C часто строят абстрактные интерфейсы в заголовочных файлах, пишущих набор операций для некоторой структуры данных; при этом реализация находится в каком-нибудь другом, несвязанном файле программы. Такой порядок представляет собой соглашение между компилятором и разработчиком, и оно не связано с абстрактными типами данных. Абстрактные типы данных в средах программирования на языке C. Однако многие такие библиотеки включают в себя операции для определенной структуры данных и пометку, предоставляющую собой те же данные, но не абстрактные типы данных. Например, библиотека строк языка C не пишется абстрактным типом данных, поскольку программы, работающие со строками, знают, как представляются строки (массивы символов), как они хранятся, и как к ним получить доступ через индекс массива или арифметику указателя.

В отличие от этого, классы языка C++ позволяют нам не только использовать разные реализации операций, но и создавать их на основе разных структур данных. Снять такие ключевые операции, характеризующие АТД, включается в том, что мы можем реализовать абстрактные, но модифицируем конкретные программы. Это вытекает из того требования, что доступ к типу данных должен осуществляться через абстрактный интерфейс. Ключевая особенность абстрактных классов: предоставляя прямой доступ к конкретным программам к данным. Например, можно было бы включить в стандартную библиотеку языка C++ абстрактные классы `Integer`, `Complex` и `Matrix`, скребок, представляющий строки в виде связанного списка и использовать эту реализацию без изменения конкретных программ.

На протяжении всей настоящей главы мы будем видеть различные примеры реализации АТД с помощью классов языка C++. После четкого понимания этих понятий в конце главы мы перейдем к обсуждению соответствующих философских и практических вопросов.

Упражнения

- 4.1 Предположим, необходимо посчитать количество пар точек, находящихся внутри заданного многоугольника. Чтобы решить эту задачу, создайте две абстрактные программы-клиента и реализуйте по-первым. Модифицируйте соответствующим образом функцию-член `findArea`; во-вторых, добавьте функцию-член `findArea` функции-членами `X` и `Y`.
- 4.2 В программе 4.1 к классу `Point` добавьте функцию-член, которая возвращает расстояние до начала координат.
- 4.3 В программе 4.3 модифицируйте реализацию `Point` таким образом, чтобы точки были представлены парами координат.
- 4.4 Напишите конкретную программу, которая считывает из координатной строки целое число N и выводит массив N точек, среди которых нет двух разных друг другу D_0 и D_1 точек, имеющих одинаковые координаты. Используйте перепределенную операцию `==`, определенную в тексте настоящей главы.

- 4.5. Наполюя представление на базе статичеа списка, подобно тому, что в программе 3.11, преобразуйте интерфейс обработки списка (list processing interface) к разделу 3.4 (программа 3.12) в реплицирую АПД на базе классов. Проделайте полученный интерфейс, модифицируя клетическую программу (программу 3.13) так, чтобы она использовала этот интерфейс; затем перейдите к реализации на базе массивов (см. упражнение 3.52).

4.1 Абстрактные объекты и коллекции объектов

Написание и приложенная структура данных часто содержат огромное количество разношерстной информации, и некоторые элементы этой информации могут принадлежать различным взаимосвязанным структурам данных. Например, файл персональных данных может содержать данные с именами, адресами и какой-либо иной информацией о служащих; вполне возможно, что каждая запись должна принадлежать и к структуре данных, используемой для поиска информации об отдельных служащих, и к структуре данных, используемой для сбора на запроса статистического характера, и т.д.

Несмотря на это разнообразие и сложность, в большом классе практических программ обработки данных производится обобщенное манипулирование данными, доступ к информации, связанной с этими объектами, требуется только в ограниченном числе особых случаев. Многие понятия манипуляции являются естественными процессами базовых вычислительных процедур, которые мы часто применяем выполнять в самых разнообразных приложениях. Многие из фундаментальных алгоритмов, рассматриваемых в настоящей книге, можно эффективно применять в какой-либо заданной структуре уровня абстракции, позволяющего манипулировать различными элементами этих манипуляций. Поэтому мы подробно рассмотрим многочисленные АПД, связанные с манипулированием подобным рошм. В этих АПД определены различные операции над коллекциями абстрактных объектов, не зависящие от типов самих объектов.

В разделе 3.1, в котором обсуждалось применение простых типов данных для написания программ, не зависящих от типов объектов, для дополнения типа элементов данных не зависящая от рошм. Этот подход позволяет использовать одни и те же коды для, скажем, работы с числами и числами с плавающей точкой или даже простыми численными рошм. При использовании указателей типы объектов могут быть сколь угодно сложными. При таком рошме часто приходится делать неясные предположения относительно операций, выполняемых над объектами (например, в программе 3.2 предполагается, что для объектов типа Number определены операции сложения, умножения и приведения к типу float) и, кроме того, мы не скрываем представление данных от клиентских программ. Абстрактные типы данных позволяют сделать данные более независимыми относительно операций, выполняемых над объектами данных.

В настоящей главе данной главы рассматриваются несколько примеров использования классов C++ с целью построения АПД для обобщенных объектов данных. Будет продемонстрировано создание АПД для обобщенных объектов типа Item, позволяющее написать клиентские программы, в которых объекты Item используются точно так же, как и патрочные типы данных. Т.е. все это делается так, и классе Item дано определенное количество операций, которые должны выполняться над обобщенными

объектами с помощью явного алгоритама. Нев эти характеристики объектов определяются входе в сократены от классических программ любой информации о представлении данных.

После реализации класса `Item` для обобщенных объектов (явно принятых решения несподобяются потребностям классу), попытаемся механизировать шаблоны языка C++ для написания кода, который выполняет обобщенные относительно типов объектов. Например, операция обмена для обобщенных элементов можно описать следующим образом:

```
template <class T>
void exchange (T& x, T& y)
{ T t = x; x = y; y = t; }
```

Аналогичным образом реализуются и другие простые операции для элементов. С помощью шаблонов можно писать функции классов, по одному классу для каждого типа элементов.

Разобравшись с классами обобщенных объектов, можно перейти к рассмотрению базовых (сложных) объектов. Многие структуры данных и алгоритмы применяются для реализации фундаментальных АД, представляющих собой коллекции абстрактных объектов и создаваемых на основе двух базовых операций:

- создавать новый объект в коллекции
- удалять объект из коллекции.

Такие АД называются обобщенными структурами (generalized stacks). Как правило, для удобства в них также либо включаются следующие операции: создать (создать) структуру данных (конструктор), уничтожить (создать) коллекцию объектов в структуре данных (или просто проверить, не пуст ли она). Многие создаваемые структуры, являющиеся разновидностями (вариантами) структуры данных (конструкторы) и уничтожения (создания) структуры данных (конструкторы уничтожения); эти операции объединены в разделе 4.3.

Когда объект уничтожается, программа спаренного языка, на каком объект создается в виде при удалении объекта из коллекции? В разных АД для коллекции объектов применяются различные критерии для определения того, какой объект удалить и операции удаления, и устанавливаются различные правила, связанные с этими критериями. Более того, помимо описанной операции уничтожения, мы сталкиваемся с рядом других естественных операций. Многие алгоритмы и структуры данных, рассматриваемые в книге, были спроектированы с целью обеспечения эффективной реализации различных подходов этих операций для разнообразных критериев выполнения операции удаления и других правил. Эти АД в критерии просты, неосложняются и не являются центральными в алгоритме члене данных по объектам данных, являющимся частью; они заслуживают пристального внимания.

Мы рассмотрим некоторые из этих фундаментальных структур данных, их свойства и примеры применения, в то же время мы используем их в качестве примеров, чтобы проиллюстрировать основные механизмы, используемые для разработки АД. В разделе 4.2 рассматривается структура данных (структура данных), в которой для удаления объектов используется следующий принцип: всегда удаляется объект.

добытый тип данных. В разделе 4.3 рассматриваются различные приемы создания стеков, а в разделе 4.4 — реализации стеков; предполагая при этом, что стек должен быть освобожден от привязаний. После обсуждения стеков мы перейдем к рассмотрению массивов и рассмотрим роль массивов АТД в контексте абстрактных типов. При этом важно помнить, что при исследовании в главе 1. После этого мы перейдем к рассмотрению абстрактных объектов и рассмотрим операции РТГО (которые на разных уровнях абстракции отличаются от стеков только тем, что в них применяются другие операции удаления элементов), а также объединенные операции, где запрещены повторившиеся элементы.

Как было показано в главе 3, массивы и списки стеков обеспечивают основные механизмы, позволяющие выполнять и реализовать заданные операции. Действительно, списки, списки и массивы — это структуры данных, лежащие в основе нескольких реализаций рассмотренных нами абстрактных операций. Как мы знаем, операции поиска и удаления элементов зависят от конкретной используемой структуры и от конкретного параметра или удаляемого элемента. В следующем параграфе главы 4 мы рассмотрим в том, чтобы выбрать структуру данных, позволяющую эффективно выполнять требуемые операции. В настоящей главе необходимо рассмотреть несколько примеров абстрактных типов данных, для которых операции списков и массивов обеспечивают подходящие решения. Абстрактные типы данных, обеспечивающие более эффективные операции, требуют более сложных реализаций, чем простая линейная движущая сила создания многих элементов, рассмотренных в настоящей главе.

Типы данных, которые состоят из коллекций абстрактных объектов (абстрактные операции), являются центральным объектом изучения в компьютерных науках, поскольку они непосредственно поддерживают фундаментальный метод мышления. Оказываясь, что при выполнении метода линейной обработки почти все с объектами чисел объектов, но обрабатывать их можно только поочередно — по одному объекту за один раз. Поэтому при обработке одного объекта требуется, чтобы все остальные были сохранены. Эта обработка может выполняться операциями удаления уже сохраненных объектов, или добавлением новых объектов. Во многих случаях обработка объектов и их удаление в действительности абстрактном критерии являются новой также вычислений. Как было показано, этому шаблону соответствуют многие классические структуры данных и алгоритмы.

В разделе 4.4+ рассматриваются реализации абстрактных объектов, соответствующих различным классам *списков* (списков). Некоторые из обсуждаемых структур данных реализованы в библиотеке языка C++ или ее расширениях (стандартная библиотека и библиотека — *Standard Template Library*). Во избежание путаницы, мы будем редко ссылаться на эти классы, а материалы книги представляются, начиная с языка C++.

Упражнения

- 4.6 Дайте определение для класса *Item*, в котором для проверки равенства чисел с плавающей точкой используется перестроенная структура *Item*. Определите, что такое с плавающей точкой *Item*, а также другая информация на равности, достигаемая больше (по абсолютной величине) от двух чисел, меньше чем 10^{-6} .

в 4.7 Дайте определение класса `Item` и переопределите операции `==` и `<` так, чтобы их можно было использовать в программе обработки игровых карт.

4.8 Перепишите программу 3.1 так, чтобы в ней использовался класс `Item` объектов класса `Item`, которые могут выводится при помощи операции `<`, генерироваться случайным образом с использованием статической функции `rand()` и для которых определены операции `+` и `/`.

4.2 АД для стека магазинного типа

Самый важный тип данных из тех, в которых определены операции сложения и умножения для численных объектов, называется *стек* или *магазин* (англ. *stack*).

Стек работает отчасти подобно ящику для выходящей корреспонденции во время дняного профессора: работа складывается стопкой, и каждый раз, когда у профессора появляется возможность просмотреть какую-нибудь работу, он берет ее сверху. Работа студента вполне может застрять на дне стопки на день или два, однако, скорее всего, добросовестный профессор к концу недели упрямится во всей стопкой и переберет ящик. Далее возможно будет заметить, что работа преподавателей профессора организована именно таким образом, и это вполне естественно. Они часто откладывают некоторые задачи и выполняют в это время другие: более того, зачастую им требуется в первую очередь вернуться к той задаче, которая была отложена последней. Таким образом, стеки магазинного типа являются фундаментальной структурой данных для множества алгоритмов.

Определение 4.2 *Стек магазинного типа* — это АД, который предоставляет две основные операции: *вставка* (англ. *push*) нового элемента и *удаление* (англ. *pop*) элемента, являющийся последним.

Когда мы говорили об АД *стек* магазинного типа, мы ссылались на описание операций *вставка* и *удаление*, которые должны быть определены достаточно хорошо для того, чтобы клиентская программа могла их использовать, а также на некоторую реализацию этих операций, функционирующую в соответствии с правилами удаления элементов такого стека: магазинный список, *выходная дека* (*last-in, first-out* сокращенно *LIFO*).

Программа 4.4 Интерфейс абстрактного типа данных стека

Используя то же самое общепринятое предположение, что и в программе 4.3, мы определили АД *стек* через предоставление публичных функций. Там эта предположения, что предоставление стека и любой другой код, связанный с реализацией, не является код клиентской программы. Кроме того, в этом интерфейсе применяется шрифт *чирок* для обозначения программ-классов, использующих стек, содержащий объекты любого класса (см. программы 4.5 и 4.6). В описании мы используем ключевое слово `Item` в качестве обозначения типа объектов стека (см. программы 4.7 и 4.8). Аргумент конструктора `STACK` задает максимальное количество элементов, которые можно поместить в стек.

```
template <class Item>
class STACK
```

```
{
```

```
public:
```

```
    // абстрактный код, связанный с реализацией
```



```
public:
    Stack();
    int empty(); const;
    void push(T const&);
    T& pop();
};
```

На рис. 4.1 показано, как изменяется содержимое стека в процессе выполнения серии операций `push()` и `pop()`. Каждая операция `push()` увеличивает размер стека на 1, а каждая операция `pop()` уменьшает размер стека на 1. На рисунке элементы стека перечислены в порядке их появления в стек, поэтому ясно, что самым первым элементом стека — `int 5` — элемент, который первым на него попал в стека и будет извлечен из стека, если следующей операцией будет операция `pop()`. В реализации элементы можно организовывать любым удобным способом, однако при этом у программы клиент не должна сохраняться иллюзия, что элементы извлекаются именно таким образом.

Как упоминалось в предыдущем разделе, для того чтобы можно было писать программы, использующие абстракцию стека, сначала необходимо определить интерфейс. Этот этап, как принято, объявляется совокупностью общедоступных функций-членов, которые будут предоставлять и реализация класса (см. программу 4.4). Все остальные члены класса должны быть приватными (`private`) и тем самым обеспечивать, что эти функции будут единственной связью между реализацией и применением реализации. В главе 1 и 2 мы уже видели несколько примеров абстрактных операций, из которых основаны требуемые абстракция. Однако на сегодняшний момент, по-настоящему значимые программы, в которых применяются эти абстрактные операции. Для реализации такой абстракции мы выбрали структуру данных и реализацию ее программы, которые используются механизмом компилятора. В разделе 4.3 рассматриваются примеры клиентских программ, использующих абстракцию стека, а в разделе 4.4 — соответствующая реализация.

Первая строка кода в программе 4.4 (являющейся абстрактного типа данных стек) добавляет в этот класс публичный `enum`, позволяющий клиентским программам полностью описать объект, который может добавиться в стек.

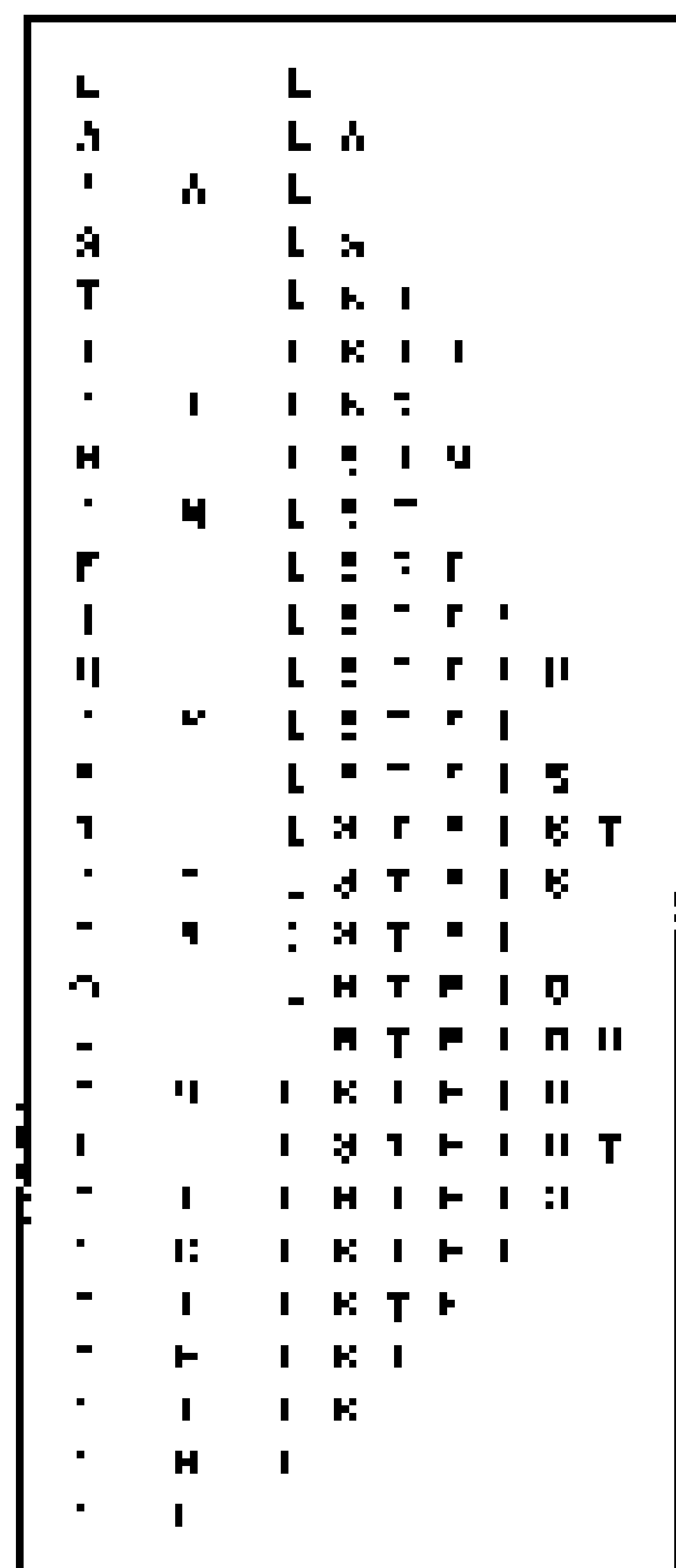


РИСУНОК 4.1 ПРИМЕР СТЕКА МАГАЗИННОГО ТИПА (ОЧЕРЕДЬ, ФУНКЦИОНИРУЮЩЕЙ ПО ПРИНЦИПУ LIFO)

Эта структура данных реализует абстракцию стека, которая может быть использована для хранения элементов, которые должны быть извлечены в обратном порядке. Например, если вы хотите использовать стек для хранения элементов, которые должны быть извлечены в обратном порядке, вы можете использовать стек. В этом случае стек будет использоваться для хранения элементов, которые должны быть извлечены в обратном порядке. Например, если вы хотите использовать стек для хранения элементов, которые должны быть извлечены в обратном порядке, вы можете использовать стек. В этом случае стек будет использоваться для хранения элементов, которые должны быть извлечены в обратном порядке.

Описание

`STACK(int) stack;`

Определим, что элементы стека тоже должны быть типа `int` (и что максимальное количество элементов, которое может вместить стек, равно `N`). Программист-клиент может создавать стеки, содержащие объекты типа `Point` или `String` или любого другого типа (или типа `String`); для этого необходимо просто изменить параметр шаблона `T` условия скобки. Мы можем считать, что в реализации указанной выше функции `push` нет необходимости.

В абстрактном типе данных назначение интерфейса состоит в том, чтобы служить в качестве соглашения между программой-клиентом и реализацией. Обширная функция обеспечивает соответствие между языком в клиентской программе и определенными функциями в реализации. Но с другой стороны, интерфейс не содержит никакой информации о том, как должны быть реализованы функции, или хотя бы как они должны функционировать. Как мы можем объяснить клиентской программе, что такое стек? Для простых стеков, подобных стеку, можно было бы открыть код по сути, что и было бы такое решение не может являться эффективным. Членство программы-клиента приводит к описанию на объектном языке и помещению их в документацию на программу.

Строгая трактовка этой ситуации требует полного игнорирования того, как должны работать функции (с использованием формальной математической нотации). Такое описание никогда не приводит к спецификации. Разработка спецификации объекта является трудной задачей. Она должна описывать любую программу, реализующую функции на математическом языке, тогда как мы привыкли описывать работу функций с помощью кода, написанного на языке программирования. На практике работа функций предоставляется в виде описаний на английском языке. Но давайте пойдем дальше, пока мы не углубились слишком далеко в философические вопросы. В настоящей книге даются подробные примеры описаний на английском языке и несколько реализаций для большинства рассмотренных АТД.

Чтобы подчеркнуть, что наша спецификация абстрактного типа «стек» предоставляет достаточно информации для написания функциональной клиентской программы, перед рассмотрением реализации рассмотрим в разделе 4.2 две клиентские программы, использующие стеки наглядного типа.

Упражнения

4.9 В последовательности

`AB*Y*CB***BZ***YD**H***`

буква обозначает операцию сложения, а звездочка — операцию умножения. Определите последовательность значений, полученных операциями сложения.

4.10 Используя те же правила, что и в упражнении 4.9, поставьте звездочки и получите последовательность `ABBY` таким образом, чтобы последовательность значений, полученных операциями сложения, была следующей: (1) `ABBY`; (2) `YBAB`; (3) `ABYB`; (4) `AYBB`; или в каждом случае доказать, что такая последовательность не возможна.

- 4.11 Предположим, что язык для положительных чисел. Разработайте алгоритм, позволяющий определить, можно ли к первой последовательности добавить символов так, чтобы эта последовательность, будучи интерпретированной как последовательность стековых операций (аналогично упражнению 4.10), дала в результате истинную последовательность.

4.3 Примеры программ-клиентов, использующих АТД стека

В рассмотренных главах мы увидели огромное количество приложений со стеком. Однако, в качестве конкретного примера рассмотрим применение стека для вычисления арифметических выражений. Например, предположим, что требуется найти значение простого арифметического выражения с операциями умножения и сложения (см. табл. 4.40).

5 4 1 1 1 9 1 8 3 2 4 2 6 3 3 1 2 1

Это выражение в скобках содержит промежуточные результаты операций. Если сначала вычисляется $9 + 1 = 8$, придется сохранить результат 87 на время, пока вычислится $4 * 1 = 4$. Стек указанного типа представляет собой идеальный механизм для сохранения промежуточных результатов таких вычислений.

Начнем с рассмотрения более простой задачи, где выражение, которое необходимо вычислить, имеет другую форму — знак операции стоит после двух операндов, а не между ними. Как будет показано, в такой форме может быть представлено любое арифметическое выражение, причем формы postfix и infix являются, по сути, синонимами — обычной формы является арифметическая запись. Вот postfixное представление выражения из предыдущего абзаца:

5 9 8 + 4 6 * = 7 + *

Формы postfix, обратная postfixной, называется префиксной или польской записью (как и в ее французском аналоге до и после знака (à l'avance et à l'arrière)).

При infixной записи чтобы отделить, например, выражение

5 * (1 + (8 + 8) * (4 * 6)) + 7

от выражения

((5 * 8) + 8) * (1 + 4 * 6) + 7

требуется скобка, но при postfixной (или префиксной) записи скобки не нужны. Чтобы увидеть почему, можно рассмотреть следующий процесс преобразования postfixного выражения в infixное: все группы из двух операндов и следующих за ними знаков операций заменяются их infixными эквивалентами и заключаются в скобки для демонстрации того, что этот результат может рассматриваться как операнд. То есть, группы ab^* и $ab+$ заменяются группами $(a * b)$ и $(a + b)$, соответственно. Затем то же самое преобразование выполняется с полученным выражением, пока оно не будет обработано во всем своем объеме. В нашем примере преобразование будет происходить следующим образом:


```

5 9 8 + 4 6 * = 7 + *
5 ( 9 1 8 ) 1 4 * 6 ) = 7 1 *
5 [ ( 9 + 8 ) * 1 4 * 6 ) ] 7 + *
5 ( ( ( 9 + 8 1 * 1 4 * 6 ) ) + 7 ) *
1 5 * ( ( ( 9 + 8 1 * 1 4 * 6 ) ) + 7 ) ]

```

Таким образом, в постфиксном выражении можно определить все операции, связанные с любой операцией, поэтому необходимость в круглых скобках отпадает.

С другой стороны, с помощью стека можно действия можно выполнять эти операции и вычислить значение любого постфиксного выражения (см. рис. 4.2). Переменная слева направо, мы интерпретируем каждый операнд как команду "внести операнд в стек", а каждый знак операции - как команду "извлечь из стека два операнда, выполнить операцию и положить результат в стек". Программу 4.5 выписана реализацией этого процесса на языке C++. Обратите внимание, что, поскольку АТД Стек создан на основе шаблона, один и тот же код может использоваться и для создания стека целых чисел в этой программе, и стека символов в программе 4.6.

Программа 4.5 Вычисление постфиксного выражения

Эта программа-клиент для стека шаблонного типа считывает любое постфиксное выражение, включающее умножение и сложение целых чисел, затем вычисляет это выражение и распечатывает полученный результат. Предположим, что программа 4.4 имеет вид, показанный при этом предлагается, что интерфейс из программы 4.4 реализован в файле `STACK.cpp` как с. 80, созданный на основе шаблона.

Когда встречаются операции, они записываются в стек; когда встречаются знаки специй, из стека вынимаются два вычисленных значения, над ними выполняются нужные операции, и результат снова записывается в стек. Порядок, в котором две операции выполняются в выражении данного кода, в языке C++ не определен, поэтому существуют невыполнимые операции, такие как вычитание или деление. Был бы более сложный.

В то же время можно предположить, что целые числа и знаки специй могут быть какими-либо другими значениями (словами, пробелами), но программа не проверяет этого. Поэтому программа `if` и цикл `while` выполняют вычисление, подобно тому, что выполняет функция `atoi` языка C++, которая преобразует строки в код ASCII в целые числа, а также для вычисления. Когда встречается символ цифры, накопленный результат умножается на 10 и к нему прибавляется эта цифра.

```

#include <iostream.h>
#include <string.h>
#include "STACK.cpp"
int main(int argc, char *argv[])
{
    char *s = argv[1]; int N = strlen(s);
    STACK<int> myStack;
    for (int i = 0; i < N; i++)
    {

```

5	9			
9	5	9		
8	5	9	0	
+	5	17		
4	5	17	4	
*	5	17	4	6
=	5	17	58	
	6	328		
7	6	328	7	
+	6	316		
	Рис. 4.2			

РИСУНОК 4.2 ВЫЧИСЛЕНИЕ ПОСТФИКСНОГО ВЫРАЖЕНИЯ

Эта программа-клиент для стека шаблонного типа считывает любое постфиксное выражение, включающее умножение и сложение целых чисел, затем вычисляет это выражение и распечатывает полученный результат. Предположим, что программа 4.4 имеет вид, показанный при этом предлагается, что интерфейс из программы 4.4 реализован в файле `STACK.cpp` как с. 80, созданный на основе шаблона.


```

if (a[i] == 'x')
    swap.push(swap.pop() + swap.pop());
if (a[i] == 'y')
    swap.push(swap.pop() + swap.pop());
if ((a[i] == '0') && (a[i] == '0'))
    swap.push(0);
while ((a[i] == '0') && (a[i] == '0'))
    swap.push(10 * swap.pop() + (a[i+1] - '0'));
}
cout << swap.pop() << endl;
}

```

Постфиксная запись и стек обеспечивают естественный способ организации ряда вычислительных процедур. В некоторых языках интерпретация и язык программирования несут характерный либо базируются на постфиксной записи и стековых операциях — при выполнении каждой операции ее аргументы извлекаются из стека, а результат возвращается в стек.

Примером такого языка является язык ReversePolish. Это абстрактный язык программирования, в котором процедуры записаны в постфиксном виде и интерпретируются с помощью грубого стека, точно как и программа 4.3. Хотя мы не можем охватить здесь все аспекты этого языка (см. ранее ссылки), он достаточно прост и мы можем получить некоторые реальные примеры, чтобы увидеть возможность постфиксной записи и абстракции стека вычислительного типа. Например, строка

```
5 9 2 add 4 6 mul mul 7 mul mul
```

является программой на языке ReversePolish. Программа на языке ReversePolish состоит из операций (таких, как add и mul) и операндов (таких, как целые числа). Программу на этом языке мы интерпретируем так же, как делали это в программе 4.3 — читаем слева направо. Если встретится операция, мы занесем ее в стек, если встретится знак операции, из стека извлечем операнды для этой операции (если они есть), а затем результат (если он есть) занесем в стек. Таким образом, на рис. 4.2 полностью описан процесс выполнения этой программы. После выполнения программы в стеке будет число 2075.

На языке ReversePolish имеется несколько простых функций, которые служат инструментами для абстрактного графического языка; кроме того, можно определить и собственные функции. Для функций с аргументами, расположенными в стеке, вызываются так же, как и обычные функции. Например, следующий код на языке ReversePolish

```
5 9 push 144 211 0 72 push 72 111 110
```

соответствует последовательности действий: вызвать функцию push с аргументами 5 и 9, затем вызвать функцию 111 с аргументами 144 и т.д. Некоторые операции относятся непосредственно к самому стеку. Например, следующие две операции относятся к верхнему стеку: поэтому, например, код

```
144 0 110 60 2014 0 110
```


составляется с помощью последовательности действий: вызвать функцию `line(a)` с аргументами 144 и 0, затем вызвать функцию `line` с аргументом 60, затем вызвать функцию `line(0)` с аргументами 144 и 6 и т.д. В программе `PosScript`, показанной на рис. 4.3, определяется и используется функция `line`. Функция в языке `PosScript`, подобная макросам: последовательности `(/line { A } def)` делая новая `line` эквивалентным последовательности операций внутри фигурных скобок. Рисунок 4.3 представляет пример программы `PosScript`, в которой определена функция и вычерчивается простая диаграмма.

В данной концепции язык интерес к языку Pascal влияет тем, что этот широко не используемый язык программирования, базирующийся абстрактных стека на лексическом типе. Действительно, в стандартных средствах многих компьютеров реализованы основные стекосые операции, поскольку они являются естественным продолжением механизма вызова функций: при входе в процедуру текущее состояние программной среды сохраняется путем занесения информации в стек, при выходе из процедуры состояние программы среды восстанавливается за счет извлечения информации из стека. Как будет показано в главе 3, эта связь между магазинными стеками и процедурами, которые организованы в виде функций, особенно важна в модульной функции. Строится замкнутое модель вычислительного процесса.

Поспрандцясь к персоничальней талдче, стмемля, что, как талдч трыуе рнв. 4.4, калдкый стек можно ткеже использовать для преобралоння нфнкрного прифнметического выражения в калдкые скобки в лосржае чуче Формы. При выхлоске этого преобралоння: лухи охлелнн лхлется в стек, а самы операнды прсто перелалены в выхлоск лухл пркормлены. Прхлн охлелн показлует, что для послелднх члел на пухлел лухлрнмы лвлнелся дрлжонення операндл, знак лстарой талдчел в стек послелднм. Пухлнну элел дрлж о-тернел: охлелн пухлел-нл стекл н талдчел лелел в пухлелной пухлел прлграммы.

Питання 4: представляє собою реакцію типу прощання. Зверніть увагу, що зрештою



1111

```

dup 0 rlineto
00 rotate
dup 0 rlineto
-120 rotate
dup 0 rlineto
60 rotate
dup 0 rlineto
60 rotate
pop
- def
0 0 moveto
144 x()
0 72 moveto
72 fill
stroke

```

РИСУНОК 4.3 ПРОСТАЯ ПРОГРАММА НА ЯЗЫКЕ POSTSCRIPT

[illegible]

ты в постфиксной выражении расположены в том же самом порядке, что и в инфиксном выражении. Поэтому также следует отметить, что левые скобки в инфиксных выражениях не нужны. Однако они необходимы, если бы отсутствовали операции, имеющие разное количество операндов (см. упражнение 4.14).

Понимая теперь, что алгоритм, разработанный в данном разделе для вычисления инфиксных выражений, представляет собой пример использования абстрактной структуры стека, мы не сможем избежать упражнения по абстракции. Во первых, отдельные данные преобразуются в промежуточные представления (постфиксные выражения). Во-вторых, для интерпретации и завершения этого выражения выполняется работа абстрактной машины, функционирующей на основе стека. В этом абстрактном и реальном этой же схеме работы участвуют многие современные компьютеры; задача трансляции программы на C++ для какого-либо компьютера разбивается на две задачи, которые концентрируются вокруг промежуточного представления. Поэтому данный пример может использоваться по задаче исполнения этой программы, точно так же как это делается для данного раздела. В разделе 4.2 будет показано разделение, но другое промежуточное представление.

Программа 4.6 Преобразование из инфиксной формы в постфиксную

```

Эта программа является еще одним примером: программа
исполняет для строки математического выражения. В данной строке
содержат символы, для преобразования (A+B) в постфикс
формы: AB+ инфикс скобки интерпретируются как (A+B) и
записываются A в стек и записывается знак + в выходной поток
записывается B в стек и записывается правая скобка в стек
вызывается знак * и он же записывается в выходной поток

#include <iostream.h>
#include <string.h>
#include "STACK.h"
int main(int argc, char *argv[])
{
    char *s = argv[1]; int N = strlen(s);
    STACK<char> s1(N);
    for (int i = 0; i < N; i++)
    {
        if (s[i] == '(')
            cout << s[i] << " ";
        if ((s[i] == '+') || (s[i] == '*'))
            s1.push(s[i]);
        if ((s[i] == ')') || (s[i] == '='))
            cout << s[i] << " ";
    }
    cout << endl;
}

```

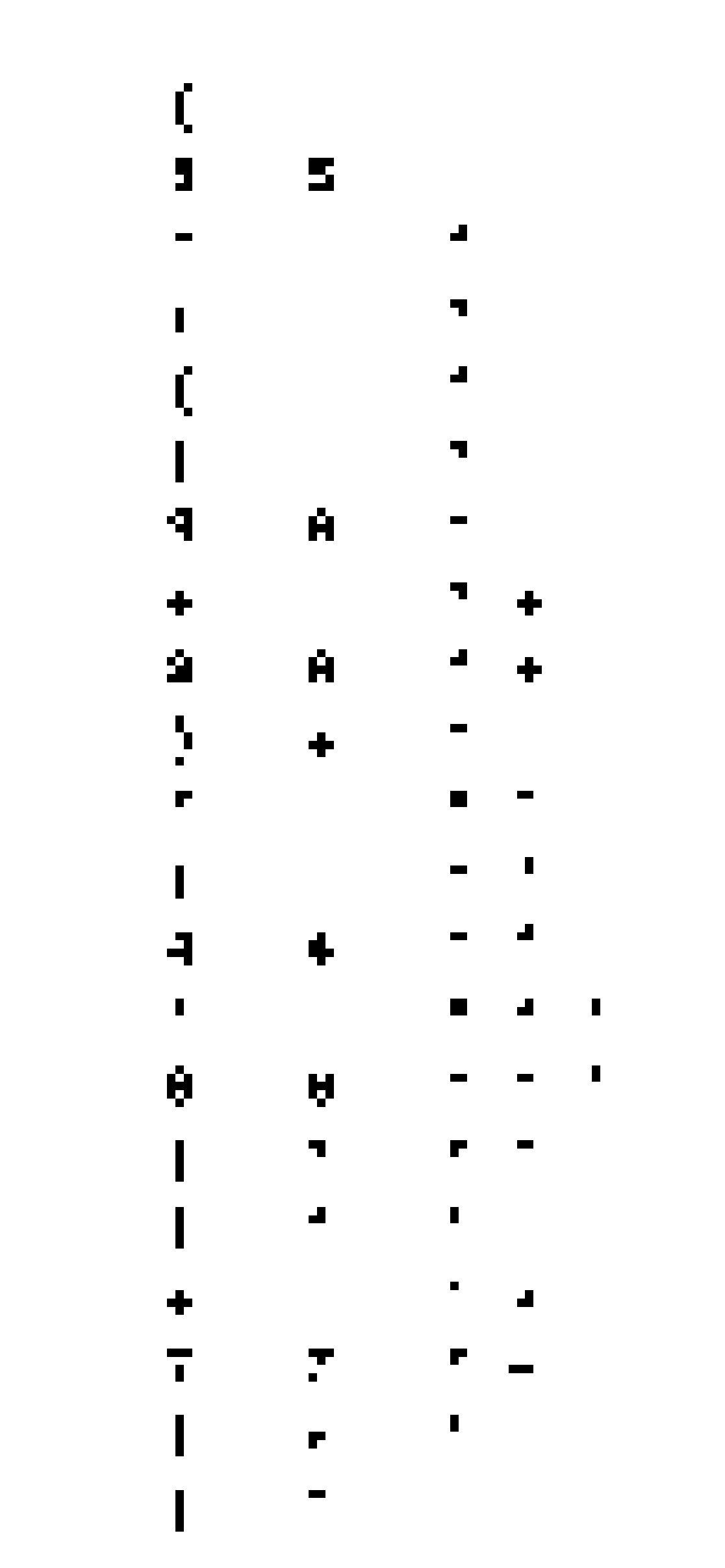


РИСУНОК 4.4
ПРЕОБРАЗОВАНИЕ
ИНФИКСНОГО
ВЫРАЖЕНИЯ В
ПОСТФИКСНОЕ

Данные
операнды помещаются
в стек и интерпретируются
как (A+B) и записываются
A в стек и записывается
знак + в выходной поток
записывается B в стек
и записывается правая
скобка в стек
вызывается знак * и он же
записывается в выходной
поток
записывается 5 в стек
и записывается 9 в стек
и записывается 10 в стек
и записывается + в стек
и записывается * в стек
и записывается 4 в стек
и записывается 12 в стек
и записывается 7 в стек
и записывается + в стек
и записывается * в стек
и записывается + в стек
и записывается .

Это правило не только иллюстрирует достаточность абстрактных типов данных и свойств $\mathcal{C} \vdash \vdash$. Здесь используются два разных стека, один из которых содержит объекты типа `char` (знаки операций), а другой — объекты типа `int` (значения). С помощью АТД в виде шаблона класса, описанного в примере 4.9, можно даже обобщить все рассмотренных клиентских программы на одну (см. упр. 4.19). Несмотря на простоту реализации решения, стоит отметить, что это, по-видимому, не самый лучший пример, поскольку различные реализации могут отличаться своей полнотой реализации, возможно, не очевидной бы автору решения, что в обоих случаях будет применены одна и та же реализации. Действительно, наличие различных уровней реализации для реализации, и сейчас мы приступим к рассмотрению этих тем применительно к стеку абстрактного типа.

Упражнения

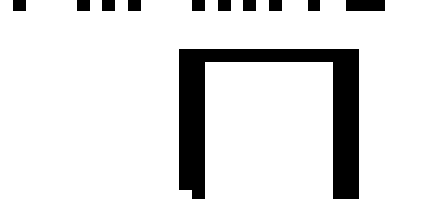
- 4.12 Преобразуйте в постфиксное выражение

$$1 + 2 * (3 * 4 * 5) + 6 * 7 + 8 * 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16 + 17 + 18 + 19 + 20$$
- 4.13 Каким же образом, как на рис. 4.7, показана содержимое стека при вычислении программы 4.5 следующего выражения

$$3 * 4 + 5 * 6 + 7 * 8 + 9 * 10 + 11 * 12 + 13 * 14 + 15 * 16 + 17 * 18 + 19 * 20$$
- 4.14 Распишите программу 4.5 и 4.6 таким образом, чтобы они выполняли операции $+$ (целочислая) и $/$ (целочисленная).

4.15 Распишите решение упражнения 4.14 таким образом, чтобы оно включало универсальные операции $+$ (целочисленная) и $/$ (целочисленная). Кроме того, реализуйте механизм абстрактного стека в программе 4.5 так, чтобы можно было использовать стек с произвольной глубиной. Например, если в качестве исходного выражения

$$(3 * 4) + ((1 + 2) * (3 * 4)) + ((1 + 2) * (3 * 4)) + ((1 + 2) * (3 * 4)) + ((1 + 2) * (3 * 4))$$
 программа должна выдать число 1.61503%.

4.16 Напишите на языке Pascal программу, которая чертит следующую фигуру: 
- 4.17 Методом индукции доказать, что программа 4.5 правильно вычисляет любое постфиксное выражение.
- 4.18 Напишите программу, которая с использованием стека реализует тип преобразования постфиксного выражения в инфиксное.
- 4.19 Объясните программы 4.5 и 4.6 в словесной форме, в которой будут использоваться два разных АТД: стек значений и стек операций (знаков операций).
- 4.20 Разработайте компилятор и интерпретатор для языка программирования, в котором каждая программа состоит из одного арифметического выражения. Выражения состоят из ряда операторов, присваивания с арифметическими выражениями, состоящими из чисел и переменных, обозначенных буквами латинского алфавита. Например, для языка абстрактных данных

$$\begin{aligned}
 & x = 1; \\
 & y = x + 1; \\
 & (y * 2) * 3 + (4 * x)
 \end{aligned}$$
 программа должна вывести число 19.

4.4 Реализации АТД стека

В данном разделе рассматриваются две реализации АТД стека: на основе массива и на основе списка. Эти реализации отличаются в основном способом применения базовых инструментальных средств, рассмотренных в главе 3. Мы полагаем, что они достаточно хорошо своей производительностью (быстродействие).

Программа 4.7 Реализации стека магазинного типа на базе массива

В этой программе для N элементов $1 \dots N$ хранятся значения $a[0] \dots a[N-1]$, начиная с первого занесенного элемента и завершая последним. Верхушкой стека (последней и единственной занесенной парой для элементов стека) является пара $a[N]$. Максимальное количество элементов, которое может вместить стек, программа может принимать как аргумент и константу `STACK`. Размер `STACK` в памяти и массив для его размера, однако код не проверяет такие ошибки, как помещение элементов в переполненный стек (или выталкивание элемента из пустого стека).

```
template <class T> Stack
class STACK
{
private:
    T* a; int N;
public:
    STACK(int maxN)
        ( a = new T[maxN], N = 0; )
    int empty() const
        { return N == 0; }
    void push(const T& item)
        { a[N++] = item; }
    T& pop()
        { return a[--N]; }
};
```

Если для рассмотренного стека применяются массивы, то все функции, объявленные в программе 4.7, реализуются очень просто, что видно из программы 4.7. Описывая операции с массивом в точности так, как показано на рис. 4.1, при этом отслеживается наличие вершины стека. Выталкивание операции выполняется путем записывания элемента в следующий массив, уменьшением индекса вершины стека, а затем увеличение этого индекса на единицу; выполнение операции добавления означает увеличение индекса на единицу и запись нового элемента, соответствующего этому индексу. Операции удаления (конструктор) осуществляется размещением массива указанного размера, а операция разрушения, если он есть, пропадет, не равен ли индекс нуля. Если рассмотренные вместе с классической программой (стек, как программа 4.5 или 4.6), эта реализация обеспечивает разнотипный и эффективный стек машинного типа.

Один потенциальный недостаток применения массивов для реализации стека известен многим: как это обычно бывает со структурами данных, создаваемых на базе массивов, то использование массивов не позволяет иметь это максимальный размер, чтобы распределить над ним оперативную память. В рассмотриваемой реализации эта информация передается в аргументе конструктора. Другой недостаток — результат выбора реализации на базе массива, он не является неотъемлемой частью АТД.

стка. Зачастую требуется определить максимальное число элементов, которое программа будет заносить в стек: если выбрать слишком большое число, то такая реализация будет неэффективной; использовать ограниченную память. Я это может быть необходимо в тех приложениях, где память является ценным ресурсом. Если выбрать слишком маленькое число, программа может вообще не работать. Преимущество АБД дает возможность рассуждать другим образом: не изменять реализацию без изменения логики функционирования программы.

Например, чтобы стек мог эффективно увеличиваться и уменьшаться, можно было бы иметь в распоряжении связанную структуру, как в программе 4.8. Стек организован в обратном порядке по сравнению с реализацией на базе массива — начиная с последнего занесенного элемента и увеличиваясь первым. Этот (см. рис. 4.3) позволяет более просто реализовать базовые стекковые операции. Чтобы добавить элемент, удаляется указ в начале списка и добавляется к нему элемент; чтобы удалить элемент, создается новый указ к объекту в начале списка. Поскольку все операции списочной структуры выполняются в $O(1)$ -ом времени, соответствующий переключатель стека, не требуется.

Программа 4.8 не проверяет такие ошибки, как попытка извлечения элемента из пустого стека, вставки элемента в переполненный стек или выход за пределы памяти. Но эти ошибки проверки двух последних условий можно избежать. На самом деле можно трактовать как незначительные ошибки в программе количество элементов в списке, для чего при каждой вставке в стек проверять, что список не превышает значения, переданное конструктору в качестве параметра, и что не происходит успеха. Значение можно задать произвольно, при которой не требуется заранее знать максимальной размер стека. В доработанной программе после каждого упражнения 4.24, сообщить о том, что стек переполнен, либо о том, что стек не заполняется с ошибкой.

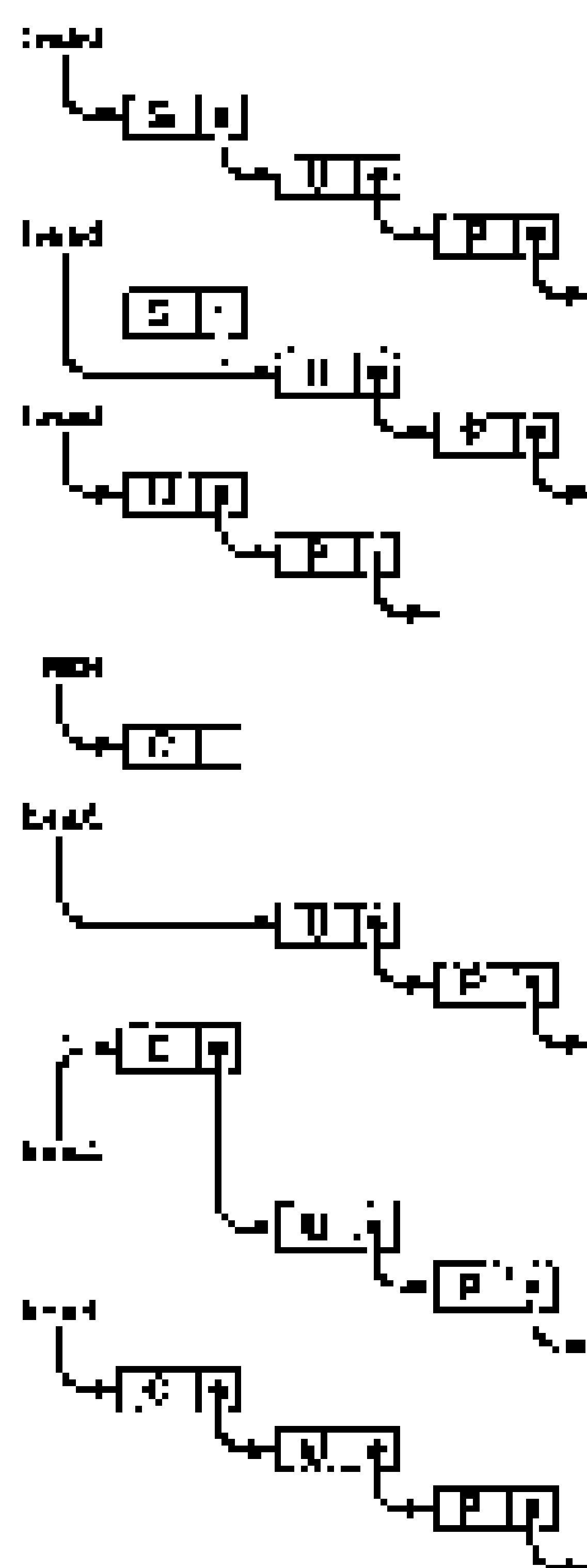


**РИСУНОК 4.5 СТЕК НЕАГАЗИРОВОГО
ТМПА НА БАЗЕ СВЯЗНОГО СПИСКА**

Глобалната промяна в климата ще
 стане, когато всяка държава не
 спазва правилата за опазване на
 околната среда. За да се избегне това,
 трябва да се вземат мерки (БФП). Например
 намаляване на изпускането на
 парникови газове, защита на
 горите, използване на енергия
 от възобновяеми източници и др.
 Това е обща задача, която трябва да
 бъде решена заедно. В противен
 случай, ако не вземем мерки, в
 бъдеще може да настъпи катастрофа.
 Затова е важно да се работи за
 намаляване на изпускането на
 парникови газове и опазване на
 околната среда.

Программа 4.8 Реализация стека магазинного типа на базе связанного списка

В этой программе АДЛ реализуется с помощью значащего члена. Предустановленный для этого значащего списка алгоритм реализован в том же способе (см. главу 3) в качестве конструктора для узлов. Он создает и соединяет каждый новый узел данным элементом в его стек.

```
template <class Item>
class STACK
{
private:
    struct node
    { Item item; node* next;
      node(Item x, node* b)
      { item = x; next = b; }
    };
    typedef node* link;
    link head;
public:
    STACK(int)
    { head = 0; }
    int empty() const
    { return head == 0; }
    void push(Item x)
    { head = new node(x, head); }
    Item pop()
    { Item x = head->item; link t = head->next;
      delete head; head = t; return x; }
};
```

Программы 4.7 и 4.8 представляют две различные реализации одно и того же АДЛ. Можно поменять одну реализацию другой, не делая никаких изменений в клиентских программах. Подобных тех, которые рассматривались в разделе 4.3. Они отличаются только принципами работы. Реализации на базе массива используют объем памяти, необходимый для размещения максимального числа элементов, которое может возникнуть стек в процессе выполнения; реализации на базе списка используют объем памяти пропорциональный количеству элементов, но при этом всегда расходует память на создание памяти для каждой ссылки на каждый элемент, а также дополнительное время на распределение памяти при каждой ссылке и освобождение памяти при каждой операции удаления. Если требуется стек больших размеров, который обычно является практически неизбежным, предпочтительнее будет отдать реализации на базе массива. Если же размер стека варьируется в широких пределах и присутствуют другие структуры данных, которые требуется хранить, не используя их во время, когда в стек находится несколько элементов, предпочтительнее следует отдать реализации на базе связанного списка.

На ходу замечание: на рисунке будет понятно, что эти же самые соображения относительно использования оперативной памяти справедливы для многих реализаций АДЛ. Работотехники часто оказываются в ситуациях, когда приходится выбирать между возможностью быстрой работы к любому элементу при необходимости быстрого уничтожить максимальное число требуемых элементов (и реализация на базе массивов) и гибким использованием памяти (пропорциональное количеству элементов, входящих

за и стек) при обеспечении быстрого доступа к любому элементу (в реализации на базе связанных списков).

Нашим аналитическим изображением отвлеченного исполнения является, конечно, по-прежнему всего интереснее различия в производительности разных реализаций АД. Однако не во всех случаях выполнения. В данном случае различия между двумя рассмотренными реализациями незначительны.

Лемма 4.1 *Можно ли для любого линейного списка, для АД отвлеченного типа можно реализовать операции откладывать и вытаскивать, используя для этого один элемент.*

Этот факт является непосредственным следствием инвариантного включения программ 4.7 и 4.8.

То, что в реализациях на базе массивов и списков список элементов стека хранится в разном порядке, для конечных программ не имеет никакого значения. В реализации могут использоваться какие угодно структуры данных, лишь бы они обеспечивали конечному абстрактному типу абстрактные типы. В общем случае результаты работы и зависеть от неопределенных эффектов абстрактного объекта, который может выполнять произвольные операции с помощью всего лишь бесконечного множества инструкций. Если вся лемма — отыскать структуры данных и эффективные реализации для группы базисных АД.

Реализация на базе связанного списка создает видимость стека, который может увеличиваться неограниченно. Он так же в практических условиях такой стек нехорошо: рано или поздно, когда список не вмещает в себя некоторую область памяти не может быть удовлетворен, как сгенерировать код. Можно также организовать стек на базе массива, который будет увеличиваться и уменьшаться динамически: когда стек достигнет определенного размера массив увеличится в два раза, а когда стек становится пустым, размер массива уменьшается в два раза. Однако реализация упомянутой стратегии не отличается по качеству выполнения от того, что этот процесс подробно рассмотрен для более сложных программ.

Упражнения

- 4.21 Определите содержимое элементов $s[0], \dots, s[4]$ после выполнения с помощью программы 4.7 операций, показанных на рис. 4.1
- 4.22 Препрограммируйте интерфейс стека на языке программирования, изменяя интерфейс программы, указав для стека операции *push* и *pop*, которые возвращают конкретные значения. Разложите их в линейной памяти в структуре данных. Разработайте операции *push* и *pop* для представления на базе массива (программа 4.7) и представьте их на базе связанного списка (программа 4.8)
- 4.23 Измените реализацию стека на языке программирования на базе массива (программа 4.7) так, чтобы она выполняла функцию *clear*(), если элемент *pop* возвращает значение, которое указывает на выполнение операции *push* и *pop*, когда стек пуст. Для проверки работы, создайте список из элементов.
- 4.24 Измените реализацию стека на языке программирования на базе связанного списка (программа 4.8) так, чтобы она выполняла функцию *clear*(), если элемент *pop*

стаканя программа пытается выполнить операцию вычисления, когда стек пуст, или операция извлечения, когда отсутствует элемент памяти при попытке pop.

4.24 Напишите реализацию стека магического типа на базе связанного списка (программа 4.6) так, чтобы при сохранении информации использовался индекс (index) (рис. 3.4).

4.26 Напишите реализацию стека магического типа на базе связанного списка, в которой элементы списка хранятся в ячейках с заданным значением элемента в завершающей последовательности элементов. Потребуется реализовать двухсвязный список.

- 4.27 Разработайте АТД, который содержит два разных стека магического типа. Воспользуйтесь репликацией на базе массивов. Один стек должен быть в начале массива, другой — в конце. (Если клиентская программа работает так, что один стек уничтожается, а второй как другой используется, эта реализация будет занимать меньше объем памяти, нежели альтернативные варианты.)
- 4.28 Реализуйте функции вычисления магических выражений, состоящих из двух чисел. Она должна выполнять программы 4.3 и 4.6, а также использовать АТД из упражнения 4.26. Дополнительно потребуется рассмотреть случаи, когда оба стека содержат элемент одного и того же типа.

4.5 Создание нового АТД

Разделы с 4.2 по 4.4 содержат пример типичного подхода программиста С++ для работы на наиболее важных абстракциях базового связанного типа. В главе 4.5 на разделах 4.2 определены основные операции; абстрактные программы из раздела 4.3 могут исполняться для операций независимо от их реализации, а программы из раздела 4.4 обеспечивают конкретное представление в программный код АТД.

Создание нового АТД часто сводится к следующему процессу. Начать с разработки клиентской программы, предназначенной для решения конкретной задачи, сформулировать операции, которые являются наиболее важными как операции, которые должны выполнять над типом. Затем определяется интерфейс и пишется код программы-клиента с целью проверки того, что необходимый АТД успешно реализован клиентской программой. Затем выполняется анализ, можно ли достать более эффективную реализацию операций в рамках АТД. Если нет, придется пойти на шаг назад и переопределить интерфейс, поскольку в нем описаны, более подходящие для эффективной реализации. Поскольку модификация шлет к клиентской программе, ее требуется соответствующим образом изменить. Вполне возможно модифицировать интерфейс и клиентской программы, получая работающую клиентскую программу и работающую реализацию; интерфейс "выпускается" в таком состоянии, т.е. он больше не изменяется. С этого момента разработчик клиентской программы и реализаций выполняемых операций может искать другие клиентские программы, использующие тот же самый АТД (например, для целей дополнения языка программирования), можно выполнять другие реализации, равно как и сравнивать производительность различных реализаций.

В других ситуациях можно начать с определения АТД. Например, человек может поработать с некоторыми вопросами: какие базовые операции над доступными данными

можно потребовать? «Синтаксис программирования» Каким аппаратом можно реализовать эффективно? Завершить разработку реализации, можно проверить ее эффективность с помощью численного решения. Перед окончательным «завершением» интерфейса, возможно, потребуется дополнительная модификация и тесты.

В главе I подробно рассмотрен один пример, где размышления о том, какой уровень абстракции использовать, привели к выбору эффективной альтернативы решению сложной задачи. Теперь посмотрим как обсуждаемый в настоящей главе абстрактный подход применяется для нахождения абстрактных контрпримеров к главе I.

В программе 4.0 определен интерфейс, позволяющий для операции (в дополнение к операции *создать объект*). Показано, что эти операции являются абстрактными, рассмотренные в главе 1, на высоком уровне абстракции. Каким бы образом эти операции в структуре данных не реализовывались, необходимо иметь возможность привязки связей двух узлов, а также указать, что конкретные два узла являются связанными.

Программа 4.9 Интерфейс АД "Отношения живых существ"

Интерфейс АТД, по которому с АСБМ была проведена интеграция с АС, - состоит из статистической группы; расчетные расстояния от объектов связи до центра класса, поддерживающего эту группу; данные о группе: количество объектов (объектов) и директоров группы; для каждого объекта данные о связи: название - регион (услуга, услуга АТД), количество объектов, связанных с объектом (объект), для каждого объекта и для каждого объекта информации.

class 30

```
{
private:
    // программный код, недоступный извне
public:
    void Set(int);
    int Get(int, int);
    void Print(int, int);
}
```

Программа 4.19 — это как бы такая программа, которая для решения задачи имеет некоторую структуру АТД с интерфейсом, представленным в программе 4.9. (Одна из особенностей этого АТД состоит в следующем: программа понималась программистом, поскольку она является с использованием абстракции, но она не является естественно преобразована в программу).

Программа 4.11 является реализацией интерфейса `vbtest-1a.1`, который описан в программе 4.6. В этой реализации (см. рисун. 1.3) применяется бёр деренг, в основе которого лежит идея описанная, предоставляющие известную информацию о функциях. В других алгоритмах, рассмотренных в главе 1, используются различные реализации $A[1]$, причем их можно тестировать независимо друг от друга, не изменяя клиентскую программу.

Этот АИД приобщит к созданию несколько менее эффективных программ, нежели программы, связанные с АИД (табл. 1), поскольку он не обременен тем преимуществом упомянутой программы, что с ней каждой итерации создаваемых (новых) мероприятий можно предложить следующие шаги (рис. 4). Дополнительная нагрузка подобного рода является главной из причин к более эффективному представлению. В данной статье

существует множество способов устранения этого недостатка, например, за счет более сложной реализации функций (см. выражения 4.9). На практике пути были бы очень короткими (особенно, если применяется сжатие путей), так что в данном случае достаточно только подсказка, по-видимому, окажется полезной.

Программа 4.10 Клиентская программа для АТД "Отношения эквивалентности"

АТД из программы 4.9 позволяет отразить алгоритм эквивалентности эквивалентности, тем самым делая его более доступным.

```
#include <stdio.h>
#include <stdlib.h>
#include "TP.h"

int main(int argc, char *argv[])
{
    int p, q, E = atoi(argv[1]);
    if (info[0])
        while (info >= p >= q)
            if (!info.find(p, q))
            {
                info.put(p, q);
                printf("%d %d\n", p, q);
            }
    }
}
```

Комбинировав программы 4.10 и 4.11 с точки зрения функциональной эквивалентности программы 4.3; однако, реализация программы на два чаша является более эффективным подходом, так как:

- позволяет сделать решение задачи быстрого времени (задача сложности $O(n^2)$ и решение за два шага не эквивалентно)
- предоставляет естественный способ хранения различных алгоритмов и структур данных, применяемых при решении этой задачи
- предоставляет возможность использовать способ проверки выполняемой работы программиста обеспечения
- обеспечивает механизм, позволяющий совершенствовать представления (переходить к новым структурам данных в том же алгоритме); без каких-либо изменений кода клиентских программ
- предоставляет абстракцию, которую можно использовать при построении других алгоритмов

Сказанное выше справедливо для любых задач, с которыми приходится сталкиваться при разработке клиентских программ, так что эти базовые принципы построения абстрактных типов данных применимы достаточно широко.

Программа 4.11 Реализация АТД "Отношения эквивалентности"

Эта реализация быстрого объединения из главы 1 является реализацией функций из программы 4.3. В данной реализации код функций `find` и `put` для его использования в других приложениях. Приведенная реализация использует `std::vector` для хранения данных, связанных с каждым элементом.


```

private:
    int *id, *ex;
    int find(int x)
    { while (x != id[x]) x = id[x]; return x; }
public:
    UF(int N)
    {
        id = new int[N]; ex = new int[N];
        for (int i = 0; i < N; i++)
            { id[i] = i; ex[i] = 0; }
    }
    int find(int p, int q)
    { return (find(p) == find(q)); }
    void unite(int p, int q)
    { int i = find(p), j = find(q);
      if (i == j) return;
      if (ex[i] < ex[j])
          { ex[i] = j; ex[j] += ex[i]; }
      else { id[j] = i; ex[i] += ex[j]; }
    }
};

```

В коде программы 4.11 снесены комментарии и раздвоения, поэтому он не допускает отдельную компиляцию клиентских программ и раздвоений. Для того чтобы программа раздвоения могла работать-а) с одним и тем же интерфейсом, б) работать нежно, как в разделе 1.1, разделим на три файла исходным образом.

Создать заголовочный файл с именем, скажем, `UF.h`, который будет содержать объявления классов, предоставлять джен-ы и объявлять функции, но не определять функции. В рассматриваемом примере этот файл будет содержать код программы 4.9, в который также включается предопределение дженных статических объявлений из программы 4.11. Затем необходимо сохранить определения функций в отдельном файле `uf.c`, который будет также содержать директиву `include` для файла `UF.h` (как это имеет место в любой канонической программе). При таком разделении возникла возможность раздельной компиляции клиентских программ и раздвоений. В самом деле, определение любой функции-члена класса можно сохранить в отдельном файле, если только функция-член объявлена в коде с, а в определении функции класса ее именем поместится имя класса и имя с. (Например, определение функции `find` в нашем примере следовало бы записать так:

```

int UF::find(int p, int q)
{ return (find(p) == find(q)); }

```

К сожалению, при раздельной компиляции разные компиляторы накладывают различные требования на раздвоение шаблонов. Проблема заключается в том, что компилятор не может создать код для функций-членов без знания типа параметра шаблона, который используется, так как он определен в главной программе. Определенная функция-членом внутри объявленного класса подменяет значение подобного рода шаблонов, связанных с компиляцией.

Однако даже без помощи прокси-файлов, по-прежнему, не хватает идеального, по крайней мере, представления данных, которые в действительности являются частью реализации. хранятся в одном файле в формате, с помощью которого этого представления можно предотвратить доступ к нему со стороны клиентских программ, но если требуется в реализации изменения, которые, в свою очередь, потребуют изменений в представлении данных, придется изменить файл и перекомпилировать все клиентские программы. Во многих случаях эта разработка программного обеспечения может отсутствовать из-за шифрования программ-клиентов, так что это может оказаться весьма затруднительным. В некоторых случаях такая организация может иметь смысл. В случае очень большого и сложного АИД можно считать оправданным на протяжении долгих и нудных, а уже потом увидеть команду программистов за разработку различных частей реализации. В этом случае общедоступная часть интерфейса послужит связующим между программистами и клиентами, а приватная часть — связующим между программистами. К тому же, когда необходимо найти самый лучший способ решения некоторой проблемы при условии, что данные не подпадают под некую конкретную структуру данных, эта стратегия зачастую обеспечивает то, что и требуется. Таким образом, можно повысить производительность, изменяя лишь незначительную часть общей системы.

В языке C++ не хватает механизмов, предназначенных специально для того, чтобы предоставить возможность писать программы с хорошо определенным интерфейсом, позволяющим полностью отделять клиентские программы от реализации. Этот механизм базируется на так называемых *абстрактных классах*, посредством которых можно декларировать или переклассифицировать некоторые члены существующего класса. Включаясь к ключевому элементу *virtual* и объявление функции члена класса, что эта функция должна быть переопределена в производном классе; декларация предоставляет способ именовать эти функции члена указывая на то, что данная функция является частью виртуальной (или *virtual*) функции, которая должна быть переопределена в любом производном классе. Производные классы обеспечивают удобный способ задания программ на основе работ других программистов и являются важным компонентом объектно-ориентированной программной системы.

Абстрактный (abstract) класс — это класс, все члены которого являются членами виртуальных функций. В таком классе, наследуемом от абстрактного класса, должны быть определены все функции-члены и любые необходимые приватные данные-члены; таким образом, по нашей терминологии абстрактный класс является интерфейсом, а любой класс, производный от него, является реализацией. Программы-клиенты могут использовать интерфейс, а система C++ может обеспечить соблюдение соглашения между реализацией и реализацией, даже когда данные и реализации функций компилируются раздельно. Например, в программе 4.12 мы видим абстрактный класс `if` для отключенной совместимости; если изменить кодную базу программы 4.11 следующим образом:

```
class IF : public class ut
```

то она будет указывать на то, что класс `IF` является производным от класса `ut` и поэтому в нем определены (как членами) все функции-члены класса `ut` — т.е. класс `IF` является реализацией интерфейса `ut`.

Программа 4.12 Абстрактный класс для АТД отношения эквивалентности

Приведенный ниже код формирует интерфейс АТД отношений эквивалентности, который отличается от того, заданного классом `AbstractSet` и реализацией (см. ниже).

```
class AbstractEquivalence
{
    public:
        virtual bool isReflexive() = 0;
        virtual bool isSymmetric(int, int) = 0;
        virtual bool isTransitive(int, int) = 0;
};
```

К сожалению, реализация абстрактных классов не может быть реализована дополнительным классом, так как при выполнении программы, вызывающей каждый из методов функции требует обращения к таблице указателей на функции-члены. Вспомогательные функции требуют наличия информации о том, как в отношении определенных классов для абстрактных классов. Так как реализации членов и методы алгоритмов структуры данных часто находятся в частях системы, контрпримеру, приведенному ниже, возможно, понадобится одна из таблиц, предоставляемую абстрактными функциями, которые для них не платит.

Есть еще один способ действий — написать структуру, не являющуюся файлом, для которой приватные члены хранятся не в интерфейсе, не в отдельном файле. Например, в классе из программы 4.9 можно было бы добавить в начале строки

```
private:
    #include "Abstract.h"
```

и в файл `Abstract.h`. Эта структура позволяет аккуратно выложить четыре компонента (клиентская программа, реализация, представление данных и интерфейс) и обеспечить согласованность реализации и структуры данных и алгоритмов.

```
#include "Abstract.h"
using namespace std;
```

в файл `Abstract.h`. Эта структура позволяет аккуратно выложить четыре компонента (клиентская программа, реализация, представление данных и интерфейс) и обеспечить согласованность реализации и структуры данных и алгоритмов.

Гибкость, которую можно достичь с помощью абстрактных классов и структур "файлов", компенсируется некоторыми недостатками. Так же, как и нарушение соглашения между программистами-клиентами и реализацией и абстракцией того, каким должен быть АТД. Все эти недостатки подчеркивают, что клиентские программы и реализации будут корректно взаимодействовать, только если также имеют друг от друга и функциональную такую абстракцию, что в общем случае весьма сложно обеспечить. Например, предположим, что класс имеет последовательный (или параллельный) поиск, реализованный нашим алгоритмом внешнего быстрого поиска (трудным для понимания в рамках возможности алгоритмов быстрого абстрагирования (или еще хуже, реализации), который даже не дает прямой реализации решения). Мы хотим строгим образом, чтобы такая классическая операция была легкой, но в данном случае это может совершенно испортить программу-клиент в пакете приложений, которая предназначена на эту операцию, обладая хорошей производительностью для других за-

лич. Практика программирования должна поощрять привычку, а не быть очень трудной и запутанной.

Описанная здесь расхоженная установка к объектам и классам привнесена в язык, компилятор, язык разметки, среду выполнения программы, что весьма доказательно завершено. Поэтому, если вы все будете придерживаться принципа, общепринятого по разделению программы на два файла, где АИД реализуется в виде класса `Set`, абстрактные функции-члены составят интерфейс, а реализацию обеспечат с интерфейсом в отдельном файле, который реализуется в программ-клиенте и компиляторе каждый раз, когда компилируются клиентские программы. Первопричиной связана с тем, что реализация в виде класса — это удобное и компактное средство представления структур данных и операций. Если для какого-либо подкласса (в понимании потребности пользователя) сложность, которая может быть обеспечена одним из тех, что упомянутых способов, может соответствующим образом изменить структуру языка.

Упражнения

- 4.29 Модифицируйте программу 4.11 так, чтобы в ней не пользовались сжатие пути деления `path` и `name`.
- 4.30 Устраните помехи и неэффективность программы, добавив в программу 4.9 операции, которая объединит операции `add` и `find` в соответствующим образом изменяя программу 4.11 и 4.16.
- 4.31 Модифицируйте интерфейс (программа 4.9) в реализацию (программа 4.11) для обеспечения эффективности так, чтобы в них присутствовали функции, выполняющие количество операций, связанных с данными.
- 4.32 Модифицируйте программу 4.11 так, чтобы для представления структуры данных в ней вместо параллельных массивов использовались массивы с `push` и `pop`.
- 4.33 Используя язык языка `C++` (или `Java`), напишите программу из трех файлов, реализующую значение простого выражения. Обеспечьте, чтобы эти операции программы (аналог программы 4.5) можно было комбинировать свободно от реализации стека (аналог программы 4.31).
- 4.34 Модифицируйте решение предыдущего упражнения — разделите представление данных и реализацию функций — разбейте программу на четыре файла. Проверьте полученный результат, подставляя различные стеки на один связного языка (аналог программы 4.5) без перемещения программы-клиента.
- 4.35 Создайте новую реализацию АИД "Отношения эквивалентности" из одной структуры класса с виртуальными функциями и сравните эффективность реализации программы 4.11 по группам заданных элементов (в стиле таблицы 1.1).

4.6 Очереди FIFO и обобщенные очереди

Очередь в абстрактной АТД (FIFO-Queue, 19.1-20.2). Первый элемент, чье время является наименьшим, является элементом очереди. Из очереди удаляются все элементы, чье время является наименьшим, и добавляется элемент, чье время было наименьшим в очереди раньше всех остальных.

Пожалуй, самым большим козлом-опрокинутым ящиком из моего профессора должен был быть функционеризатор очереди FIFO, поскольку дисциплина "первый пришел, первый ушел" настолько кажется естественной для обработки входящих корреспонденций. Однако профессор не всегда вовремя обращал на технические тонкости и даже позволял себе спадывать на лекции! В итоге какая-нибудь студентка-записочница могла зайти на лекцию, но другие студенты обрабатывались сразу же после лекции. В очереди FIFO документы обрабатываются по порядку, и каждый должен ожидать своей очереди.

Программа 4.13 Интерфейс АТД "Очередь FIFO"

Этот интерфейс идентичен интерфейсу стека из главы 4.4, за исключением μιας функции. Эти две АТД отличаются только тем, доказывая, что совершенно не строится на одних интерфейсах.

```
template <class T>
class Queue
{
public:
    // абстрактный код, записанный из файла queue
    virtual:
        Queue();
        int empty();
        void put(T&);
        T& get();
};
```

Очереди FIFO с типичной частотой встречаются в повседневной жизни. Когда вы стоите в очереди людей, чтобы посмотреть какой-нибудь фильм или купить продукты, вы обслуживаетесь в порядке FIFO. Аналогичным образом в вычислительных системах очереди FIFO часто используются для обслуживания задач, которые необходимо выполнить по принципу: первым пришел, первым обслужен. Другим примером, иллюстрирующим различие между стеком и очередью FIFO, может служить ситуация в супермаркете: продукты в бакалейной лавке. Если бакалейщик выкладывает продукты на полки, а покупатель берет товары также с полки, то получается стек. Бакалейщик может стать путаным и сбиваться, поскольку первая задняя часть полки может быть очень долго в процессе использования. Бакалейщик выкладывает товары на заднюю часть полки, бакалейщик гарантирует, что время, в течение которого товар находится на полке, ограничивается временем, необходимым для приобретения всех товаров, выставленных на полку. Этот же принцип применяется во множестве покупных ситуаций.

Программа 4.14 Реализация очереди FIFO на базе связанного списка

Различия между реализацией FIFO и стеком на базе того типа (программа 4.13) состоит в том, что новые элементы добавляются в конец, а удаляются из начала. Следовательно, в этом случае требуется изменить `tail` на последний элемент списка, и теперь добавление функции `put` может добавлять в список новый элемент. Связываясь с этим, на программном уровне требуется изменить указатель `tail` так, чтобы указывал уже на последний элемент. Функции `QUEUE`, `get` и `empty` идентичны аналогичным функциям в реализации очереди на базе того типа на базе связанного списка из программы 4.6. Поэтому для новых функций понадобится только один список структур данных, а именно `tail`, и указатель `tail` должен указывать только на последний элемент.

```
typedef struct node
{
    int data;
    struct node *next;
} node;

void put(int x)
{
    struct node *new;
    new = (node *) malloc(sizeof(node));
    new->data = x;
    new->next = NULL;
    if (empty)
    {
        head = new;
        tail = new;
    }
    else
    {
        tail->next = new;
        tail = new;
    }
}

int get()
{
    if (empty)
    {
        return -1;
    }
    else
    {
        struct node *p = head;
        head = head->next;
        return p->data;
    }
}

void delete()
{
    struct node *p = head;
    while (p->next != NULL)
    {
        p = p->next;
    }
    p->next = NULL;
}
```

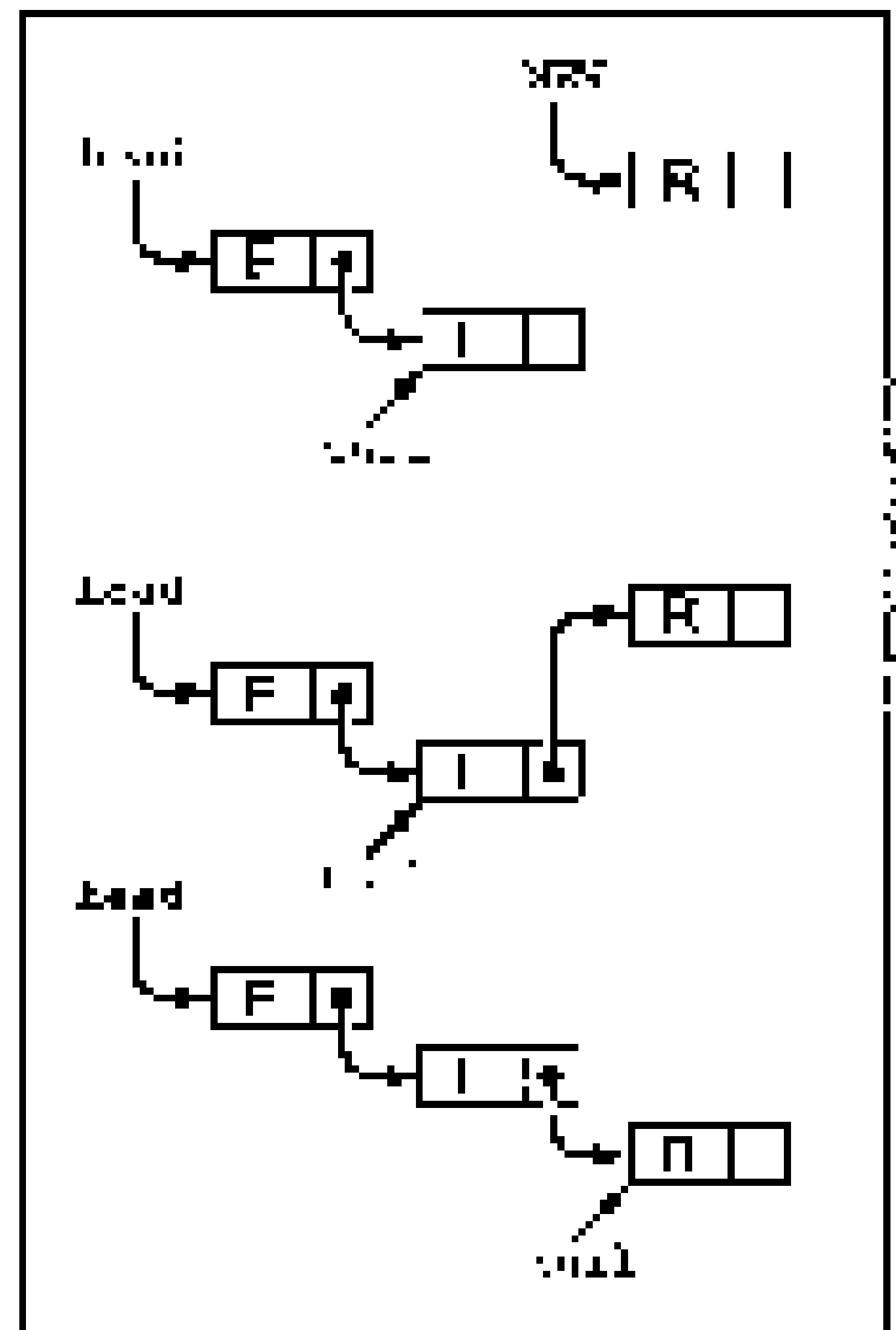


РИСУНОК 4.7 ОЧЕРЕДЬ НА БАЗЕ СВЯЗНОГО СПИСКА

В алгоритмической очереди в конце очереди (то есть последний элемент списка) добавляются новые элементы, а удаляются элементы из начала списка. Следовательно, для реализации очереди требуется изменить указатель `tail` на последний элемент списка. Функции `QUEUE`, `get` и `empty` идентичны аналогичным функциям в реализации очереди на базе того типа на базе связанного списка из программы 4.6. Поэтому для новых функций понадобится только один список структур данных, а именно `tail`, и указатель `tail` должен указывать только на последний элемент.

Для реализации очереди FIFO есть не только несколько вариантов, также при этом необходимо соблюдать осторожность в отношении, чтобы при выполнении операций добавления и удаления элементов было поставлено. Это условие означает, что удаление элементов очереди внутри массива, как это можно было

был предложен при буквенной интерпретации рис. 4.6. Следовательно, как и в реализации на базе стека, требуется поддерживать два индекса: индекс начала очереди и индекс конца очереди. Содержимым очереди являются элементы, индекс которых находится в рамках упомянутых двух индексов. Чтобы извлечь элемент, мы вычитаем его из начала (head) очереди, после чего индекс head увеличивается на единицу; чтобы вставить элемент, мы добавляем его в конец (tail) очереди, а индекс tail увеличивается на единицу. Как иллюстрирует рис. 4.8, последовательность операций вставки и извлечения приводит к тому, что все выглядит так, будто очередь движется по массиву. Она устроена так, что при достижении конца массива осуществляется переход на его начало. С детальным репликацией рассмотренного процесса можно ознакомиться в коде программы 4.15.

Программа 4.15 Реализация очереди FIFO на базе массива

В рассмотренной очереди отбрасываются все элементы массива, расположенные между индексами head и tail; при этом учитывается переход с конца на начало массива. Если индекс head и tail равны, очередь считается пустой, однако если они стали равными в результате выполнения операции push, то очередь считается не пустой. Так (иные решения не только ошибочные ситуации не выполняются, но также каждая операция на базе массива выполняется быстрее на элементе очереди, ожидающего обслуживания клиентом). При написании программы можно использовать следующие соображения.

```
template <class T> class Queue
{
private:
    const static int N = 1000; head, tail;
public:
    Queue(int maxN):
    { q = new T[maxN+1];
      N = maxN+1; head = 0; tail = 0; }
    int empty() const
    { return head % N == tail; }
    void put(const T& item)
    { q[tail++] = item; tail = tail % N; }
    T& get()
    { head = head % N; return q[head++]; }
};
```

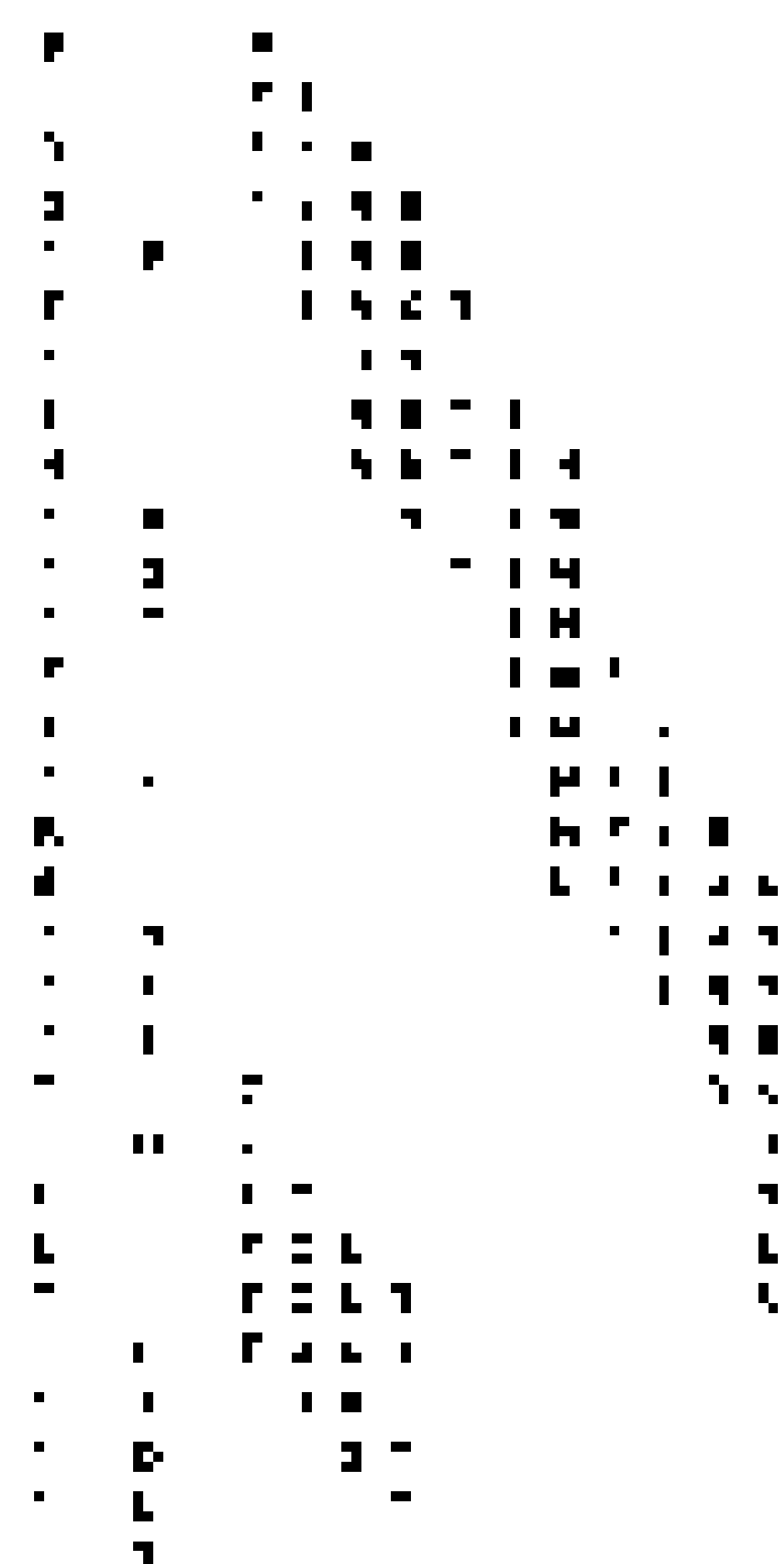


РИСУНОК 4.8 ПРИМЕР ОЧЕРЕДИ FIFO, РЕАЛИЗОВАННОЙ НА БАЗЕ МАССИВА

Для того чтобы реализовать очередь с использованием массива, необходимо: 1) определить количество элементов, которое может храниться в очереди; 2) реализовать операции вставки и извлечения элементов из очереди. Для реализации очереди с использованием массива, необходимо: 1) определить количество элементов, которое может храниться в очереди; 2) реализовать операции вставки и извлечения элементов из очереди. Для реализации очереди с использованием массива, необходимо: 1) определить количество элементов, которое может храниться в очереди; 2) реализовать операции вставки и извлечения элементов из очереди.

Пример 4.2. Для СДЛ "Морков. FIFO" неуместно реализовать операции `rel` и `put` с постоянным временем выполнения, поскольку для моркови, для любого момента

этот факт становится явным, стоит только немедленно провозвести на код программы 4.14 и 4.15.

Те же соображения относительно неэффективности алгоритмов памяти, которые более изложены в разделе 4.4, применимы и к очереди FIFO. Предположим на базе массива требуется реализация очереди оперативной памяти с помощью, достаточным для табулирования максимумом необходимого количества элементов очереди. В случае же представления на базе связанного списка оперативная память используется пропорционально числу элементов в структуре данных: это происходит за счет дополнительного расхода памяти на связи (между элементами) и дополнительного расхода времени на разрывления и освобождения памяти для каждой операции.

Хотя по причине фундаментальной разницы между списками и ресурсными программами (см. главу 5), со списками приходится сталкиваться чаще, чем с очередями FIFO, будут также встречаться алгоритмы, для которых очередь является естественной базисной структурой данных. Как уже отмечалось, очереди и списки используются в вычислительных приложениях чаще всего для того, чтобы отложить выполнение того или иного процесса. Хотя многие приложения, когда очередь является основой структуры данных, требуют зависимости от того, какие операции удаляются элементами воздействуют в операциях ввода, общее время выполнения программы не иерархически зависит, может зависеть от применяемой дисциплины. Когда в подобных приложениях встречаются большие количества операций ввода-вывода для удаления, выполняемых под структурами данных с богатыми возможностями, различия в производительности образуют существенную важность. Поэтому в настоящей книге столь существенно внимание уделяется таким АД. Если бы программа выполняла простую программу не интересная, могла было бы сделать единственной АД с оптимальным отношением к ресурсам, однако производительность программы является важным фактором, поэтому каждая программа, по сути, соответствует своему АД. Чтобы оценить эффективность отдельного АД, необходимо учитывать расходы (т. е. расход машинного времени) двух видов: издержки, обусловленные разницей, которые зависят от структуры данных, выбранных для реализации, и издержки, обусловленные отдельными программами выполнения операций. В заключение данного раздела необходимо отметить также АД, которые будут рассматриваться подробно в следующих главах.

Иногда возникает необходимость очереди FIFO является частным случаем более общего АД: обобщенной (generalized) очереди. Частные случаи обобщенной очереди различаются только правилами удаления элементов. Для этого для очереди будет так: "удалить элемент, который был введен последним"; для очереди FIFO правило гласит: "удалить элемент, который был введен первым"; существуют и другие варианты.

Простым, тем не менее, мощным вариантом является обобщенная очередь (generalized queue), подчиняющаяся следующему правилу: "удалить элемент, который был введен первым, который может ожидать, что с заданной вероятностью получит его

бой из элементов очереди. Используя представленные на базе массива (см. рис. 4.48), для неотрицательной очереди можно реализовать операции с постоянным временем исполнения. Представление на базе массива требует (как и в случае со стеком и очередью ЛIFO), чтобы операции глотать и выкидывать были распределены поровну. Однако в данном случае невыразительное представление на базе связанного списка можно привлекать тем в случае стека и очереди ЛIFO, поскольку эффективная реализация как в массиве, так и в списке одинакова, а в случае очереди ЛIFO — только в списке (см. рис. 4.49). Чтение с высокой степенью вероятности избежать конфликтов с помехами при вводе/выводе, поэтому очередь можно использовать в качестве базы для реализации рекурсивных алгоритмов (см. раздел 1.7).

При описании стека и очереди ЛIFO элементы идентифицировались по времени доступа к очереди. В качестве альтернативы эти абстрактные понятия можно описать в терминах последовательности первых упорядоченных элементов в базовых операциях во время удаления элементов в начале и конце списка. Если элементы добавляются в конец списка и удаляются также с конца, получается стек (точно как в реализации на базе массива); если элементы добавляются в начало и удаляются в начале, также получается стек (точно как в реализации на базе связанного списка); если же элементы добавляются в конец, а удаляются с начала, то получается очередь ЛIFO (точно как в реализации на базе связанного списка). В конечном итоге, если элемент добавляется в начало, а удаляется с конца, также получается очередь ЛIFO (этот вариант не соответствует ни одной из реализаций — для его точной реализации можно было бы хранить представление на базе массива, а вот представление на базе связанного списка для этой цели не подойдет из-за необходимости поддерживать указатели на конец очереди в случае удаления элементов в конце очереди). Разница между этими тремя случаями, необходимая для абстрактности типа данных *double-ended queue*, достигается с помощью очереди, в которой и добавление, и удаление разрешаются с обеих сторон. Его реализация на базе массива в качестве упорядоченной (см. рис. 4.45—4.47); при этом необходимо отметить, что реализация на базе массива является простым расширением программы 4.1а, а для реализации на базе связанного списка потребуется двусторонний список, вместо обычного списка, для чего можно будет пойти с одной стороны.

В главе 9 рассматриваются очереди с приоритетами, в которых элементы имеют номер, а порядок удаления элементов зависит как от даты элемента с самым низким номером, так и от даты элемента с самым высоким номером. Алгоритм "Очередь с приоритетами" полезен во множестве приложений, и потому представляет эффективную реализацию для этого АЛП в таких же многих областях, как исследование операций и компьютерные науки. Важные факторы исследования были также эффективны и использовались АЛП в приложении, посвященная новым алгоритмам поиска старой реализации в крупном, сложном приложении и сравнение результатов, можно сразу же определить, является ли новый алгоритм эффективным. Более того, отмечая, как он предоставляет новую реализацию некоторых общих проблем, алгоритм предлагается, можно сразу же определить, является ли новый алгоритм более эффективным, нежели старый. Структурные данные и алгоритмы, которые рассматриваются в главе 9 в плане решения данной проблемы, столь же интересны, сколько они важны и эффективны.

В них в 12-ти различных комбинациях (примечание). Это означает, что в каждой из которых у человека есть свои ключи, в принципе удачные, так как он может изменить элемент, один из которых будет другим. Если такой элемент существует. Это АИИ. Можно, если можно из него сделать, и можно быть полноточным: в сочетании с другими.

Каждый из этих АЧД дает возможность решить задачу, которую невозможно решить в результате непосредственного изучения конкретных признаков и признаков: но тем не менее эти задачи, в частности, в 7 и 4-й областях со множественными примерами, являются классификация обобщенных случаев, имеющих и еще более оригинальные АЧД, которые могут быть получены независимо друг от друга.

Упражнения

В 4.36 Н: найти максимальное значение $u[i]$. $u[i+1]$ — это значение программы 4.15 с переменной, показанной на рис. 4.6. Считайте, что $\max N$, как в на рис. 4.8, равно 1).

1: 4.37 B .18C104C2H1.1645.1H

745 / 1 / ONE , . . 37 . / / 13 / M ' ' 1

Значит, отсюда следует, что операция put и операция get являются последовательными операциями. Возвращаемая операция get , когда эта последовательность влечет, что выполняется или эквивалентно этой операции put .

4.38 Микропрограмма применяется для тестирования работы ППД на базе микросхем. Программу 4.10) так, чтобы в ней выполнялась функция $\text{exp}(x)$, скомпилированная на языке ассемблера. Операцию $\text{exp}(x)$ можно реализовать, используя операцию $\text{exp}(x)$, скомпилированную на языке ассемблера.

4.39 Моделирование превращения в текст реализовано через процедуру `TO` на базе функции `to_text` с текстом программы 4.141 так, чтобы в ней выполнялась функция `split`, если символ является символом строки, иначе выводился пустой список. Для этого процедура `TO` использует оставшуюся память в пиле.

ר. 4.400 : 1 אגרתהגדולההעתיקה

$$E_{\text{kin}} + U + U_{\text{ext}} = E + pL + \frac{1}{2} \rho \omega^2 L^2 = -$$

Бука — операция раскрытия значения операндов до и после знака. Бука: значение регистра — операция до и после знака, знак плюс означает операцию до и после, а минус — операция до и после. Неудача при раскрытии значения — ошибка, возникающая при раскрытии до и после знака. Неудача при раскрытии значения — ошибка, возникающая при раскрытии до и после знака.

[illegible][illegible]

слова, инкапсулируя ее как пользовательскую операцию над стеком в смысле упр. 4.41, получить новую последовательность.

• 4.43 Заполните интерфейс для A111 'дек'.

4.44 Для интерфейса дека (упр. 4.43) напишите реализацию, в которой в качестве базовой структуры данных используется массив.

4.45 Для интерфейса дека (упр. 4.43) напишите реализацию, в которой в качестве базовой структуры данных используется двусторонний список.

4.46 Для приведенного в тексте интерфейса очереди FIFO (программа 4.14) напишите реализацию, в которой в качестве базовой структуры данных используется двусторонний список.

4.47 Напишите программу-клиент, которая проведет экспериментальный A111 'дек' (упр. 4.43), считывая с командной строки в качестве первого аргумента строку команды подобную приведенной в упр. 4.40, показывая выходящие указатели экрана. В интерфейсе и реализации добавьте структуру-член `printf` распечатывающего содержимое дека после каждой операции, как это показано на рис. 4.6.

• 4.48 Создайте A111 'Неупорядоченная очередь' (напишите интерфейс и реализацию), в котором в качестве базовой структуры данных используется массив. Обеспечьте для каждой операции постоянное время выполнения.

•• 4.49 Создайте A111 'Неупорядоченная очередь' (напишите интерфейс и реализацию), в котором в качестве базовой структуры данных используется связный список. Напишите как можно более эффективные реализации операций `insert` и `delete` и привелируйте связанный с ними издержки для наилучшего случая на входных данных.

• 4.50 Напишите программу-клиент, которая выбирает для операции `delete` произвольным образом элемент в неупорядоченную очередь чисел от 1 до 99, а затем удаляет этот элемент и выводит результат.

4.51 Напишите программу-клиент, которая считывает в командной строке в качестве первого аргумента целое число N , а затем распечатывает результат удаления в деке карт из N пиков. Для этого она должна загрузить в неупорядоченную очередь N элементов (см. упр. 4.3) и затем распечатать результат выбора на этой очереди пяти карт за один раз.

• 4.52 Напишите программу, которая решает задачу связности. Для этого она должна получать все пары в неупорядоченную очередь, а затем с помощью алгоритма взвешенного быстрого поиска (программа 1.3) включать их в очередь.

4.7 Повторяющиеся и индексные элементы

Во многих приложениях обрабатываемые абстрактные элементы имеют одну или несколько характеристик, позволяющих представить их в виде массива. Это часто бывает так, что элементы можно упорядочить по какому-либо признаку, например по значению функции, характеризующей их. В частности, в данном разделе рассматриваются такие изменения стандартных стеков, очередей FIFO и обобщенных очередей, которые применяются в этих структурах данных при наличии повторяющихся элементов.

Например, компания, поддерживающая список растений по адресам покупателей, может потребоваться расширить этот список информацией из других списков, собранных с различных источников. Это предполагает с помощью соответствующих операций найти, где этот, по-видимому, не требуется заносить информацию о растениях, адреса которых уже присутствуют в списке. Далее можно будет увидеть, что тот же принцип применяется в самых разнообразных приложениях. В качестве еще одного примера рассмотрим задачу маршрутизации сообщений в сложной сети перекрестных линий. Можно пользоваться первыми сообщениями одновременно по нескольким маршрутам, однако каждому отдельному потоку узлу сети не требуется только одна копия этого сообщения в своей внутренней структуре данных.

Одним из подходов к решению данной проблемы — это возложить на программистов задачу по обеспечению того, что в АЛД не будут присутствовать повторяющиеся элементы. Предположительно, программисты могли бы выполнять эту задачу, используя какой-нибудь другой АЛД. На практике цель создания любого АЛД связана с обеспечением быстрого решения прикладных задач с помощью вычислительных ресурсов, можно прийти к выводу, что обнаружение и устранение дубликатов элементов и решение таких ситуаций — это часть задачи, относящаяся к компетенции АЛД.

Использование структуры данных при создании повторяющихся элементов сводится к изменению абстрактного интерфейса, изменению операций и прочее для такого АЛД. Будет так или иначе, как и для соответствующего АЛД, в котором данный принцип не используется. Его функциональное равенство реализации является фундаментальным требованием. Вообще говоря, при модификации описания какого-нибудь АЛД получается совершенно новый АЛД — АЛД, обладающий совершенно другими свойствами. Эта ситуация демонстрирует также необходимость при спецификации АЛД: обеспечить, чтобы конкретные реализации и реализации принадлежали одной спецификации интерфейса, достаточно трезво, однако обеспечить применение такого высокоуровневого принципа, как данный — совсем другое дело. Тем не менее, мы заинтересованы в подобных алгоритмах, поскольку программисты получают такие свойства для реализации данных способов, а реализации могут показывать, представляемые полными программными, для обеспечения более эффективного решения задач.

На рис. 4.9 проиллюстрирована работа модифицированного АЛД "Стек без повторяющихся элементов" для

1			
2		1 2	
3			
4	1	1 2	
5		1 2 3	
6		1 2 3 4	
7		1 2 3 4 5	
8		1 2 3 4 5 6	
9		1 2 3 4 5 6 7	
10		1 2 3 4 5 6 7 8	
11		1 2 3 4 5 6 7 8 9	
12		1 2 3 4 5 6 7 8 9 10	
13		1 2 3 4 5 6 7 8 9 10 11	
14		1 2 3 4 5 6 7 8 9 10 11 12	
15		1 2 3 4 5 6 7 8 9 10 11 12 13	
16		1 2 3 4 5 6 7 8 9 10 11 12 13 14	
17		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	
18		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	
19		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	
20		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18	
21		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19	
22		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	
23		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	
24		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22	
25		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	
26		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24	
27		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	
28		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26	
29		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27	
30		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28	
31		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	
32		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	
33		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	
34		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	
35		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33	
36		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34	
37		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35	
38		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36	
39		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37	
40		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38	
41		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39	
42		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	
43		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	
44		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	
45		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	
46		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	
47		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	
48		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	
49		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	
50		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48	
51		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49	
52		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50	
53		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51	
54		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52	
55		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53	
56		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54	
57		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55	
58		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56	
59		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57	
60		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58	
61		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59	
62		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60	
63		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61	
64		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62	
65		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63	
66		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64	
67		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65	
68		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66	
69		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67	
70		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68	
71		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69	
72		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70	
73		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71	
74		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	
75		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73	
76		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74	
77		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75	
78		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76	
79		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77	
80		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78	
81		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79	
82		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80	
83		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81	
84		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82	
85		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83	
86		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84	
87		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85	
88		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48	

исход. элементу применяется тот же код, а другой добавляется в очередь. Если установка элемента вызвала ошибку проверки, то он в стек такого элемента. Если возникла ошибка при установке элемента игнорируется. Это решение не зависит от используемого представления списка — на базе массива, списка или чего-нибудь еще. Для реализации алгоритма удалить старый элемент требуется изменить работу программы (рис. 4.57).

Применяя изложенное выше, можно сказать, что язык на стеклах представляет стек без упорядочивающих элементов, функционирующий по принципу "интегрировать новый элемент". состоит в поддержке двух структур данных. Первая, как и прежде, содержит элементы стека и позволяет обращаться к ним по адресу, в котором лежит соответствующий элемент стека. Вторая структура данных является массивом, позволяющим отслеживать, какие элементы находятся в стеке: индексом или этим индексом являются элементы стека. Вектор рассматриваемых массивов рассматривается как частный случай реализации таблицы символов, которая обсуждалась в разделе 12.2. Кроме того, что элементы представляют собой целые числа и индексы от 0 до $M-1$, эту же идею можно применить по отношению к любой обобщенной очереди.

Программу 4.15 Стек индексов элементов, в которых запрещены повторяющиеся элементы

[illegible]

```

template <class T>
class STACK
{
private:
    T** arr; int N;
public:
    STACK(int maxN)
    {
        arr = new T*[maxN]; N = 0;
        = new Item[maxN];
        for (int i = 0; i < maxN; i++) arr[i] = 0;
    }
    int empty() const
    { return N == 0; }

```

E		F	
I		F I	
N		F I N	
S		F I N S	
'	F	I N S	
T		I N S T	
'	I	N S T	
I		N S T I	
H		N S T I H	
'	N	S T I	4
'	S	- ; H	
'	T	I H	
F		; H F	
I		4 F I	
'	H	F :	
H		F ' C	
S		F S S	
'	F	: H S	
'	I	F S	
'	H	S -	
T		- -	
'	R	I	
S		P	
H		S -	
I		" "	
'	C	-	
'	D	-	
A			

РИСУНОК 4.7 ОЧЕРЕДЬ ПРО
БЕЗ ПОВТОРЯЮЩИХСЯ
ЭЛЕМЕНТОВ.

ФУНДАМЕНТЫ: 49 ВО
ПРИНЦИПЫ "УДАЛЕНА
СТАРЕША ЭЛЕМЕНТ"

[illegible]


```

void push(T* t) {
    if (t[1] == 1) return;
    a[n++] = *t; t[1] = 1;
}
T* pop() {
    if (a[--n] == 0) return 0;
}

```

Этот частный случай встречается довольно часто. Он наиболее типичный пример того, как сложная структура данных может быть реализована на основе, состоящей из элементов, называемых *элементами стека* (stack *element*). Обычно имеется набор из N объектов, хранящихся в каком-то другом массиве, в куче или в более сложной структуре. Необходимо переместить этот набор объектов через структуру обобщенной очереди. Объекты выносятся в очередь по очереди и обрабатываются при удалении, причем каждый объект должен обрабатываться только один раз. Проблема для проектирующего элементов, в которых используются классы массива, позволяет напрямую достичь этой цели.

Каждый из этих вариантов (или еще один или несколько) элементов, которые входят или не входят элемент ведет к новому АТД. Различия могут заключаться незначительными, тем не менее, они зависят от динамических характеристик АТД (как это зависит от стороны клиентских программ), а также от выбора алгоритма и структуры данных для реализации различных операций. По сути, это неструктурный, как и как структура, но это АТД как элемент. Кроме того, необходимо учитывать и другие варианты, например, может ли элемент желать иметь интерфейс с целью идентификации элементов и управлять тем, что эти элементы представляют собой, или же представлять произвольные возможности выбора: идентифицировать новый элемент или удалить старый.

Когда мы рассмотрим абстрактный такой термин, как *абстрактный стек* (abstract *stack*), мы всегда по умолчанию имеем в виду, что мы имеем дело с абстрактными типами данных, где каждый тип имеет свой набор операций. Набор операций с абстрактными типами операций, причем для эффективной реализации этих операций каждый тип требует своей, в некоторых случаях довольно сложной, реализации.

Упражнения

4.43 Для стека из повторяющихся элементов, реализованного по принципу "удалить старый элемент", нарисуйте рисунок, иллюстрирующий рис. 4.9.

4.44 Модифицируйте стандартную реализацию стека на базе массива из раздела 4.4 (программа 4.7) таким образом, чтобы в ней была реализована повторяющаяся операция по принципу "игнорировать данный элемент". Используйте метод "трубчатый стиль", заключающийся в сканировании всего стека.

4.45 Модифицируйте стандартную реализацию стека на базе массива из раздела 4.4 (программа 4.7) таким образом, чтобы в ней была реализована повторяющаяся операция по принципу "удалить старый элемент". Используйте метод "трубчатый стиль", заключающийся в сканировании всего стека в направлении его элементов.

- 4.56 Выполните упражнения 4.54 и 4.55 для реализации стека на базе сбалансированного списка из раздела 4.4 (страницы 4.8).
- 4.57 Разработайте реализацию стека на основе тэгов, в которой реализовываемые элементы запрашиваются по принципу "удалить старший элемент". Элементы стека являются целые числа в диапазоне от 0 до $48^{15} - 1$, а операции над ними должны выполняться в режиме вычисления. Необходимо реализовать реализацию стека на базе двустороннего списка и использовать указатели на узлы этого списка, а не массивы индексных значений.
- 4.58 Выполните упражнения 4.54 и 4.55 для очереди ПГО.
- 4.59 Выполните упражнения 4.56 для очереди ПГО.
- 4.60 Выполните упражнения 4.57 для очереди ПГО.
- 4.61 Выполните упражнения 4.54 и 4.55 для реализованной очереди.
- 4.62 Напишите клиентскую программу для АТД, полученную в упражнении 4.61, которая использует реализованную очередь для извлечения элементов.

4.8 АТД первого класса

Интерфейсы и реализации АТД "Стек" и "Очередь ПГО" в разделах с 4.2 по 4.7 являются важной частью сокращенной клиентской программы, которую мы изучаем и реализуем. Эти АТД не только помогут и будут служить в качестве основы для создания других реализаций, рассматриваемых в книге.

Однако, когда такие типы данных используются в программах таким же образом, как и встроенные типы данных, например, `int` или `float`, программы могут работать быстрее. В данном разделе мы рассмотрим, как конструировать АТД, с которыми программы-клиенты могут работать так же, как и со встроенными типами данных, без нарушения принципа сокращения затрат реализации от клиентских программ.

Определение 4.4 Тип данных первого класса — это тип данных, который может использоваться в программах таким же образом, как и встроенные типы данных.

Например, тип данных первого класса можно использовать как переименованный (т.е. не иметь переименований этих типов и присваивать им значения). Такие типы данных можно также использовать в аргументах и возвращаемых значениях функций. В этом определении, равно как и в других, относящихся к типам данных, нельзя достигнуть точности, не углубившись в вопросы истории, классификации операций. Как будет показано, важно было утверждать, что можно считать $a = 0$, $a \neq 0$ и b не являются объектами интеллектуально понимаемого класса, и совсем другое дело — точно определить, что означает эта операция.

В классе можно предположить, что все типы данных имеют базовый универсальный набор можно определенных операций, в действительности же каждый тип данных характеризуется своим собственным набором операций. Это означает, что каждая типная сущность сама по себе представляет точному определению типа данных, характерных для каждого, но это предположение необходимо дать определение, что такое определение, заданный для встроенных типов данных, а это делается редко. Чаще всего важными являются лишь несколько критических операций и полное достояние применяется только для объектных типов данных так же, как и для встроенных типов

Во многих языках программирования создание типов данных первого класса является делом трудным или даже невозможным: в языке C — преобразовывая базовые конструкции — это константные выражения и возможность переопределения переменных. Действительно, в языке C++ есть возможность создавать классы, которые являются типами данных первого класса; более того, является четкий способ инкапсуляции тех классов, которые таковыми не являются.

Метод, который применяется в языке C++ для реализации типов данных первого класса, применим к любому классу: в частности, он применим к абстрактным классам и, таким образом, объектам, реализующим интерфейс создания объектов, операций со значениями и операциями ПГО по числам действительных, как и к другим типам данных C++. При изучении алгоритмов для точности работы с числами важно понимать, что она предоставляет единственный способ выражения вычислительных операций, относящихся к таким АТД. Например, проще всего генерировать значения комплексного числа двумя способами — с помощью одной операции. Далее будут рассматриваться алгоритмы, которые реализуют такие операции для АТД "Стереонаблюдатель" (глава 9) и "Таблица символов" (глава 19).

Если доступ к типу данных первого класса осуществляется только через интерфейс, то это действительно АТД первого класса (см. определение 4.1). Определение интерфейса — это набор функций, объявленных в АТД; в основном, так же, как это происходит с типами данных, например, `int` или `float`, — важно иметь интерфейс для каждого программируемого абстрактного уровня, поскольку это гарантирует целостность абстрактного уровня на низшем уровне. Каждый объект имеет реализацию. В таком случае знание того, что реализация имеет конкретные свойства, не работает, или, наоборот, системы, которые имеют реализацию, но не абстрактные операции, что, в свою очередь, означает, что реализация абстрактных операций должна быть реализована с помощью любого применения. Как приложения программ, для которых компьютерная среда не важна.

Программа 4.17 Драйвер комплексных чисел (корни из единицы)

Эта программа проверяет корректность реализации абстрактного уровня комплексных чисел, который позволяет проводить вычисления непосредственно с комплексными числами. В этой главе объяснено, как использовать этот драйвер к заданному в арифметической библиотеке C++ объекту. Для программы требуется реализовать АТД комплексных чисел, который реализован в файле. При компиляции с помощью компилятора C++ с помощью опции `-std=c++11` (см. главу 4.12) производится вывод таблицы, показанной на рис. 4.12.

```
#include <iostream>
#include <stdlib.h>
#include <iomanip>
#include "COMPLEX.C"
int main(int argc, char *argv[])
{
    int N = atoi(argv[1]);
    cout << N << " complex roots of unity" << endl;
    for (int k = 0; k < N; k++)
    {
        float theta = 2.0*3.14159265358979/k;
        complex z(cos(theta), sin(theta)); // k = 0
        cout << z << " : " << z << " : " << endl;
        for (int j = 0; j < N-1; j++) // k = 1..N-1
            cout << z << endl;
    }
}
```


Для начала в качестве примера рассмотрим АТД первого класса, соответствующий абстракции `ComplexNumber`. Наша цель — получить возможность записывать программы, подобные программе 4.17, которая выполняет алгебраические действия над комплексными числами, используя операции, описанные в АТД. Данная программа объявляет и имплементирует комплексные числа, а также выполняет операции `+` и `*`. Можно было бы рассмотреть и другие операции, но в качестве примера достаточно рассмотреть только эти две. На практике используется класс `Complex` из библиотеки `CLN`, в котором имеется обширный набор перемноженных операций, включая даже тригонометрические функции.

В программе 4.17 применяются все основные свойства рациональных свойств комплексных чисел. Стоит немного отклониться от основной темы и кратко рассмотреть эти свойства. В некотором смысле это даже не отступление, поскольку связь между фундаментальными свойствами между комплексными числами как математической абстракцией и их представлением в компьютерной программе исторически интересна.

Число $i = \sqrt{-1}$ называется мнимым числом. Хотя $\sqrt{-1}$ как математическое число не имеет смысла, мы называем i и выполняем над i алгебраические операции, заменяя i^2 на -1 всякий раз, когда его потребуем. Комплексное число состоит из двух частей: вещественной и мнимой — так же, как число можно записать в виде $a + bi$, где a и b — вещественные числа. Для умножения комплексных чисел применяются обычные алгебраические правила, при этом всякий раз i^2 заменяется на -1 . Например:

$$(a + bi)(c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i.$$

При умножении комплексных чисел вещественные или мнимые части могут сокращаться (принимать значение 0), например:

$$(1 - i)(1 - i) = 1 - i - i + i^2 = -1;$$

$$(1 + i)^2 = 4i^2 = -4;$$

$$(1 + i)^3 = 16.$$

Решая обе части последнего уравнения на $16 = (\sqrt{2}i)^4$, мы находим, что

$$\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}} \right)^4 = 1.$$

Важно отметить, имеется много комплексных чисел, которые при возведении в степень дают 1. Они называются комплексными корнями из единицы. Действительно, для каждого N имеется ровно N комплексных чисел z , для которых справедливо $z^N = 1$. Легко можно показать, что этим свойством обладают числа:

$$\cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right),$$

где $k = 0, 1, \dots, N-1$ (см. упр. 4.58). Например, если в этой формуле взять $k = 1$ и $N = 8$, получим корень восьмой степени из единицы, который мы также уже видели.

Программа 4.17 выполняет все корни N -ой степени из единицы для любого натурального N и затем возводит их в N -ую степень, используя операцию `^`, определенную в данном АТД. Выходные данные программы показаны на рис. 4.18. При этом важ-

достаток, что значение из этого класса, возведенное в N -ую степень, дает один и тот же результат — 1 или $-1 + 0i$. Полученные значения почти во всем строго соответствуют ограниченной точности вычисления. Интерес к этой программе заключается тем, что она использует класс `Complex` точно так же, как обычный тип данных. Шрифт будет привычно показан, чтобы это было возможно.

Даже в этом простом примере видно, что тип данных был абстрактным, поскольку имелось еще одно стандартное представление, которым можно было бы пользоваться — полярные координаты (см. упр. 4.17). Программа 4.18 — это интерфейс, который может использоваться таким образом, как программа 4.17; а программа 4.19 — это реализация, в которой используется стандартное представление данных (для числа типа `float` для вещественной части и другое — для мнимой).

Программа 4.18 Интерфейс АТД первого класса для комплексных чисел

Этот интерфейс для комплексных чисел позволяет реализовать создать объект типа `Complex` (я реализовал его для типа `float` как тип `Float`), абстрактные функции $+$ и $-$ для сложения и вычитания, а также для комплексных операций $*$ и $/$. Хотя это и не стандартные стандартные механизмы, действующие для всех классов, интерфейс использует объект класса `Complex` в операциях преобразования, а также в операции для преобразования значения функции.

```
class Complex
{
private:
    // программный код.
    // зависящий от реализации
public:
    Complex(float, float);
    float Re() const;
    float Im() const;
    Complex operator+ (Complex);
};
```

Когда в программе 4.17 мы показали $x = 4$, так x и 4 являются объектами класса `Complex`, система распределяет память для нашего объекта и копирует в новый объект значение, относящееся к объекту 4. Если использовать объект класса `Complex` как аргумент или возвращаемое значение функции, план будет таким же. Кроме того, когда объект выведен из процесса общения, система освобождает связанную с ним память. Например, в программе 4.17 система освобождает память, связанную с объектами i и x класса `Complex`, после цикла `for` так же, как и память, связанную с переменной i типа `Point`. Кратко говоря, класс `Complex` используется полностью автономным образом, т.е. он относится к типу данных первого класса.

0	1.000	0.000	1.000	0.000
1	0.707	0.707	1.000	0.000
2	0.000	1.000	1.000	0.000
3	-0.707	0.707	1.000	0.000
4	-1.000	0.000	1.000	0.000
5	-0.707	-0.707	1.000	0.000
6	0.000	-1.000	1.000	0.000
7	0.707	-0.707	1.000	0.000

РИСУНОК 4.12 КОМПЛЕКСНЫЕ КОРНИ ИЗ ЕДИНИЦЫ

Эта таблица содержит комплексные корни из единицы, полученные в программе 4.17, когда мы вычисляли с помощью $N=8$. В результате вычисления объект `Complex` использовался для формирования формулы вычисления комплексных функций (см. упр. 4.20). Доказательство каждой из формул дано в табл. 4.1.

$$\begin{matrix} \sqrt{2} & \sqrt{2} \\ 1 & 1 \\ 2 & 2 \end{matrix}$$

(для $N=8$ корней). Для вычисления N корней из единицы все N корней из единицы $1 + 0i$ разделим на N равных частей.

Программа 4.19 АТД первого класса для комплексных чисел

Этот код реализует АТД, описанный в программе 4.18. Для представления вещественной и мнимой частей комплексного числа используются данные типа `float`. Это тип данных первого класса, так как в представлении данных присутствуют указатели. Когда объект класса `Complex` применяется к себе в качестве аргумента, либо в качестве аргумента функции, либо как ее значение возврата, система делает его копию, решившая в памяти создать объект и копировать данные в память как в случае с обычными типами данных.

Параграфовая аннотация к этой функции описывает выполнение данных по структуре (см. стр. 4.16).

```

#include <iostream.h>
class Complex
{
private:
    float re, im;
public:
    Complex(float x, float y)
        ( re = x, im = y );
    float Re() const
        { return re; }
    float Im() const
        { return im; }
    Complex operator+(const Complex& rhs)
        { float t = Re();
          re = Re()*rhs.Re() + Im()*rhs.Im();
          im = re*rhs.Im() + Im()*rhs.Re();
          return *this;
        }
};

ostream& operator<< (ostream& o, const Complex& c)
{ o << c.Re() << " + i " << c.Im(); return o; }

```

Довольно легко, в языке C++ любой класс, обладающий свойством, что ни один из его данных-членов не является указателем, становится в членом данных первого класса. При копировании объекта копируется каждый его член: когда объект присваивается значению какого-нибудь иного объекта, то каждый его член переатрибутируется; когда объект является значением области видимости, значимость он имеет автоматически. В системе существует стандартный механизм функционирования в каждой из упомянутых ситуаций для выполнения необходимых функций в каждом классе: конструктор копирования по умолчанию конструктор деструктора, получающий аргументом `*this`.

Однако, когда некоторые данные-члены являются указателями, ситуация становится функцией, используемая по умолчанию, оказывается совершенно другой. В стандартном приспосабливаемом конструкторе копирования, неопределенный (или указатель), должен копироваться указателем, но действительно ли это то, что нам требуется? Это важный вопрос фундаментального характера, который необходимо решать каждый раз при проектировании любого АТД. Или, если говорить более широко, вопрос управления памятью или является решающим, когда при разработке программного обеспечения используются АТД. Далее приводится пример, который поможет понять эти вопросы более детально.

Здесь, правда, подразумевается, что класс `QUEUE` принадлежит к тому же типу данных, что и класс `int`. Эта поправка не будет корректно функционировать с реализацией, представленной в программах 4.4 и 4.15, но является неграмотной попыткой копирования из внеязычного языка `for`.

Программа 4.4: "Очередь" в программе 4.22 имеет ко-структор ко-структа `int`, который не является стандартным. Единственным путем для того, чтобы избежать необходимости ко-структа для объекта `q` в памяти, это то, что ко-структор помещен в функцию `main` в начале, завершающей.

```
#include <iostream.h>
#include <stdlib.h>
#include "queue.h"
const const int N = 4;
int main(int argc, char *argv[])
{ int N = atoi(argv[1]);
  QUEUE* q[10];
  for (int i = 0; i < N; i++, cout << endl)
  { int m = rand(); % N; int n = rand() % N;
    queue[i].put(m);
    cout << i << " " << m << "\n";
    if (!queue[i].empty())
      cout << queue[i].get() << " " << m;
    cout << endl;
    for (int k = 0; k < N; k++, cout << endl)
    { QUEUE* q = queue[k];
      cout << k << " " << "\n";
      while (!q.empty())
        cout << q.get() << " " << "\n";
    }
  }
}
```

Еще хуже обстоят дела, когда объект класса `QUEUE` захватывает ресурсы внешнего источника, требующие памяти, стиснувшись в указатель. Но не вся память, занимаемая объектом, является внешним ресурсом. Например, как только указатель прекратит свое существование, не будет возможности доступа к данной области памяти. Вот наш пример: `main` выводит `(main) (0x00000000)` — серийный и тайпем, после требуемой работы выводится `0` — адрес нулевого объекта, который начал свою существование в `main`.

В языке C++ существуют специальные механизмы, упрощающие создание классов, позволяющие корректно соединять ко-структоры и деструкторы, а также указывать. В частности, в `main` можно включить такие функции члена:

- **Конструктор деструктора** — для того чтобы создать новый объект, который является копией данного объекта.
- **Перезагрузка деструктора деструктора** — для того чтобы объект мог присутствовать в памяти столько операторов прикладного.
- **Деструктор** — для того чтобы освободить память, зарезервированную под объект на время его существования.

Как же системе необходимо выполнять указанные операции. Она использует эти функции-члены. Если они не включены в класс, система ориентируется на стандартный

функциям. Они работают так, как это описано для класса `Simplex`, однако ведут к локальной реализации координат и указателя на узел памяти, если какому-нибудь элементу даны следующие указания. Программа 4.21 — это интерфейс очереди ГРО, в которой заданы три переопределенных функции. Подобно конструкторам, они имеют стандартные сигнатуры, в которых входят имя класса.

Программа 4.21 Интерфейс АТД первого класса "Очередь"

Член-функция `Queue` принимает указатель на объект, который должен содержать указатели, больше подходящие для хранения тип данных, а его интерфейс наследуется от абстрактного интерфейса, переопределенную функцию `enqueue` и деструктор, как это описано в данном разделе простого интерфейса очереди ГРО на странице 4.18.

```
template <class T>
class Queue
{
private:
    // Implementation-dependent code
public:
    Queue(int);
    Queue(const Queue&);
    Queue& operator=(const Queue&);
    ~Queue();
    int empty() const;
    void put(const T&);
    T& get();
};
```

Когда при создании объекта присваивается начальное значение либо объект передается как параметр или используется в функции, система автоматически вызывает конструктор копирования `Queue(const Queue&)`. Операции присваивания `Queue& operator=(const Queue&)` выполняется в случае применения операции `=`, дабы присвоить значение одной очереди другой. Деструктор `~Queue()` выполняется в том случае, если необходима освобождать память, связанную с какой-либо очередью. Если в программе присутствует объявление без установки начального значения, то по умолчанию `Queue<int> q;` (или создание объекта `q` еще она вызывает конструктор `Queue()`). Если объект явно инициализируется в процессе такого объявления, как `Queue<int> q = p` (или эквивалентной форме `Queue<int> q(p)`), система использует конструктор копирования `Queue(const Queue&)`. Эта функция должна создавать новую копию объекта `p`, а не просто ссылаться на него. Как обычно для множественных параметров, ключевое слово `const` выражает намерение не изменять объект `p`, но не запрещать его доступ для доступа к информации и ее модификации.

Программа 4.22 представляет собой реализацию конструктора копирования, переопределенной операции присваивания и деструктора для реализации очереди на базе связанного списка из программы 4.14. Деструктор `delete` переопределен по всей очереди и с помощью `delete` освобождает память, распределенную под каждый узел. Если клиентская программа предоставляет некоторый объект в качестве аргумента операции присваивания не предполагая, что такая-либо операция в данном случае вызовет деструктор, а затем с помощью `put` копируется каждый элемент очереди, это

лей выраза от языка сортировки. Конструктор копирования создает очередь (т.е. делает ее пустой) и типиз с помощью операции присваивания выполняет копирование.

Из-за перегрузки в классе конструктора копирования, перегруженной операцией присваивания и деструктора (см. ниже), можно заметить, как это делается в программе 4.22, можно предположить, любая класс языка C++ в АТД первого класса. Эти функции обычно действуют по принципу простого обмена структур данных. Однако эти дополнительные шаги предпринимаются не всегда, так как:

- часто используется только один экземпляр объекта определенного класса;
- даже при наличии нескольких экземпляров может оказаться необходимым преобразование непосредственного копирования структуры данных.

Короче говоря, создавая возможность задать тип данных первого класса, мы создаем также необходимость достижения компромисса между качеством и скоростью. Поэтому приходится пойти на его реализацию и использовать, в первую очередь, когда это кажется наиболее удобным.

Программа 4.22 Реализация очереди первого класса на базе связанного списка

Есть полная и полная реализация реализации класса "Очередь FIFO" из программы 4.14 и программа 4.22, которая является реализацией очереди. Для этого класса требуется дубликат, приведенный ниже реализация конструктора копирования, перегруженной операцией присваивания и деструктора. Эти функции являются функциями, используемыми по умолчанию, и вызываются, когда необходимо копировать объект объекта.

Деструктор "QUEUE" является функцией, которая удаляет очередь, проходя по списку, создавая список, вызывая для каждого узла функцию delete. Таким образом, когда объект удаляется, освобождаясь от пользователя, освобождается также все память, занимаемая объектом.

Если перегруженная операция копирования вызывается для списка, когда объект привносится список, создается дубликат, т.е. q = q', и никакие действия не выполняются. В противном случае вызывается функция delete, чтобы удалить элемент, освобождая от памяти каждый объект (в результате получаются пустой список). Затем для каждого элемента списка, который содержится в списке q, вызывается функция delete, которая освобождает память, занимаемую объектом, и таким образом создается список этого списка. Таким образом, возвращаемый список является копией списка, на котором вызван объект, который вызывается значением другого объекта.

Конструктор копирования QUEUE(const QUEUE& q): объявляет очередь и создает копию списка, создавая очередь, присваивая созданный список созданного аргумента.

```
private:
    void deleteList()
    {
        for (link = head; link != 0; link = link->next) delete link;
    }
public:
    QUEUE(const QUEUE& q)
    { head = 0; *this = q; }
    QUEUE operator=(const QUEUE& q)
```



```

1
    if (this == &obj) return "this";
    delete[] t;
    link t = new obj;
    while (t != 0)
        { put(t->item); t = t->next; }
    return "this";
}
-queue()
{ delete[] t; }

```

Принадлежные рассуждения также объясняют, почему библиотека ядра C++ не предоставляет класс `string`, несмотря на привлекательность итерационной структуры данных типа строки в стиле языка C. С строки не принадлежат к типу данных первого класса, поскольку они не содержат лучше указателей. Вспомогательные программы не только впервые сталкивались с некорректным связанным копированием, когда копирует указатель на C-строку, ожидая, что будет копироваться сама строка. В противоположность этому, класс `string` языка C++ является типом данных первого класса, поэтому при работе с большими строками заслуживает особую осторожность.

В качестве другого примера можно привести программу 4.20 таким образом, чтобы она периодически распечатывала лишь несколько первых элементов очереди. И можно было бы объяснить простейшие операции языка C++, когда они становятся очень большими. Однако, когда очередь будет очень большой, произойдет нечто, что трудно проанализировать самостоятельно, поскольку при инициализации локальной переменной в цикле `for` вызывается конструктор копирования, который создает копию всей очереди, даже если требуется всего лишь несколько элементов. Далее, в конце каждой итерации локальная переменная, объявленная оператором `delete`, так как она связана с локальной переменной. Для программы 4.20 в нынешнем виде, когда выполняется доступ к каждому элементу копии, дополнительно требуется освобождение памяти, что является очень медленным процессом на некоторый произвольный коэффициент. Однако, если требуется доступ всего лишь к нескольким элементам очереди, платит, чтобы получить цену было не так дорого. По сути, в таком случае для операции копирования было бы пропорционально стоит стать равным, поскольку на умножение, которое копирует только указатели, и добавить в этот АПД операции, обеспечивающие доступ к элементам очереди без их копирования.

Утечка памяти — это трудно выявляемый дефект, который иногда может поразить даже самые лучшие системы. Хотя освобождение памяти, являющейся объектом, не было лишним делом, в принципе, прояснить на практике очень сложно быть уверенным в том, что удалось отыскать все места до последней распределенной памяти памяти. Механизм освобождения в языке C++ таков, однако система не может гарантировать, что объект структур данных освобожден так, как задумывалось. Книга объясняет, почему это происходит, его указатели являются неполными, и любой указатель, не являющийся «использованным» конструктором, автоматически ведет к утечке памяти. Одним из типовых источников утечки памяти является также книга в

классе вообще привыкли определять абстрактный — конечно же, не стоит забывать, что деструктор, выполняющий то же самое, каржиной должен быть. Особое внимание этому вопросу необходимо уделить в следующих, когда используются абстрактные классы, наглядные классы из программы 4.13.

В ряде случаев существует естественное представление данных — в таких случаях система самостоятельно определяет, какой объект данных больше не используется, и удаляется, после чего освобождая его. Но одно из двух решений не является полностью удовлетворительным. Если полагать, что распределение памяти — ошибочная задача, то тогда это можно было бы решить иначе: другое же считают, что это слишком важно, чтобы его можно было поручить системе, поэтому же считают, что это слишком важно, чтобы его можно было поручить программе.

Список задач, которые могут возникнуть при реализации реализаций АД, будет очень длинным даже в случае самых простых АД, даже те, которые рассматривались в этой главе. Нужно ли нам начинать подвизаться в одной из этих областей сразу после окончания главы? Или лучше ли нам в одной классической программе использовать разные реализации для объектов одного и того же типа, если известно, что они характеризуются разными свойствами? Следует ли вычислять — и как? — информацию об эффективности реализации? Какую форму должны иметь эти данные? Необходимость ряда задач подчеркивается, поскольку можно видеть различия в основных характеристиках алгоритмов и структур данных, и понимать, каким образом классические программы могут эффективно их использовать. В заключение следует отметить, что эта глава является частью книги. Хотя многие реализации являются простыми, но они выражены в тех же программах, и не только программы, но и алгоритмы. Мы стремимся не забывать об этих вещах, мы не хотим с тем, чтобы разработанные алгоритмы и структуры данных могли послужить основой для создания высококачественных программных средств и других высококачественных программ (или, наоборот, ошибок).

Упражнения

- 4.63 Измените операции $+$ и $-$ для работы с комплексными числами (программы 4.13 и 4.19).
- 4.64. Преобразуйте АД 'Степеньная зависимость' из раздела 4.7 в таблицу данных первого класса.
- 4.65 Создайте АД первого класса для многоугольников в плоскости, описываемых с помощью карт.
- 4.66 Используя АД из упражнения 4.65, напишите программу, которая автоматически определит принадлежность ряда различных наборов карт при игре в покер.
- 4.67 Разработайте реализацию для АД 'Комплексное число' на базе представления комплексных чисел в полярных координатах (r, θ) в форме $re^{i\theta}$.
- 4.68 Числитель многочлена $P' = az^N + \dots + 1$ для доказательства теоремы, что $e^{i\theta} = 1$, а P' комплексных корней N -й степени из единицы равен

$$\cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right)$$

$$\text{где } k = 0, 1, \dots, N-1$$

- 4.69 Перечислите корни N -ой степени из единицы для значений N от 2 до 8.
- 4.70 С использованием рекурсии и стека на файле `integers.txt` создайте реализацию перегруженной операции `**` для программы 4.19, которая выводит выходные данные, показанные на рис. 4.13 для программы 4.17.
- 4.71 Опишите точно, что произошло в результате запуска программы 4.20 модифицированной операцией, вопреки такой простой реализации, как программы 4.14 или 4.15.
- 4.72 Разработайте реализацию данного и текстового АТД первого класса 'Очередь ПГО' (программа 4.21), в которой в качестве базовой структуры данных используется массив.
- 4.73 Напишите интерфейс `ADT` первого класса для стека.
- 4.74 Разработайте реализацию АТД первого класса для стека на упражнении 4.73, который в качестве базовой структуры данных использует массив.
- 4.75 Разработайте реализацию АТД первого класса для стека на упражнении 4.73, который в качестве базовой структуры данных использует связный список.
- 4.76 Используя примененные ранее АТД первого класса для комплексных чисел (программы 4.18 и 4.19), модифицируйте программу вычисления коэффициентов выражений из раздела 4.1 таким образом, чтобы она выполняла постфиксное выражение, состоящее из комплексных чисел с целыми коэффициентами. Для простоты считайте, что все комплексные числа имеют целочисленные целые коэффициенты как для вещественной, так и для мнимой частей, и записываются без пробелов. Например, для следующего выражения
- $$1 + 2i + 3i + 1 - 2i + 3 + 4i + 1$$
- программа должна вывести результат $8 + 4i$.
- 4.77 Выпишите минимальный набор процессов модифицированной очереди в программе 4.21, чтобы определить переставку (как фактического аргумента N и M), того, что очередь, выбранная для N -ой операции `del`, будет пустой, а также описать количество элементов в этой очереди после N -й операции `del` или `ins`.

4.9 Пример использования АТД в приложении

В качестве иллюстративного примера рассмотрим модифицированный АТД, который характеризует отношения между областями применения и типами алгоритмов и структур данных, которые обсуждались в настоящей главе. Этот пример АТД, представляющего леммы, построен на области применениях леммы, в которой компоненты леммы описываются с абстрактными математическими объектами. Цель заключается в том, чтобы получить возможность писать программы, которые могут оперировать с леммами и выполнять вычисления над ними.

$$\left(1 - \left(1 + \frac{x^2}{2} - \frac{x^3}{6}\right)\right) - x - x^2 + x^3 = 1 + \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{3} + \frac{x^5}{3} - \frac{x^6}{6}.$$

Кроме того, необходима возможность конкатенации заданного значения x . Для $x = 0.5$ обе стороны приведенного выше уравнения имеют значение 1.1325125. Операции умножения, сложения и вычитания полиномов являются децентрализованными операциями в огромном числе математических вычислений. Программа 4.23 — это простой пример, в котором выполняются основные операции, соответствующие полиномиальным уравнениям.

```
(x + 1)' = x' + 2x + 1,
(x - 1)' = x' + 4x' + 4x' + 4x + 1,
(x + 1)' = x' + 4x' + 6x' + 4x + 1,
(x + 1)' = x' + 5x' + 10x' + 10x' + 5x + 1.
```

Универсальные основы можно развить дальше и включить такие операции, как сложение, интегрирование, дифференцирование, операции с специальными функциями и т.д.

Программа 4.23 Клиентская программа для АТД "Полином" (бинарные коэффициенты)

Эта клиентская программа использует АТД "Полином", определенный в интерфейсе (программа 4.24), для выполнения алгебраических операций над полиномами с целыми коэффициентами. В программе из командной строки передается число N и число x , значение которых N и x используются для вычисления $(x + 1)^N$ и $(x - 1)^N$ и $(x + 1)^N + (x - 1)^N$ и выводятся результаты расчета полинома для $x = 0$.

```
#include <csapp.h>
#include <stdlib.h>
#include "poly.h"

int main(int argc, char *argv[])
{
    int k = atoi(argv[1]); //long y = atoi(argv[2]);
    const << "Binomial coefficients" << endl;
    int i; for (i = 0; i < k; i++) {
        y = y * k; const << y << endl; }
    const << y + 1 << endl;
}
```

Первый шаг — определить АТД "Полином" так, как это иллюстрируется интерфейсом на программе 4.24 для такой короткой гонимой метасимвольной абстракции, как полином. Поскольку мы только хотим, что бы не стоило выражать полином так же, как в случае АТД для комплексных чисел, который обсуждается в разделе 4.5; необходимо, чтобы абстрактный АТД себя вел точно так же, как и его конкретные реализации.

Для реализации функций, определенных в этом интерфейсе, требуется выбрать конкретную структуру данных для представления полиномов и затем реализовать соответствующие алгоритмы для работы с этой структурой данных, чтобы получить реализацию функциональности в соответствии с ожидаемыми канонической программой. Как обычно, набор структуры данных влияет на систематическую эффективность алгоритма, поэтому стоит подумать несколько полноты. Так же как для стеков и оче-


```

friend POLY operator*(POLY &p, POLY &q)
{
    POLY t(0, (p.m-1)+1, q.m-1);
    for (int i = 0; i < p.m; i++)
        for (int j = 0; j < q.m; j++)
            t.a[i+j] += p.a[i]*q.a[j];
    return t;
}
}

```

В ходе выполнения перегруженных операций $+$ и $*$ сохранились новые полиномы. Поэтому данная реализация связана с утечкой памяти. Утечку памяти можно легко ликвидировать, добавив в реализацию конструктор копирования, перегруженный оператор перемещения и деструктор. Так мы стали и поступить в случае полиномов очень больших размеров, обработки большого количества небольших полиномов, а также создали АТД для использования в каком-нибудь приложении (см. упр. 4.79).

Многочлены и реализации АТД "Полномы" представляют из себя многообразие — то, как обычно, лишь одна из возможностей. Если полиномы являются очень большими, а факторизация полиномов темного, то представление из списка связанного списка может оказаться более подходящим. Например, во главе 5 мы рассмотрим программу 4.25 для выполнения такого умножения:

$$(1 - x^{1000})(1 + x^{1000}) = 1 + x^{1000} + x^{2000} + x^{3000},$$

поскольку в ней будет использоваться многообразие многочленов с подмножеством подмножества показываемых коэффициентов. В упражнении 4.78 вариант со связным списком рассматривается более подробно.

Упражнения

- 4.78 Напишите реализацию для программы в тексте АТД "Полномы" (программа 4.24), в которой в качестве базисной структуры данных используются связанные списки. Списки не должны содержать нулей, 0x0-элементов не должно быть нулевых коэффициентов.
- 4.79 Устраните утечку памяти в программе 4.25, добавив в нее конструктор копирования, перегруженный оператор присваивания и деструктор.
- 4.80 Добавьте перегруженные операции $+$ и $*$ в АТД "Полномы" из программы 4.25.
- 4.81 Расширьте рассмотренный в главе АТД "Полномы", включив в него операции интегрирования и дифференцирования полиномов.
- 4.82 Модифицируйте полученный АТД "Полномы" из упражнения 4.81 так, чтобы в нем значащими были все члены с экспонентами, большими или равными какому числу M , которое поступает в качестве аргумента во время инициализации.
- 4.83 Расширьте АТД "Полномы" из упражнения 4.81 так, чтобы он включал деление и сложение полиномов.
- 4.84 Разработайте АТД, который позволяет эффективно производить вычисления сложения и умножения целых чисел произвольной точности.

- 4.85 Напишите АЛД, разработанный в соответствии с выражением 4.84, модифицирующий программу на исполнение postfixных выражений из раздела 4.3 так, чтобы эти выражения можно было записывать в виде, состоящем из целых чисел, прикладываясь к ним скобки.
- 4.86 Напишите алгоритмическую программу, которая с помощью АЛД "Поллиум" из выражения 4.85 выполняет интеграл, находит разложение функции в ряд Тейлора и оперирует с ними в аналитической форме.
- 4.87 Разработайте АЛД, который позволяет клиентским программам взаимодействовать с сервером для выполнения запросов на получение точной информации.
- 4.88 Разработайте АЛД, который позволяет клиентским программам выполнять операции над матрицами объектов. Эти объекты представляются матрицей сложения, умножения, умножения и деления.
- 4.89 Напишите интерфейс для АЛД "Сложная строка", который поддерживает операции добавления строк, сравнения двух строк, копирования двух строк, копирования одной строки в другую и получение длины строки. Другим интерфейсом должна быть реализован интерфейс, доступный в стандартной библиотеке C++.
- 4.90 Напишите реализацию для интерфейса из выражения 4.89, используя тот, где это необходимо стандартную библиотеку C++.
- 4.91 Напишите реализацию для полученного в выражении 4.89 интерфейса строки. Напишите программу, которая на основе списка. Пропустите каждую из операций, для каждой из них.
- 4.92 Напишите интерфейс и реализацию АЛД "Множество объектов", в котором объектами являются множества, операции над множествами: U до $M - 1$ (где M — заданная константа) и операции над множествами: объединение двух множеств, пересечение двух множеств, дополнение множества, разность двух множеств и выходящее из множества. Для представления каждого множества используйте массив из $M - 1$ элементов, принимающих значения 0–1.
- 4.93 Напишите алгоритмическую программу, которая реализует АЛД, описанный в выражении 4.92.

4.10 Перспективы

Приступая к изучению алгоритмов и структур данных, следует учитывать также фундаментальные понятия, лежащие в основе АЛД. Рассмотрим три главных понятия:

- АЛД являются важными и широким набором инструментов, которые используются для разработки программного обеспечения, и многие из изученных структур служат реализацией широкого круга приложений для различных АЛД.
- АЛД позволяют эффективно реализовать алгоритмы, чтобы они не только выполнялись, но и служили для них самым разным способом.
- АЛД предоставляют собой удобный механизм, который используется в широком диапазоне приложений и структур данных, связанных с их производительностью.

Судя по АТД можно предполагать, для создания другого. Абстрактные подходы указывают на структуру, сформированную в языке C++, разработанную для построения связанных списков. Далее абстракция сводится к выводу и выводу, доступных в C++, для выявления при построении стеков многоуровневого типа. Наконец, стек многоуровневого типа используется для формирования «выделенной» структурно-структурной. Плановые АТД позволяют создавать большие вычисления основанных уровней абстракции, от существующих в компьютере машинных инструкций, до разобраных возможностей языка программирования, вплоть до структуры, планов и другой функциональности высшего уровня, которая обеспечивается авторскими (см. главы 3 и 4). Некоторые предложения требуют еще более высокого уровня абстракции, по которому стек обрывается к числам 0, 1 до 8. Абстрактные типы данных — это лишь язык, который обеспечивает создание все более и более мелких абстрактных моделей, в том же направлении есть эффективное использование компьютеров для решения современных задач.

Рекурсия и деревья

Рекурсия относится к одному из фундаментальных понятий в математике и в компьютерных науках. В языках программирования рекурсивной программой называют программу, которая обращается к самой себе (подобно тому, как в математике рекурсивной функцией называют функцию, которая определена через понятие самой этой функции). Рекурсивная программа не может вызывать себя до бесконечности, поскольку в этом случае она никогда не завершится. Но (точно так же рекурсивная функция не может всегда определяться понятием самой функции, поскольку тогда непременно стали бы циклическими). Следовательно, чтобы доказать корректность рекурсивной программы — наличие нужных утверждений, позволяющих программе прекратить вызывать себя — приходится обращаться к математике. Это удобно, для выполнения которого функция нередко определяется логическим образом этой функцией. Все практические вычисления можно представлять рекурсивными структурами.

Изучение рекурсии переплывается с изучением рекурсивно определенных структур, называемых деревьями. Деревья используются как для упрощения понимания и анализа рекурсивных программ, так и в качестве данных структур данных. В главе 1 мы уже встречались с применением деревьев (данных не рекурсивных). Различия между рекурсивными программами и деревьями лежат в основе значительной части материала книги. Деревья используются для упрощения понимания рекурсивных программ, в свою очередь, рекурсивные программы используются для построения деревьев: в конечном итоге, глобальная картина связей между ними и рекурсивные отношения) применяется при анализе алгоритмов. Рекурсия

пожелает разработывать нашиные и эффективные структуры данных и алгоритмы для достижения этого спектра применимости.

Основная цель этой главы заключается в исследовании рекурсивных программ и структур данных как практических инструментов. Вначале исследуется тесная связь между математической рекурсией и простыми рекурсивными программами к примеру ряд примеров применения рекурсии. Затем рассматривается фундаментальные рекурсивная схема, известная под названием "раздел и владей", которая используется для решения задач общего характера в нескольких последующих разделах этой книги. После этого применяется общий подход к реализации рекурсивных программ, называемый "шаблонными абстракциями", который предоставляет эффективные и эффективные решения для обширного класса задач. Затем подробно рассматриваются деревья, их математические свойства и связанные с ними алгоритмы, в том числе различные методы обхода дерева (как рекурсивно), которые лежат в основе рекурсивных программ обработки деревьев. В завершение будут аннотироваться тесно связанные с рекурсивной алгоритмы обработки графов - в частности, особое внимание будет уделяться фундаментальным рекурсивным программам поиска и обхода графов-дерево-графов, которая является основой для многих алгоритмов обработки графов.

Как будет показано далее, многие интересные алгоритмы легко и просто реализовать через рекурсивные программы, а многие разработчики алгоритмов предпочитают выражать методы рекурсивно. Кроме того, во многих подробностях будут исследоваться и не рекурсивные методы реализации. Часто можно не только создать простые алгоритмы с использованием циклов, которые, в основном, эквивалентны рекурсивным, но и попытаться перекурсивные альтернативы, которые приводят к такому же конечному результату через иную последовательность вычислений. Рекурсивная формулировка проблемы предоставляет структуру, в рамках которой достигаются более эффективные альтернативы.

Исчерпывающее исследование рекурсии и деревьев могло бы послужить темой отдельной книги, поскольку эти концепции находят применение во множестве компьютерных приложений, а также находят место на рынке компьютерных наук. Фактически можно было бы сказать, что эта книга является основной основой рекурсии и деревьев, поскольку с фундаментальной точки зрения упомянутые темы затрагиваются в каждой главе.

5.1 Рекурсивные алгоритмы

Рекурсивный алгоритм — это алгоритм, решающий задачу путем решения одного или нескольких более узких вариантов той же задачи. Для реализации рекурсивных алгоритмов в C++ используются рекурсивные функции — функции, которые вызывают сами себя. Рекурсивные функции в C++ соответствуют рекурсивным определениям математических функций. Многие рекурсии связаны с математическими примерами, которые непосредственно вычисляют математические функции. Как будет показано, большие механизмы можно расширить, что приводит к обобщенной парадигме программирования.

Программа 5.1 Функция вычисления факториала (рекурсивная реализация)

Эта рекурсивная функция вычисляет факториал $N!$ не только с помощью рекурсивного определения. Она позволяет проверить правильность значения, когда выполняется в нестрогостях и достаточно малым значением N , чтобы $N!$ можно было представить типом `int`.

```
int factRec1(int N)
{
    if (N == 0) return 1;
    return N * factRec1(N-1);
}
```

Рекурсивные отношения (см. раздел 4.3) являются рекурсивно определенными функциями. Рекурсивное отношение между функциями, областью допустимых значений которой являются натурательные числа, определяемые либо некоторыми начальными значениями, либо (рекурсивно) множеством ее собственных значений, полученных из одного изначального значения. Вероятно, наиболее известной из таких функций является функция вычисления факториала, которая определяется рекурсивным отношением:

$$N! = N \cdot (N-1)! \quad \text{для } N \geq 1, \text{ причем } 0! = 1$$

Что определение непосредственно соответствует рекурсивной функции `fact` в программе 5.1.

Программа 5.1 абсолютно проста для цикла. Например, следующая программа вычисляет факториал $N!$ с помощью цикла:

```
int fact1(int N)
{
    int i = 1, f = 1;
    while (i <= N) f *= i++;
    return f;
}
```

Как было показано, рекурсивную программу всегда можно преобразовать в не-рекурсивную, которая выполнит такое же вычисление. И наоборот: используя рекурсию, можно избежать, предполагая, что выполнение циклов, можно избежать, и избежать циклов.

Рекурсия используется потому, что зачастую она позволяет писать сложные алгоритмы в компактной форме без ущерба для эффективности. На пример, рекурсивная реализация функции вычисления факториала избавляет от необходимости использования локальных переменных. В системах программирования, поддерживающих обращение к функциям, поддержка рекурсивной реализации определяется возможностями, которые используют эквивалент экстремного случая. Большинство современных систем программирования имеют тщательно разработанные механизмы для выполнения такой задачи. Как будет показано, несмотря на это, применение очень легко можно создать рекурсивную функцию, которая окажется весьма эффективной, и поэтому необходимо стараться, чтобы последствия не привели к потере эффективности и эффективности эквивалентных реализаций.

Программа 5.2 Сложная рекурсивная программа

Если аргумент N является неотрицательным, эта функция вычисляет сумму всех чисел от 0 до N включительно. Если N является отрицательным, функция вычисляет сумму всех чисел от 0 до $N/2$ включительно. Для гарантированного завершения этой программы нужно использовать индукцию, поскольку на каждый рекурсивный вызов передается аргумент меньше заданного.


```

int puzzle(int N)
{
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}

```

Пространства 5.1 иллюстрирует базисные возможности рекурсивной программы: она может вызвать себя (или другую программу) для решения временных задач и содержит условие завершения, при котором завершается непосредственно процесс выполнения программы; иначе говоря, чтобы избежать бесконечности работы программы, нужно прописать условие завершения каждой функции.

- Программа вычисляет $\bar{U}$ (наименьшее значение)
- Если допустить, что программа вычисляет $\bar{U}$ для $\bar{\epsilon} \in \bar{A}$ (индуктивные предположения), то она вычисляет и $\bar{A}'$.

Таким же образом можно прийти к быстрому завершению разработки алгоритма для решения поставленной задачи.

В языках программирования, подобных С++, существует одна из наиболее интересных для педагогов особенностей программы. Это, как правило, использование рекурсивных функций применительно к тем, которые позволяют проводить транзитивную оценку работы методов математической индукции, как «показательная» была. Хотя в этой книге вопрос форматах подтверждения применимости не рассматривается, но интересно отметить, что сложная программа для выполнения сложных задач, а также требуется определенная гарантия, что она будет решена корректно. Методами, подобными рекурсивным функциям, могут обеспечить такую гарантию, в то же время обеспечивая компактные решения. В частности, для создания правил математической индукции, необходимо удостовериться, что создаваемые рекурсивные функции удовлетворяют двум основным условиям:

- Они должны быть решены за $O(1)$ для каждого значения.
- В каждом рекурсивном вызове должны использоваться только значения $i-1$ и $i-2$.

Для устранения двоякой спорности — она двоякая — пока, что мы должны рассмотреть допустимый индуктивный примерный для каждой составленной рекурсивной функции. Тем не менее, они служат параллельным руководством при разработке реализаций.

Пример 5.2 — известный пример, иллюстрирующий необходимость наличия хотя бы одного аргумента, ссылающегося на себя рекурсивно (функция, вызывающая себя).

```

puzz = 0
    puzzle(10)
        puzzle(5)
            puzzle(10)
                puzzle(8)
                    puzz = 4
                        puzzle(2)
                            puzz = 5

```

РИСУНОК 5.1 ПРИМЕР ЦЕПочки РЕКУРСИВНЫХ ВЫЗОВОВ

Սույն փոփոխություններն արտահայտվում են հետևյալ ձևով՝

Հոդ	Հոդի նկարագրություն	Հոդի փոփոխություն
1	Հոդի 1-ին մասը	Հոդի 1-ին մասը
2	Հոդի 2-րդ մասը	Հոդի 2-րդ մասը
3	Հոդի 3-րդ մասը	Հոդի 3-րդ մասը
4	Հոդի 4-րդ մասը	Հոդի 4-րդ մասը
5	Հոդի 5-րդ մասը	Հոդի 5-րդ մասը
6	Հոդի 6-րդ մասը	Հոդի 6-րդ մասը
7	Հոդի 7-րդ մասը	Հոդի 7-րդ մասը
8	Հոդի 8-րդ մասը	Հոդի 8-րդ մասը
9	Հոդի 9-րդ մասը	Հոդի 9-րդ մասը
10	Հոդի 10-րդ մասը	Հոդի 10-րդ մասը
11	Հոդի 11-րդ մասը	Հոդի 11-րդ մասը
12	Հոդի 12-րդ մասը	Հոդի 12-րդ մասը
13	Հոդի 13-րդ մասը	Հոդի 13-րդ մասը
14	Հոդի 14-րդ մասը	Հոդի 14-րդ մասը
15	Հոդի 15-րդ մասը	Հոդի 15-րդ մասը
16	Հոդի 16-րդ մասը	Հոդի 16-րդ մասը
17	Հոդի 17-րդ մասը	Հոդի 17-րդ մասը
18	Հոդի 18-րդ մասը	Հոդի 18-րդ մասը
19	Հոդի 19-րդ մասը	Հոդի 19-րդ մասը
20	Հոդի 20-րդ մասը	Հոդի 20-րդ մասը
21	Հոդի 21-րդ մասը	Հոդի 21-րդ մասը
22	Հոդի 22-րդ մասը	Հոդի 22-րդ մասը
23	Հոդի 23-րդ մասը	Հոդի 23-րդ մասը
24	Հոդի 24-րդ մասը	Հոդի 24-րդ մասը
25	Հոդի 25-րդ մասը	Հոդի 25-րդ մասը
26	Հոդի 26-րդ մասը	Հոդի 26-րդ մասը
27	Հոդի 27-րդ մասը	Հոդի 27-րդ մասը
28	Հոդի 28-րդ մասը	Հոդի 28-րդ մասը
29	Հոդի 29-րդ մասը	Հոդի 29-րդ մասը
30	Հոդի 30-րդ մասը	Հոդի 30-րդ մասը
31	Հոդի 31-րդ մասը	Հոդի 31-րդ մասը
32	Հոդի 32-րդ մասը	Հոդի 32-րդ մասը
33	Հոդի 33-րդ մասը	Հոդի 33-րդ մասը
34	Հոդի 34-րդ մասը	Հոդի 34-րդ մասը
35	Հոդի 35-րդ մասը	Հոդի 35-րդ մասը
36	Հոդի 36-րդ մասը	Հոդի 36-րդ մասը
37	Հոդի 37-րդ մասը	Հոդի 37-րդ մասը
38	Հոդի 38-րդ մասը	Հոդի 38-րդ մասը
39	Հոդի 39-րդ մասը	Հոդի 39-րդ մասը
40	Հոդի 40-րդ մասը	Հոդի 40-րդ մասը
41	Հոդի 41-րդ մասը	Հոդի 41-րդ մասը
42	Հոդի 42-րդ մասը	Հոդի 42-րդ մասը
43	Հոդի 43-րդ մասը	Հոդի 43-րդ մասը
44	Հոդի 44-րդ մասը	Հոդի 44-րդ մասը
45	Հոդի 45-րդ մասը	Հոդի 45-րդ մասը
46	Հոդի 46-րդ մասը	Հոդի 46-րդ մասը
47	Հոդի 47-րդ մասը	Հոդի 47-րդ մասը
48	Հոդի 48-րդ մասը	Հոդի 48-րդ մասը
49	Հոդի 49-րդ մասը	Հոդի 49-րդ մասը
50	Հոդի 50-րդ մասը	Հոդի 50-րդ մասը
51	Հոդի 51-րդ մասը	Հոդի 51-րդ մասը
52	Հոդի 52-րդ մասը	Հոդի 52-րդ մասը
53	Հոդի 53-րդ մասը	Հոդի 53-րդ մասը
54	Հոդի 54-րդ մասը	Հոդի 54-րդ մասը
55	Հոդի 55-րդ մասը	Հոդի 55-րդ մասը
56	Հոդի 56-րդ մասը	Հոդի 56-րդ մասը
57	Հոդի 57-րդ մասը	Հոդի 57-րդ մասը
58	Հոդի 58-րդ մասը	Հոդի 58-րդ մասը
59	Հոդի 59-րդ մասը	Հոդի 59-րդ մասը
60	Հոդի 60-րդ մասը	Հոդի 60-րդ մասը
61	Հոդի 61-րդ մասը	Հոդի 61-րդ մասը
62	Հոդի 62-րդ մասը	Հոդի 62-րդ մասը
63	Հոդի 63-րդ մասը	Հոդի 63-րդ մասը
64	Հոդի 64-րդ մասը	Հոդի 64-րդ մասը
65	Հոդի 65-րդ մասը	Հոդի 65-րդ մասը
66	Հոդի 66-րդ մասը	Հոդի 66-րդ մասը
67	Հոդի 67-րդ մասը	Հոդի 67-րդ մասը
68	Հոդի 68-րդ մասը	Հոդի 68-րդ մասը
69	Հոդի 69-րդ մասը	Հոդի 69-րդ մասը
70	Հոդի 70-րդ մասը	Հոդի 70-րդ մասը
71	Հոդի 71-րդ մասը	Հոդի 71-րդ մասը
72	Հոդի 72-րդ մասը	Հոդի 72-րդ մասը
73	Հոդի 73-րդ մասը	Հոդի 73-րդ մասը
74	Հոդի 74-րդ մասը	Հոդի 74-րդ մասը
75	Հոդի 75-րդ մասը	Հոդի 75-րդ մասը
76	Հոդի 76-րդ մասը	Հոդի 76-րդ մասը
77	Հոդի 77-րդ մասը	Հոդի 77-րդ մասը
78	Հոդի 78-րդ մասը	Հոդի 78-րդ մասը
79	Հոդի 79-րդ մասը	Հոդի 79-րդ մասը
80	Հոդի 80-րդ մասը	Հոդի 80-րդ մասը
81	Հոդի 81-րդ մասը	Հոդի 81-րդ մասը
82	Հոդի 82-րդ մասը	Հոդի 82-րդ մասը
83	Հոդի 83-րդ մասը	Հոդի 83-րդ մասը
84	Հոդի 84-րդ մասը	Հոդի 84-րդ մասը
85	Հոդի 85-րդ մասը	Հոդի 85-րդ մասը
86	Հոդի 86-րդ մասը	Հոդ

прямо по количеству с которыми каждый рекурсивный элемент должен использовать меньше значений элементов, и потому для ее представления можно использовать метод математической индукции. Действительно, известно, завершится ли эта операция для каждого значения N . Поскольку значение N не имеет никаких пределов. Для больших численных значений, которые могут быть представлены типом `long`, можно проверить, что программа прерывалась (см. рис. 5.1 и упражнение 5.1), но для больших численных значений (скажем, для 64-разрядных слов), не известно, исполняла ли эта программа в какой-нибудь цикл.

Примеры 5.3 — конкретные реализации алгоритма Тьюинга для вычисления приближенного значения π для двух целых чисел. Алгоритмы основаны на наблюдении, что наибольший общий делитель двух положительных x и y совпадает с минимальным значением $ax + by$ по модулю 1 (связка от взаимных x и y). Число π можно вычислить, используя тот же алгоритм, когда оно делит π и x по модулю y , поскольку x равно x плюс целое число, кратное y . Впервые это было отмечено в примере 5.2. При реализации алгоритма Тьюинга глубина рекурсии зависит от арифметических свойств используемых значений x и y и от логарифмической аппроксимации.

Программа 5.3 Авторный Зыклов

В соответствии с требованиями к документам, подлежащим хранению в 2000 г. в архиве Российской Федерации метод отчисления наибольшего долгователя (группы должников)

```

10-  gnd(100000, 200000)
11-
12-      it / 0 == 0) return 0;
13-      return gnd(100000, 200000);
14-  }

```

Программа 5.4 — пример с несколькими рекурсивными вызовами. Она отличается от предыдущей тем, что существует возможность тех же изменений, что и в программе 4.1, но применительно к профизмам (а не к сиффизмам) выражения, находясь в состоянии внешнего стека. В этой главе будут приведены множество других примеров использования рекурсивных функций. Это программ, в которых используются внешние стеки. Конкретная взаимосвязь между несколькими парами таких программ следует из следующего обсуждения.

Программа 5.4 Рекурсивная программа для оценки префиксных выражений

Для цэтай пазначка-сц вышэйжо не лічыюся адпаведальнасцю за свае дзеянні і ўдзяку ўсім тым, хто дапамог мне ў гэтым. Я ўдзякую вам за ўсё, што вы зрабілі для мяне і для ўсёй Беларусі.

выражения. Эта функция является рекурсивной, однако рекурсия приносит в данном случае дополнительные трудности и только усложняет понимание программы. Индекс изменяется после вычисления каждого подвыражения:

```
char *a; int i;
int eval()
{
    int k = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        i++; return eval() + eval();
    if (a[i] == '-')
        i++; return eval() - eval();
    while ((a[i] != '0') && (a[i] != '9'))
        k = 10*k + (a[i++] - '0');
    return k;
}
```

Пример обработки префиксного выражения программой 5.4 показан на рис. 5.3. Множество рекурсивных вызовов маскирует сложную серию вычислений. Подобно детерминисту рекурсивных программ, работу этой программы проще всего понять, воспользовавшись методом следствия. Именно так она работает применительно к простым выражениям, можно предположить, что эта программа применима к сложным выражениям. Программа представляет собой простой пример рекурсивного алгоритма вычисления — для начальной стадии используется для преобразования программы C++ в машинный код.

Реальная проверка правильности оценки выражения программой 5.4 — задача более сложная задача. Вызов процедуры работы операторов функций с различными аргументами, а в этой книге нет места еще более сложным рекурсивным программам и структурам данных. Следовательно, мы не ставим перед собой математической задачи доказательства полноты индуктивной логичности при выводе каждой возникающей рекурсивной программы. В данном случае способность программы "угадать", как следует разделить операции и сопоставить с указанным оператором, иногда кажется гениальной (хотя можно подумать, что мы сразу поймем, как реализовать это задание по первому уроку). Однако в действительности вычисление достаточно просто (пока что) нужно лишь прогнать при каждом вызове функцию оценивания (определяется первым символом выражения).

В принципе, любой язык Fort можно заменить эквивалентной рекурсивной программой. Часто ре-

```
eval() = + 7 = + 4 2 + 8 9 6
eval() = 7 + + 4 8 + 0 9
eval() 7
eval() = + 4 6 + 3 9
eval() = + 6
eval() 6
eval() 6
return 24 = 4*6
eval() = 8 9
eval() 8
eval() 9
return 17 = 8 + 9
return 408 = 24*17
return 455 = 7*408
eval() 6
return 276 = 455*6
```

РИСУНОК 5.3 ПРИМЕР ОЦЕНКИ ПРЕФИКСНОГО ВЫРАЖЕНИЯ

Для вычисления префиксного выражения функция оценивания работает рекурсивно, обрабатывая выражения слева направо. Для каждого оператора вычисления для данного выражения вычисляется значение выражения. Если оператор является бинарным, то для вычисления значения выражения необходимо вычислить значения обоих операндов. Если оператор унарный, то для вычисления значения выражения необходимо вычислить значение одного операнда. Если оператор является константой, то для вычисления значения выражения необходимо вычислить значение константы.

курсивная программа предоставляет более естественный способ выражения значимости, чем цикл `for`, поэтому можно попытаться преодолеть недостатки механизмов предоставления систем, поддерживающих рекурсию. Однако, это не всегда имеет единственный недостаток, о котором следует помнить. Как должно быть понятно из примеров, приведенных на рис. 5.1–5.3, при выполнении рекурсивной программы таковые функции выполняются один за другим, так и не достигнута точка, когда вместо рекурсивного вызова выполнялся возврат. В большинстве сред программирования такие промежуточные этапы функций реализуются с помощью записей в устройстве стеков. Сущность подобного рода реализации будет рассмотрена на протяжении данной главы. Будем опасаться, что максимальная степень сложности вызова функций в ходе выполнения. В общем случае глубина будет зависеть от используемых данных. Например, глубина рекурсии в примере, приведенном на рис. 5.3 и 5.4, составляет соответственно 9 и 4. При использовании рекурсивной программы важно помнить, что среда программирования должна поддерживать стек, размер которого пропорционален глубине рекурсии. При решении сложных задач необходимо для этого стека иметь память может оказаться от использования рекурсивного решения.

Структура данных, построенная на основе указателей, рекурсивна по своей природе. Например, список узлов связанной структуры в главе 3 (раздел 3.1) является рекурсивным. Следовательно, рекурсивные программы предоставляют эффективные механизмы для написания часто используемых функций, работающих с такими структурами данных. Программа 5.5 содержит четыре примера. Подобные реализации являются популярными частями в основном потому, что их гораздо проще понять, чем их не-рекурсивные аналоги. Однако, рекурсивные программы, подобные программе 5.5, следует использовать исключительно при обработке очень больших списков, поскольку глубина рекурсии может быть пропорциональна длине списка и, соответственно, требуемый для рекурсивного стека объем памяти может превзойти доступные пределы.

Программа 5.5 Примеры рекурсивных функций для связанного списка

Эти рекурсивные функции для выполнения простых задач обработки списков могут быть использованы для замены функций в главе 3, позволяющих выполнять обработку списка может быть пропорциональна длине списка.

Первая функция `count` подсчитывает количество узлов в списке. Вторая, `traverse`, выполняет функцию `visit` на каждом узле списка, с начала до конца. Обе функции легко реализуются с помощью цикла `for` или `while`. Третья функция `traverseR` не имеет ничего общего с предыдущими аналогами. Она выполняет функцию `visit` для каждого узла списка, но в обратном порядке.

Четвертая функция `length` удаляет из списка все узлы, не имеющие значения в узле. Последняя реализует эту функцию, но не с помощью цикла `for` или `while`, а с помощью рекурсии. Предполагается, что вы знаете, как шарф использовать функцию `visit` в списке. Каждая итерация цикла `while` создает стек вызовов, показанный на рис. 5.6, в данном случае `link` и `link->next` являются указателями.

```
int count(link x)
{
    if (x == 0) return 0;
    return 1 + count(x->next);
}
```


- 5.9 Создайте рекурсивную программу для оценки postfixных выражений.
- 5.10 Создайте рекурсивную программу для оценки infixных выражений. Можете считать, что операторы всегда заключаются в круглые скобки.
- 5.11 Создайте рекурсивную программу, которая преобразует infixные выражения в postfixные.
- 5.12 Создайте рекурсивную программу, которая преобразует postfixные выражения в infixные.
- 5.13 Создайте рекурсивную программу для решения задачи Паскаля Фибоначчи (Задача 1.3).
- 5.14 Создайте рекурсивную программу, которая удаляет последний узел из списка.
- 5.15 Создайте рекурсивную программу для удаления из обратного порядка связывания узлов в связном списке (см. программу 3.3). Используйте тройную переменную.

5.2 Разделяй и властвуй

Во множестве рассматриваемых в книге программ используется для рекурсивных выводов, каждый из которых работает приблизительно с основным элементом данных. Эта рекурсивная схема — вероятно, наиболее важный случай хорошо известного хода «разделяй и властвуй» разработки алгоритмов, который служит основой для множества алгоритмов.

Возьмем в качестве примера рассмотренную задачу отыскания максимума из N элементов, сохраненных в массиве $a[0], \dots, a[N-1]$. Для этого можно легко написать за один проход массива демонстрационный пример:

```
for (i = a[0], j = 1; j < N; j++)
    if (a[j] > i = a[j]);
```

Рекурсивное решение типа «разделяй и властвуй», приведенное в программе 5.6 — это один пример (хотя и совершенно иной) алгоритм решения той же задачи; он использовался только с целью иллюстрации концепции «разделяй и властвуй».

Программа 5.6 Применение алгоритма «разделяй и властвуй» для отыскания максимума

Эта функция находит максимум $a[l], \dots, a[r]$ из массива $a[l], \dots, a[r]$ и $a[m+1], \dots, a[r]$ (предполагается, что $a[l]$ — элемент, в обоих случаях рекурсивно) и возвращает больший из $a[l]$ и максимума массива $a[m+1], \dots, a[r]$. В программе предполагается, что $item$ — тот первый элемент, для которого операции $>$ определены. Если рассуждать о массиве как о части массива, оба случая имеют одинаковый смысл, если же четные и нечетные размеры различаются на 1.

```
long max11cmn a[], int l, int r
{
    if (l == r) return a[l];
    int m = (l+r)/2;
    long u = max11cmn(a, l, m);
    long v = max11cmn(a, m+1, r);
    if (u > v) return u; else return v;
}
```


Человек часто подчас 'размышляет и выискивает' решение задачи, что он объясняет, более эффективно, чем простые итерационные алгоритмы (в конце главы рассматриваются несколько примеров таких алгоритмов). Кроме того, данный подход заслуживает подробное и всестороннее, поскольку он способствует пониманию сути и специфики определенных видов вычислений.

На рис. 5.4 показаны рекурсивные вызовы, возникающие при запуске программы 5.6 применительно к примеру задачи. Структура вызовов кажется сложной, но обычно об этом можно не беспокоиться — для проверки программы мы воспользуемся методом математической индукции. В дальнейшем при необходимости используется рекурсивное соотношение.

Как обычно, сам код предполагает проверку правильности вычисления методом индукции:

- При $n = 0$ я немедленно возвращаю максимальный элемент массива a , размер которого равен L .
- Для $N > 0$ код разбивает массив на два, размер каждого из которых меньше N , исходя из индуктивного предположения, находит максимальные элементы в обеих частях и возвращает большее из этих двух значений, которое должно быть максимальным значением для всего массива.

Более того, рекурсивную структуру программы можно использовать для исследования характеристик ее производительности.

Лемма 5.1 Рекурсивная функция, которая разбивает массив размерности N на две меньшие (почти) равные части, рекурсивно вызывает себя менее N раз.

Если одна часть имеет размерность k , а другая — $N - k$, то общее количество рекурсивных вызовов некоторой функции равно

$$T_N = T_k + T_{N-k} + 1, \quad \text{при } N \geq 1, T_0 = 0$$

Решение $T_N = N + 1$ можно получить непосредственно методом индукции. Если суммарный размер частей меньше N , доказательство того, что количество вызовов меньше чем $N + 1$ вытекает из того же рассуждения по методу индукции. Аналогичные рассуждения можно повторить, спрашивая, справедливо ли данное утверждение для общего случая (см. упражнение 5.20).

0 1 2 3 4 5 6 7 8 9 10
1 1 1 1 1 1 1 1 1 1 1

```

Y max(0, 10)
  Y max(0, 5)
    Y max(0, 2)
      Y max(0, 1)
        Y max(0, 0)
          Z max(1, 1)
            Y max(0, 2)
              Z max(0, 5)
                Y max(2, 4)
                  Y max(0, 5)
                    E max(0, 4)
                      Z max(5, 5)
                        P max(5, 10)
                          P max(5, 0)
                            H max(6, 7)
                              A max(6, 6)
                                M max(7, 7)
                                  P max(8, 8)
                                    L max(9, 10)
                                      L max(9, 5)
                                        E max(10, 10)

```

РИСУНОК 5.4 РЕКУРСИВНЫЙ ПОДХОД К РЕШЕНИЮ ЗАДАЧИ ОТЫСКАНИЯ МАКСИМУМА

Для исследования сложности алгоритма можно использовать метод отыскания максимального значения в массиве.

Программа 5.6 — типичный пример для выбора алгоритмов типа 'разделяй и властвуй', поскольку содержит одноступенчатую рекурсивную структуру, но другие примеры могут отличаться из приведенного в двух аспектах. Во-первых, программа 5.6 не имеет обязательной оценки вычислений для каждой пары функций, в отличие от общего случая вычисления линейно связанного с количеством выходов. Как будет показано, другие алгоритмы типа 'разделяй и властвуй' могут выполнять разный объем вычислений для различных входов функций, и поэтому для определения объема времени выполнения требуется более сложный анализ. Время выполнения таких алгоритмов зависит от конкретного способа выделения их частей. Во-вторых, программа 5.6 — типичный пример алгоритмов типа 'разделяй и властвуй', для которых суммарный размер частей равен общей размерности. Другие алгоритмы типа 'разделяй и властвуй' могут разделить задачу на части, суммарная размерность которых меньше или больше размерности всей задачи. Для алгоритмов в качестве основы к рекурсивным алгоритмам типа 'разделяй и властвуй', поскольку каждая часть задачи меньше, но анализировать их труднее, нежели программу 5.6. Мы подробно рассмотрим примеры алгоритмов таких типов алгоритмов, как только сообразим

Например, алгоритм бинарного поиска, приведенный в разделе 2.6, является алгоритмом типа "разделяй и властвуй", который может применяться и к другим работам только в случае, если эти работы можно разделить на две части. Рекурсивная реализация бинарного поиска не описана в главе 12.

На рис. 5.5 показаны содержимое внутренних стека, содержимого стека при выполнении для реализации вычислений, изображенных на рис. 5.4. Извлеченная из рисунка модель является абстрагированной, но она позволяет взглянуть на структуру вычисления по методу 'разломай и выкачай' внутри. Если программа имеет вид рекурсивных вызовов, фактически внутренний стек содержит одну запись, соответствующую первому вызову функции во время исполнения (эта запись содержит значения аргументов, локальные переменные и адрес возврата), и локальную запись, соответствующую второму вызову функции во время ее выполнения. Аналогичным образом строится модель на рис. 5.5 снизу — появление в стек сразу двух записей: сохраняемая локальная переменная должна сохраняться в стек. Такая организация адресов в структуре вычисления и адресовывает основу

0	0								
1	1	1	1						
2	2	2	2	2	2				
3	3	3	3	3	3	3	3		
4	4	4	4	4	4	4	4	4	
5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9
10	10	10	10	10	10	10	10	10	10
11	11	11	11	11	11	11	11	11	11
12	12	12	12	12	12	12	12	12	12
13	13	13	13	13	13	13	13	13	13
14	14	14	14	14	14	14	14	14	14
15	15	15	15	15	15	15	15	15	15
16	16	16	16	16	16	16	16	16	16
17	17	17	17	17	17	17	17	17	17
18	18	18	18	18	18	18	18	18	18
19	19	19	19	19	19	19	19	19	19
20	20	20	20	20	20	20	20	20	20
21	21	21	21	21	21	21	21	21	21
22	22	22	22	22	22	22	22	22	22
23	23	23	23	23	23	23	23	23	23
24	24	24	24	24	24	24	24	24	24
25	25	25	25	25	25	25	25	25	25
26	26	26	26	26	26	26	26	26	26
27	27	27	27	27	27	27	27	27	27
28	28	28	28	28	28	28	28	28	28
29	29	29	29	29	29	29	29	29	29
30	30	30	30	30	30	30	30	30	30
31	31	31	31	31	31	31	31	31	31
32	32	32	32	32	32	32	32	32	32
33	33	33	33	33	33	33	33	33	33
34	34	34	34	34	34	34	34	34	34
35	35	35	35	35	35	35	35	35	35
36	36	36	36	36	36	36	36	36	36
37	37	37	37	37	37	37	37	37	37
38	38	38	38	38	38	38	38	38	38
39	39	39	39	39	39	39	39	39	39
40	40	40	40	40	40	40	40	40	40
41	41	41	41	41	41	41	41	41	41
42	42	42	42	42	42	42	42	42	42
43	43	43	43	43	43	43	43	43	43
44	44	44	44	44	44	44	44	44	44
45	45	45	45	45	45	45	45	45	45
46	46	46	46	46	46	46	46	46	46
47	47	47	47	47	47	47	47	47	47
48	48	48	48	48	48	48	48	48	48
49	49	49	49	49	49	49	49	49	49
50	50	50	50	50	50	50	50	50	50
51	51	51	51	51	51	51	51	51	51
52	52	52	52	52	52	52	52	52	52
53	53	53	53	53	53	53	53	53	53
54	54	54	54	54	54	54	54	54	54
55	55	55	55	55	55	55	55	55	55
56	56	56	56	56	56	56	56	56	56
57	57	57	57	57	57	57	57	57	57
58	58	58	58	58	58	58	58	58	58
59	59	59	59	59	59	59	59	59	59
60	60	60	60	60	60	60	60	60	60
61	61	61	61	61	61	61	61	61	61
62	62	62	62	62	62	62	62	62	62
63	63	63	63	63	63	63	63	63	63
64	64	64	64	64	64	64	64	64	64
65	65	65	65	65	65	65	65	65	65
66	66	66	66	66	66	66	66	66	66
67	67	67	67	67	67	67	67	67	67
68	68	68	68	68	68	68	68	68	68
69	69	69	69	69	69	69	69	69	69
70	70	70	70	70	70	70	70	70	70
71	71	71	71	71	71	71	71	71	71
72	72	72	72	72	72	72	72	72	72
73	73	73	73	73	73	73	73	73	73
74	74	74	74	74	74	74	74	74	74
75	75	75	75	75	75	75	75	75	75
76	76	76	76	76	76	76	76	76	76
77	77	77	77	77	77	77	77	77	77
78	78	78	78	78	78	78	78	78	78
79	79	79	79	79	79	79	79	79	79
80	80	80	80	80	80	80	80	80	80
81	81	81	81	81	81	81	81	81	81
82	82	82	82	82	82	82	82	82	82
83	83	83	83	83	83	83	83	83	83
84	84	84	84	84	84	84	84	84	84
85	85	85	85	85	85	85	85	85	85
86	86	86	86	86	86	86	86	86	86
87	87	87	87	87	87	87	87	87	87
88	88	88	88	88	88	88	88	88	88
89	89	89	89	89	89	89	89	89	89
90	90	90	90	90	90	90	90	90	90
91	91	91	91	91	91	91	91	91	91
92	92	92	92	92	92	92	92	92	92
93	93	93	93	93	93	93	93	93	93
94	94	94	94	94	94	94	94	94	94
95	95	95	95	95	95	95	95	95	95
96	96	96	96	96	96	96	96	96	96
97	97	97	97	97	97	97	97	97	97
98	98	98	98	98	98	98	98	98	98
99	99	99	99	99	99	99	99	99	99

**РИСУНОК 5.5 ПРИМЕР ДИНАМИКИ
ВНУТРЕННЕГО СТЕКА**

[illegible]

диск N на один стержень направо, после чего эти два перемещают $N - 1$ дисков на один стержень влево (поверх диска N). В правильности работы этого решения можно удостовериться методом индукции. На рис. 5.7 показаны перемещения для $N = 3$ и рекурсивные вызовы для $N = 3$. Структура алгоритма достаточно очевидна, давайте рассмотрим его подробно.

Программа 5.7 Решение задачи о ханойских башнях

Мы сводим задачу на движение дисков (рекурсивно) для диска, кроме нижнего, влево, затем двигаем нижний диск направо, после чего перемещаем (рекурсивно) башню вверх нижнего диска.

```
void hanoi(int n, int a)
{
    if (n == 0) return;
    hanoi(n-1, -a);
    shift(n, a);
    hanoi(n-1, -a);
}
```

По первым, рекурсивная структура этого решения немедленно диктует количество необходимых перемещений.

Лемма 5.2 Рекурсивная структура решения задачи о ханойских башнях имеет глубину, равную $2^N - 1$ перемещению.

Как обычно, мы пока неясно точно знаем, что количество перемещений удовлетворяет условию рекуррентности. А данным образом количество перемещений диска, удовлетворяющее условию рекуррентности, определяется формулой, иначе известной формуле 2.7:

$$T_N = 2T_{N-1} + 1, \quad \text{для } N \geq 2, T_1 = 1.$$

Предсказанный результат можно непосредственно проверить методом индукции: мы имеем $T(1) = 2^1 - 1 = 1$, и, если $T(k) = 2^k - 1$ для $k < N$, то $T(N) = 2(2^{N-1}) + 1 = 2^N - 1$.

Если можно перемещать по одному диску в секунду, то завершение работы не потребует, по меньшей мере, 256 столетий (см. рис. 2.1). Разумеется, если так не поступают олимпийцы. Скорее всего, закон света может наступить даже прежде, чем это случится. Поскольку мы никак не сможем бы воспользоваться программой 5.7 для быстрого завершения, какой диск нужно перемещать следующим. А теперь давайте опишем алгоритм, который сводит к простому (не рекурсивному) решению, решающему задачу о ханойских башнях. Хотя к не хочется выводить буквы алфавита в решении, однако этот метод имеет большое значение для понимания важности правильного анализа алгоритмов.

Чтобы понять решение задачи о ханойских башнях, давайте рассмотрим простую задачу рисования точек на линейке. Каждая 1/2 дюйма на линейке отмечена точкой, каждые 1/4 дюйма отмечены несколькими более короткими черточками, 1/8 дюйма — еще более короткими и т.д. Задача состоит в создании процедуры для рисования точек точек при заданном радиусе линии, при условии, что в заданном расположении имеется процедура `mark(x, h)` для рисования точек высотой h в условиях единицы в позиции x .

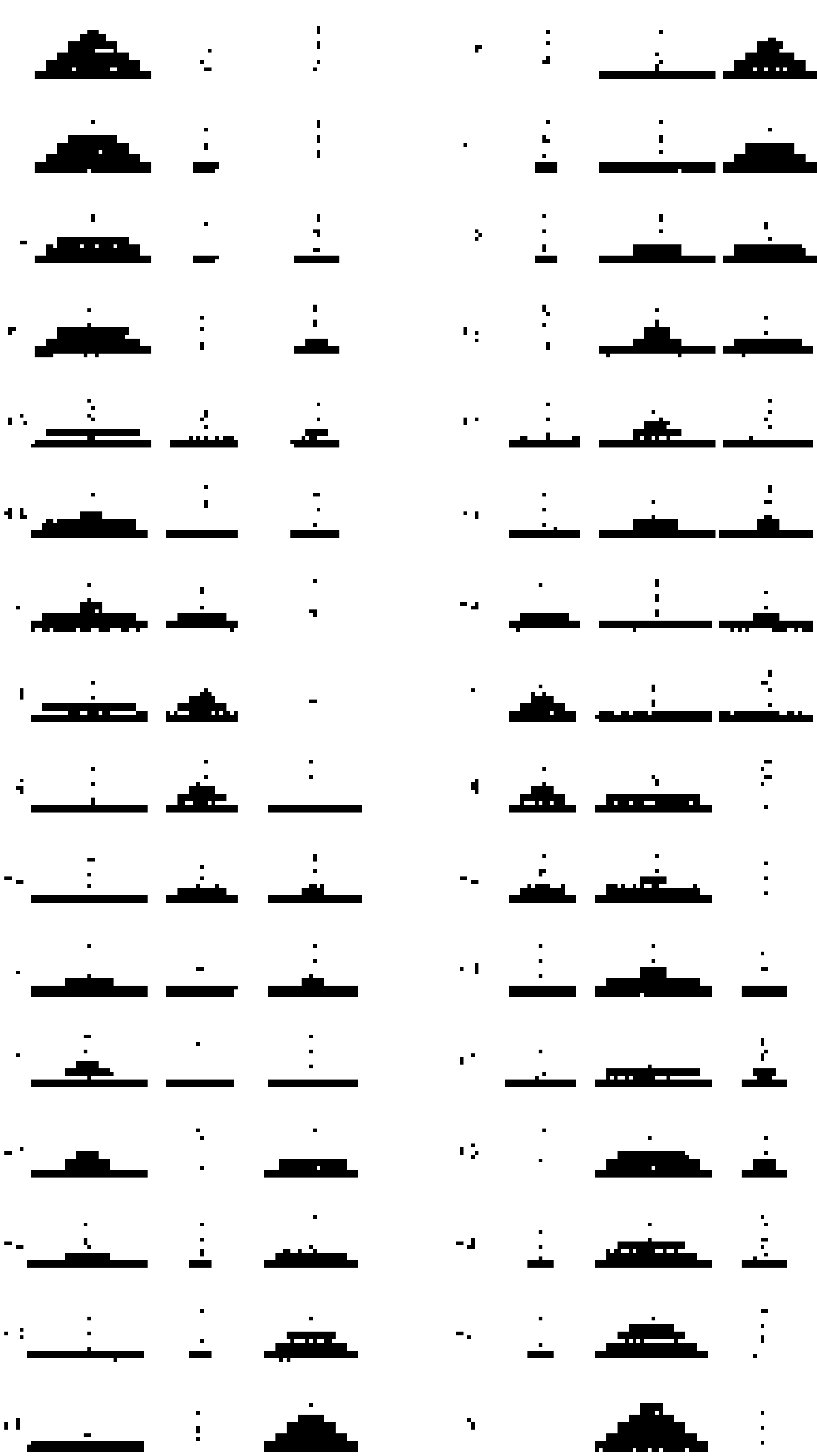


Рисунок 3.7. Криптограммы данных

На рисунке 3.7 представлены результаты работы алгоритма шифрования данных. Визуально можно увидеть, что исходные данные (исходные значения) имеют определенную структуру, которая сохраняется при шифровании. Это свидетельствует о том, что алгоритм шифрования является обратимым, то есть позволяет восстановить исходные данные из зашифрованных данных. Визуально можно увидеть, что исходные данные (исходные значения) имеют определенную структуру, которая сохраняется при шифровании. Это свидетельствует о том, что алгоритм шифрования является обратимым, то есть позволяет восстановить исходные данные из зашифрованных данных.

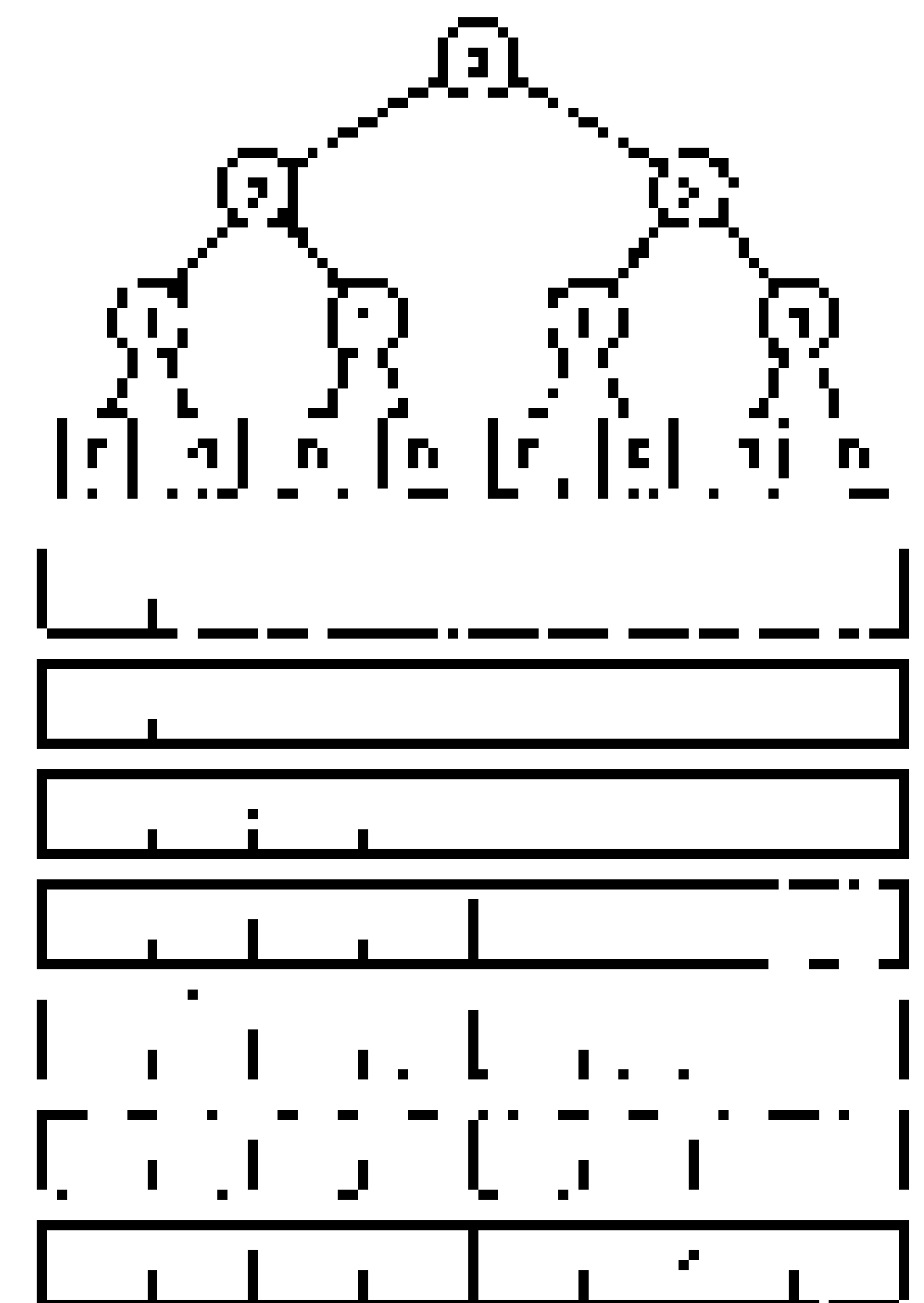
```
lans[i+3, +1]
lans[i+2, -1]
lans[i+1, +1]
lans[i+0, -1]
shift(1, +1)
lans[i+0, -1]
shift(2, -1)
lans[i+1, +1]
lans[i+2, -1]
shift(1, -1)
lans[i+2, -1]
shift(3, +1)
lans[i+1, -1]
shift(1, -1)
lans[i+0, -1]
shift(2, -1)
lans[i+1, +1]
lans[i+0, -1]
shift(1, +1)
lans[i+0, -1]
```


Если требуется разрешение ровно $1/2^n$ дюймов, давайте наметим масштаб, чтобы задача свелась к помещению точек в каждой точке в интервале от 0 до 2^n , за исключением конечных точек. Таким образом, каждая метка должна иметь высоту и ширину в единицах, метки в середине левой и правой половин должны иметь высоту $n - 1$ и ширину в единицах $n - 1$. Программа 5.8 — простой алгоритм "разделяй и властвуй" для выполнения этой задачи; работа данного алгоритма проиллюстрирована в иллюстрациях к рис. 5.8. С точки зрения рекурсии в основе метода лежит следующая идея. Для именования меток из интервала, заданного высотой h и шириной w , дадим такие параметры. Затем сложится метка (более короткая) в левой половине (рекурсивно), поместим длинную метку в середине и нарисуем метку (более короткую) в правой половине (рекурсивно). Если посмотреть на иллюстрацию, рис. 5.8 иллюстрирует, что с помощью этого метода метки складываются по порядку, слева направо — фокус заключается в вычислении лишь интервала. Терапия рекурсии, проиллюстрированная на рисунке, помещает точку в середине и проиллюстрирует это с помощью рис. 5.8, что длина меток уменьшается на 1 для каждого вызова рекурсивной функции. Если посмотреть на рис. 5.8 в поперечном направлении, мы получаем метки в том порядке, в каком они рисуются. Поскольку для каждого размера узла получаем рисунок метки, связанные с высотой функции h и шириной w , связанная с длиной узла, а затем метки, связанные с высотой функции h и шириной w .

Программа 5.8 Применение алгоритма "разделяй и властвуй" для рисования линейки

Для рисования линии n дюймов мы рисуем метку n дюймов в высоту, затем рисуем длинную метку в середине, а затем рисуем левую и правую половины. Это построение подготавливает для использования со значением i i равным степени 2 — свойства, описанное в 2.20 рекурсивным описанием (см. упражнение 5.27).

```
void rule(int l, int r, int h)
{ int m = (l+r)/2;
  if (h > 0)
  {
    rule(l, m, h-1);
    mark(m, h);
    rule(m, r, h-1);
  }
}
```



```
rule(0, 8, 3)
rule(0, 4, 2)
rule(0, 2, 1)
rule(0, 1, 0)
mark(1, 1)
rule(1, 2, 0)
mark(2, 2)
rule(2, 4, 1)
rule(2, 3, 0)
mark(3, 1)
rule(3, 4, 0)
mark(4, 0)
rule(4, 6, 2)
rule(4, 5, 1)
rule(4, 4, 0)
mark(5, 1)
rule(5, 6, 0)
mark(6, 0)
rule(6, 8, 1)
rule(6, 7, 0)
mark(7, 1)
rule(7, 8, 0)
```

РИСУНОК 5.8 ВЫЗОВЫ ФУНКЦИИ, РИСУЮЩИХ ЛИНЕЙКИ

Для рисования линейки мы рисуем длинную метку в середине, а затем рисуем левую и правую половины. Это построение подготавливает для использования со значением i i равным степени 2 — свойства, описанное в 2.20 рекурсивным описанием (см. упражнение 5.27).

Этот небольшой шаг отделяет рисующие линии от рисованных декартовых узоров, показанных на рис. 5.12. Этот рисунок иллюстрирует, как простое рекурсивное описание может привести к сложным декартовым (см. упражнение 5.30).

Рекурсивно описанные геометрические узоры, напоминающие показанный на рис. 5.12, являются частью более сложной традиции рисования и более сложных рекурсивных функций (таких, как такеловые рекурсивно определенные функции, работающие как с действительными значениями, так и в комплексной плоскости), которые разработаны поразительно разнообразными и сложными узорами. На рис. 5.13 приведен еще один пример — звезда Коха, которая определяется рекурсивно итерационным образом: звезда Коха 0 порядка — просто выступ, показанный на рис. 4.3, а звезда Коха n порядка — это звезда Коха порядка $n - 1$, в которой каждый линейный сегмент заменен соответствующим образом ориентированным рисунком 0 порядка.

Подобно решениям задач рисования линейных и хвойных башен, эти авторские линейные по отношению к количеству шагов, на эти количества связано экспоненциальной сложностью с максимальной глубиной рекурсии (см. упражнения 5.29 и 5.33). Кроме того, они могут быть непосредственно связаны с подсчетами в соответствующей системе счисления (см. упражнение 5.34).

Задача о хвойных башнях, рисование линейных и фрактальных рисунков является частью более широкой области, а связь с линейными числами показывает, однако все эти вопросы интересуют нас прежде всего потому, что обеспечивают тематическое единство изостратических разработок алгоритмов — деление задач на подзадачи и независимое решение одной или обеих подзадач; возможно, это — наиболее важный из поведенческих ролей технологий, рассматриваемых в книге. В табл. 5.1 подробно описаны бинарный поиск и сортировка слиянием, которые не только являются важными и широко используемыми на практике алгоритмами, но и служат типичными примерами разработки алгоритмов типа «разделяй и властвуй».

Таблица 5.1 Основные алгоритмы типа «разделяй и властвуй»

Бинарный поиск (гл. главы 2 и 17) и сортировка слиянием (см. главу 5) — алгоритмы типа «разделяй и властвуй», которые обеспечивают гарантированно оптимальную производительность соответственно, поскольку в бинарном поиске рассматриваемое пространство делится на две части, а сортировка слиянием делится на две части, а сортировка слиянием делится на две части.



РИСУНОК 5.12 ДВУМЕРНАЯ ФРАКТАЛЬНАЯ ЗВЕЗДА
Этот фрактал
состоит из 10 000
сегментов, каждый из
которых имеет
длину 1/1000
единицы измерения

Упражнения

- 5.16 Создайте рекурсивную функцию, которая возвращает максимальный элемент массива, типичная рекурсивная версия массива с максимальным элементом оставшейся части массива (пятиэлементный рекурсивный).
- 5.17 Создайте рекурсивную функцию, которая возвращает максимальный элемент в списке списков.
- 5.18 Модифицируйте программу 'разделяй и властвуй' для отыскания максимального значения в массиве (проблема 5.6). Пусть n — размер массива A на две части, одна из которых имеет размер $k = 2^{k-1} - 1$, а другая $n - k$ (чтобы размер хотя бы одной части был меньше 2).
- 5.19 Нарисуйте дерево, которое иллюстрирует рекурсивные вызовы, выполняемые программой на упражнение 5.14 при размере массива равном 11.
- 5.20 Методом индукции докажите, что количество базовых функций, выполняемых любым алгоритмом типа 'разделяй и властвуй', который лежит между n и n^2 , в суммах составляет не менее n и не более n^2 (размер части рекурсивно, линейно связан с размером массива).
- 5.21 Докажите, что рекурсивное решение задачи о китайских башнях (программа 5.7) является оптимальным. То есть, покажите, что любое решение задачи, по меньшей мере, $2^n - 1$ перемещений.
- 5.22 Создайте рекурсивную программу, которая n -числом n и n -ой числу Фибоначчи с $2^{n-1} - 1$ ветками.
- 5.23 Исследуйте таблицу n порядковых чисел, полученную приведенной на рис. 5.9, на предмет скрытых свойств n -го числа, связывающих найденные n -го предпоследнего (последнего) ненулевого бита на рис. 5.7] при решении задачи о китайских башнях.
- 5.24 Создайте программу, которая обрабатывает задачу о китайских башнях путем запоминания максимума, содержащего все переменные, как это было в программе 5.7.
- 5.25 Создайте рекурсивную программу, которая запомнит массив размера n на 2^n нулей и единиц, чтобы можно было представить все n -порядковые числа, как списки на рис. 5.7.
- 5.26 Отобразите результаты использования рекурсивной программы рисования линий (программа 5.8) для следующих значений аргументов: `rule(4, 11, 4)`, `rule(4, 20, 4)` и `rule(7, 30, 5)`.
- 5.27 Докажите следующее свойство программы рисования линий (программа 5.8): если разность между n и размером двум аргументам является степенью 2, то все ее рекурсивные вызовы также обладают тем свойством.
- 5.28 Создайте функцию, которая эффективно подсчитает количество окончательных путей в заданном представлении шашки числа.
- 5.29 Сколько квадратов изображено на рис. 5.12 (включая те, которые скрыты большим квадратом)?
- 5.30 Создайте рекурсивную программу на C++, результатом которой будет функция-программа в форме списка писем функций `h u g b o x`, обеспечивающих

отображение нижнего рисунка рис. 5.12; функция `box` рисует квадрат размером `n` пикселов в точке с координатами `(x, y)`. Реализуйте функцию `box` в пакете команд `RawScript` (см. раздел 4.3).

5.31 Создайте возмущенную рекурсивную программу (аналогичную программе 5.9), которая обеспечит отображение верхнего рисунка рис. 5.12, как описано в упражнении 5.30.

- 5.32 Создайте программу `RawScript`, обеспечивающую рисование нижней части рис. 5.12.

• 5.33 Сколько линейных сегментов содержит левый Коха n -го порядка?

- 5.34 Рисование левых Коха n -го порядка сводится к выполнению последовательности команд типа: "повернуть на α градусов, затем нарисовать сегмент линии длиной $1/3$ ". Установите соответствие в системе счисления, позволяющее рисовать левую часть путем увеличения значения сегмента и последующего вычитания из базисного значения угла α .

- 5.35 Модифицируйте программу рисования левых Коха, приведенную на рис. 5.1, для создания другого фрактала, основанного на фигуре, состоящей из 5 линий нулевого порядка, вычерченных путем вращения на одну «основную единицу» в положительных, отрицательных, положительных, отрицательных и положительном направлениях (см. рис. 4.3).

5.36 Создайте рекурсивную функцию типа "разделяй и властвуй" для рисования по провансуанскому сегменту типичных пространств незначительных деформаций для заданных конечных точек. Примите, что координаты изменятся в интервале от 0 до M . Совет: лучше всего вычертите точку midway середины сегмента.

5.3 Динамическое программирование

Следует отметить алгоритмы типа "разделяй и властвуй", рассмотренных ранее в разделе 5.2, — разбиение квантовых задач на независимые подзадачи. Когда подзадачи не являются независимыми, ситуация усложняется, в первую очередь потому, что для подобных рекурсивных реализаций даже простейших алгоритмов этого типа может потребоваться непрактично большое количество времени. В этом разделе рассматривается систематический подход для избежания этой опасности в некоторых случаях.

Например, программа 5.10 — эффективная рекурсивная реализация рекуррентного соотношения, определяющего числа Фибоначчи (см. раздел 3.3). Не зная, сколько раз программа — это весьма неэффективно. Действительно, количество рекурсивных вызовов для вычисления F_n равно F_{n+1} . Но F_n приблизительно равно ϕ^n , где $\phi = (1 + \sqrt{5})/2$ — золотое сечение. Как это ни удивительно, но для программы 5.10 время этого элементарного вычисления определяется экспоненциальной зависимость. Рисунок 5.14, на котором приведены рекурсивные вызовы для небольшого примера, весьма наглядно демонстрирует требования к большому количеству вычислений.

И наконец, можно также вычислить значение N -го числа Фибоначчи за время, пропорциональное значению N , используя следующий массив:

```

F[0] = 0; F[1] = 1;
for i = 2; i <= N; i++
    F[i] = F[i-1] + F[i-2];

```

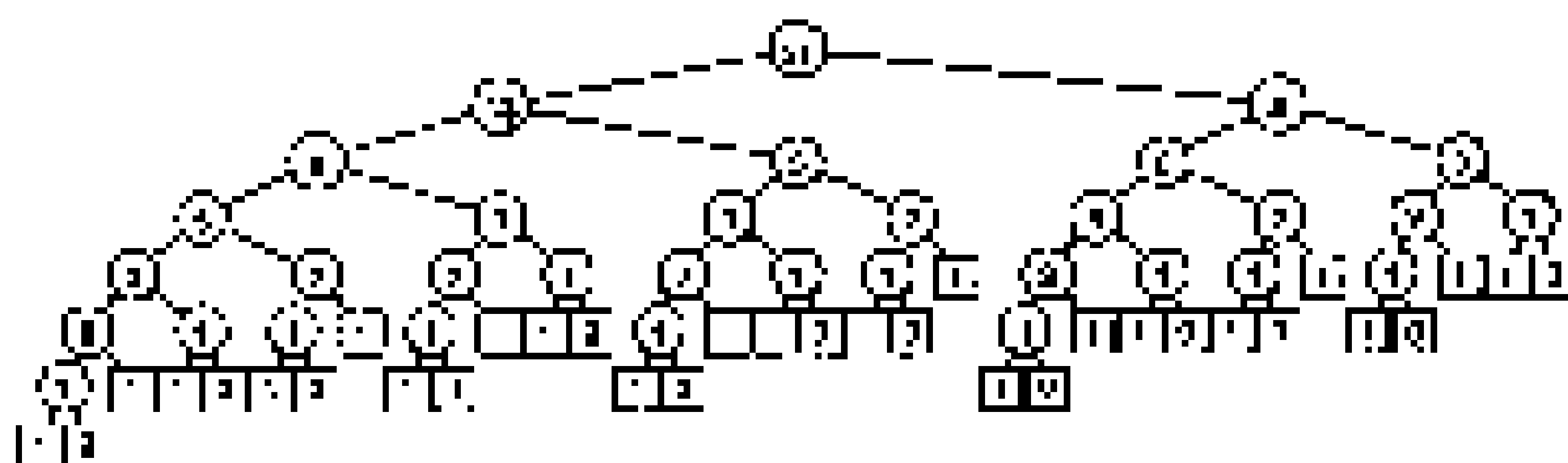



РИСУНОК 5.14 СТРУКТУРА РЕКУРСИВНОГО АЛГОРИТМА ДЛЯ ВЫЧИСЛЕНИЯ ЧИСЕЛ ФИБОНАЧЧИ

Следва да се разгледа и въпросът за изчисляване на $\hat{F}_j$ по отношение на избора на разпределение на данните. В статистическата теория разпределенията на данните се определят по различни критерии. В статистическата теория разпределенията на данните се определят по различни критерии. В статистическата теория разпределенията на данните се определят по различни критерии.

Програшкы 5.10 Числа фибоначчи (рекурсивна-реализация)

Эта программа, которую можно загрузить на компьютер и нащелкать кнопки, выводит на экран, в зависимости от выбора пользователя, время вычисления $T_{\text{выч}}$ прямо с помощью ЭВМ. Время вычисления $T_{\text{выч}}$ в $\mu\text{с}$ $\approx 1,8$ ($\approx 0,5$) и ≈ 1 для вычисления $F_{\text{выч}}$ на ЭВМ, соответственно $\phi^2 \approx 50$, если для вычисления $F_{\text{выч}}$ компьютеру требуется около секунды. Таким образом, время вычисления $T_{\text{выч}}$ для вычисления $F_{\text{выч}}$ на ЭВМ $\approx 1,8$ ($\approx 0,5$) и ≈ 1 $\mu\text{с}$.

```

int Fib(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
}

```

Указом предусматривается исключительный, по своему размеру и количеству выделенных площадей, $S_{\text{н}} = 153631903$ наибольшее число Флювиальных, которое может быть представлено 12-разрядным целым, поэтому достаточно использовать только 40 бит информации.

Этот способ представляет непосредственный способ получения численных решений для любых рекуррентных соотношений. В случае с числами Фибоначчи можно даже обойтись без массива и считать только два предыдущих значения (см. упражнение 5.3⁷); для многих других часто встречающихся рекуррентных соотношений (см., например, выражение 5.4⁸) необходимо поддерживать массив, хранящий все полученные значения.

Рекуррентное соотношение — это рекурсивная функция с целочисленным значением. Рассуждая, приведенное в предыдущем абзаце, показываем, что любую такую функцию можно приближенно вычислить, начиная все значения функции, начиная с начального, используя на каждом шаге ранее вычисленные значения для предыдущего рекуррентного соотношения. Что тем не менее является основой для вычисления про-

$$\begin{array}{l}
 2 F(6) \\
 \quad 1 F(5) \\
 \quad \quad 3 F(4) \\
 \quad \quad \quad 2 F(3) \\
 \quad \quad \quad \quad 1 F(2) \\
 \quad \quad \quad \quad \quad 1 F(1) \\
 \quad \quad \quad \quad \quad \quad 0 F(0) \\
 \quad \quad \quad \quad \quad \quad 1 F(1) \\
 \quad \quad \quad 1 F(2) \\
 \quad \quad \quad \quad 1 F(1) \\
 \quad \quad \quad \quad \quad 0 F(0) \\
 \quad 2 F(3) \\
 \quad \quad 1 F(2) \\
 \quad \quad \quad 1 F(1) \\
 \quad \quad \quad \quad 0 F(0) \\
 \quad \quad \quad 1 F(1) \\
 1 F(4) \\
 \quad 2 F(3) \\
 \quad \quad 1 F(2) \\
 \quad \quad \quad 1 F(1) \\
 \quad \quad \quad \quad 0 F(0) \\
 \quad \quad \quad 1 F(1) \\
 \quad 1 F(2) \\
 \quad \quad 1 F(1) \\
 \quad \quad \quad 0 F(0)
 \end{array}$$

есть. Например, при наличии типов предметов, представленных на рис. 5.16, груз, располагающий объемом, равным объему любого ранца 17, может быть только пять (но не шесть) предметов A общей стоимостью равной 21 или предметов B и C суммарной стоимостью равной 24 или предметов в одной из комбинаций других комбинаций. Наша цель — найти эффективный алгоритм для определения максимума среди всех возможностей при любых заданных выборе предметов и вместимости ранца.

Решения задачи о ранце важны во многих приложениях. Например, транспортная компания может потребоваться определить наилучшие способы загрузки грузовика или транспортного самолета. Подобные приложения могут распространяться и другие сферы этой задачи: например, количество элементов каждого пакета может быть ограничено или могут быть достигнуты для двух пакетов. Многие из таких приложений могут решаться путем перебора, но и же проблема, с которой мы сейчас имеем дело, слишком сложна для решения только что сформулированной базовой задачей: другие варианты оказываются гораздо более сложными. Можно часто обнаружить решение и перебором, но задача этого типа, что же является в части 5.

В рассмотренном примере задача о ранце при каждом выборе предмета мы предполагаем, что можем (рекурсивно) определить оптимальный способ заполнения оставшегося пространства в ранце. Для ранца размера cap , для каждого доступного типа элемента i он разделяется на две стоимости: элемент i , который можно было бы внести, оставшаяся i на элемент в ранце при оптимальной упаковке оставшихся элементов. Этот оптимальный способ упаковки — просто способ упаковки, определенный или который будет определен для меньшего ранца размера $cap-items[i].size$. В этом решении используется следующий принцип: оптимальное принятие решения в данной точке не требует перебора. Как только установлено, как оптимально упаковать ранцы меньших размеров, эти данные не требуются для дальнейшего исследования оставшихся элементов.

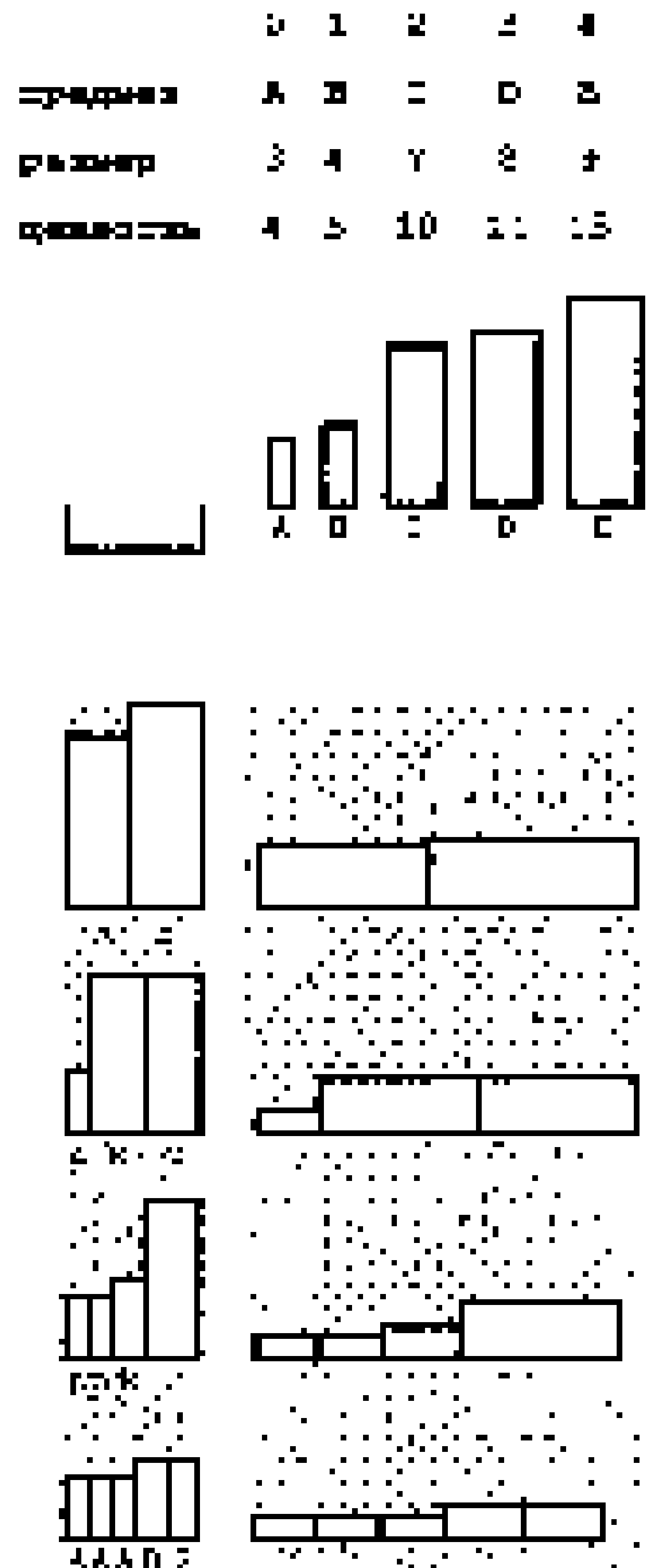


РИСУНОК 5.16 ПРИМЕР ЗАДАЧИ О РАНЦЕ

Решения данной задачи при каждом выборе предмета (элемент) являются вариантами ранца и могут быть оптимальными или неоптимальными по сравнению с оптимальным решением задачи (задача о ранце). На этом рисунке показаны четыре различных способа заполнения ранца, каждый из которых имеет cap для заданного способа заполнения ранца. Эти способы являются оптимальными способами заполнения ранца.

Программа 5.12 – прямое рекурсивное решение, основанное на принципе разделения. Эта программа также написана для решения поставленной задачи. Поскольку из-за большого объема получены значительные (см. рис. 5.17) время решения зависит от количества элементов массива. Но для решения проблемы можно автоматически генерировать динамическое программное решение, как показано в программе 5.13. Как и ранее, это решение исключает все возможные вычисления (см. рис. 5.18).

Програма 5.12 Задача о ранце (рекурзивна реализација)

А-а! Главная таму, чам мы ўчорашняе і сённяшняе раздзяляем, рэчыву падзяляем на часткі чалавечай фізічнасці. Не вядзецца кіраванасць гэтай праблеме, паколькі для нас вынікаюць патрабаванні вайны, асабліва цяжкія і неабходныя з сучаснага элементнага, агульнага, звычайна, дамагчыся большай задачы рашэння пачаць не ўдзяцца. Там на гэтым, праграма "радыкальна" мяняе нашы рэсурсы, ёсць і гэты момант развешчывае іх. (радыкальна 5.10). У нас падзяляецца, і гэтаксама падзяляецца структура і раздзяляе істнасць, якую я падзяляем і падзяляем абразам.

```
typedef struct { int size; int val; } Item;
```

Кроме того, с помощью заголовочных инициалов можно вводить в текст слова. Для этого достаточно ввести слово в меню (рекурсивно) и нажать на клавишу, которую можно было бы считать, эквивалентом клавиши **Tab**, а затем набрать нужные инициалы.

```

int knap(int cap)
{
    int i, space, max, l;
    for (i = 0; max = 0; i < K; i++)
        if (space = cap - items[i].size) > 0)
            if (i = knap(space) + items[i].val > max)
                max = i;
    return max;
}

```

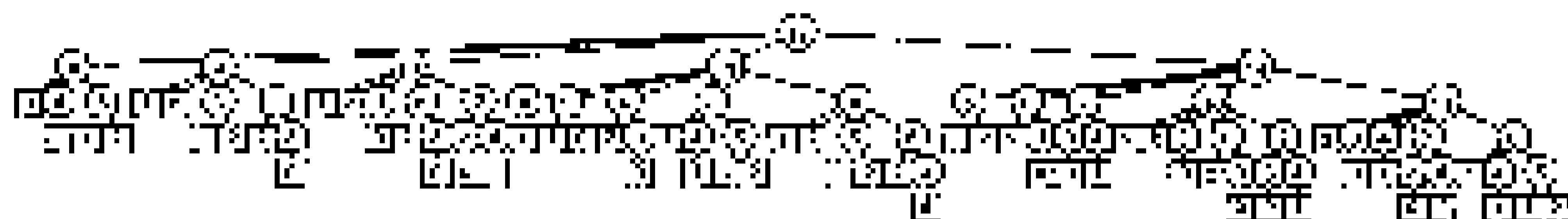


РИСУНОК 5.17 РЕКУРСИВНАЯ СТРУКТУРА АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ О РАЗРЕ.

[illegible]

Программа 5.13 Задача о ранце (динамическое программирование)

Эта программа, которая представляет собой модифицированную версию программы 5.12, связана с эффективным использованием памяти. Зависимость от количества элементов под М. проста: создавая любую вычисляемую значимую функцию, а затем вычисляя значения рекурсивных вызовов, то у нас будет созданная память, когда она требуется (используя адресованные для значения для представления вычисленных значений). Индекс элемента сохраняется, поэтому значения можно восстановить содержимое для того, чтобы вычислить ItemKnapsack[M] находится в базе. Вызванная процедура возвращает в оптимальной упаковке ранца размера M-ItemKnapsack[M].size. Следовательно, ItemKnapsack[M-ItemKnapsack[M].size] находится в ранце и т.д.

```
int knap(int M)
{
    int i, j, w, wmax, wmax1 = 0, w;
    if (ItemKnapsack[K] != null) return ItemKnapsack[M];
    for (i = 0; i < M; i++) {
        w = 0;
        for (j = 0; j < K; j++) {
            if ((wmax1 = ItemKnapsack[i].wmax) > 0)
                if ((w = knap(wmax1) + items[j].w) > wmax)
                    wmax = w;
        }
        ItemKnapsack[M] = new ItemKnapsack();
        return wmax;
    }
}
```

.....

Динамическое программирование принципиально исключает все повторные вычисления в любой рекурсивной программе, при условии, что мы можем себе позволить хранить таблицу функций для аргументов, которые меньше чем интересующий вызов.

Лемма 5.1 Динамическое программирование уменьшает время выполнения рекурсивной функции до минимального значения, которое достигается за минимальное количество вызовов, которое является тем самым базисом аргументов. Это означает, что количество рекурсивных вызовов ограничено полиномом.

См. упражнение 5.50.

Понимательно к задаче о ранце из леммы следует, что время выполнения пропорционально M^2 . Таким образом, задача о ранце может решаться эффективно, когда емкость ранца не очень велика; для почти больших емкостей время и требуемый объем памяти могут оказаться недопустимо большими.

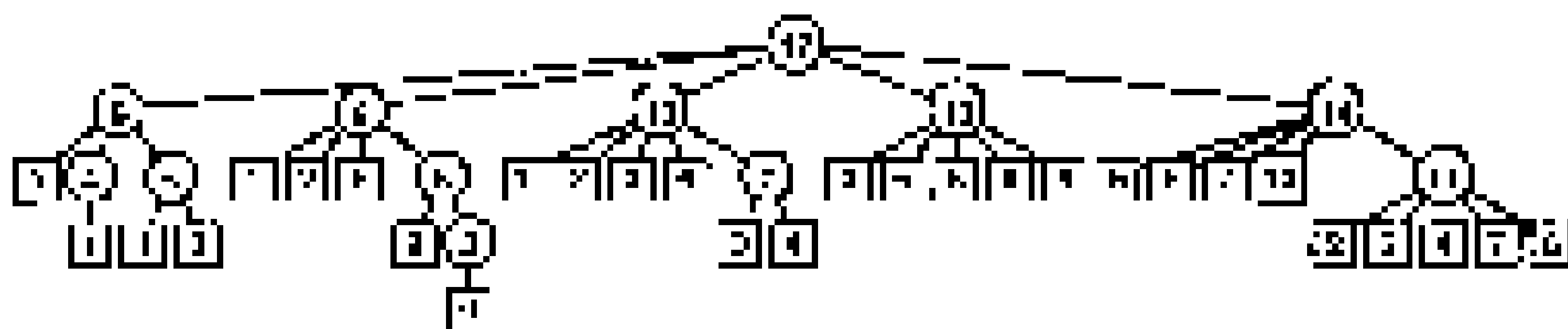


РИСУНОК 5.16 ПРИМЕНЕНИЕ МЕТОДА НИСХОДЯЩЕГО ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ ДЛЯ РЕАЛИЗАЦИИ АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ О РАНЦЕ

Подобно тому, как это было описано для задачи о рюкзаке, можно использовать динамическое программирование для решения задачи о ранце. Однако, в отличие от задачи о рюкзаке, для задачи о ранце не существует оптимального решения, которое можно найти за полиномиальное время.

Входящее динамическое программирование также применимо к задаче о ранге. Действительно, метод эффективного программирования можно применить во всех случаях, когда применим метод исходного программирования, хотя при этом необходимо убедиться, что значимая функция может быть вычислена в соответствующем зеркале, чтобы каждый нужный элемент присутствовал в требуемой момент. Для функции, значения которой для целочисленных аргументов, полученных рассмотрением, можно просто увеличить порожек аргумента (см. упражнение 5.51), для более сложных рекурсивных функций определение граничного порядка может быть сложной задачей.

Например, мы не обязаны ограничиваться рекурсивными функциями только с одним базисным аргументом. При наличии функции с несколькими базисными аргументами, решение можно хранить в многомерных массивах, по одному для каждого аргумента. В других ситуациях целочисленные аргументы вообще не требуются, и вместо этого можно использовать абстрактную формулировку статистической задачи, позволяющую разделить задачу на более сложные. Примеры задач этого рассмотрены в частях 5-8.

При использовании входящего динамического программирования несколько значений вычисляются, при повторении процесса негата динамического программирования, они вычисляются заранее. В обоих случаях исходящее динамическое программирование представляет собой следующее:

- оно представляет собой некоторую динамическую систему, решение которой
- порождает решение, определяемое тем, что;
- решение всех подзадач может не быть решено.

Приложения, в которых применяется динамическое программирование, различаются по способу подбора к задаче алгоритма и по типу кодирования.

Важный момент, который нельзя пропустить, заключается в том, что динамическое программирование становится эффективным, когда можно точно указать значимые функции, которые могут потребоваться, столь точно, что мы не можем себе позволить их хранить (иначе это будет означать, что мы не можем себе позволить их хранить (иначе это будет означать, что мы не можем себе позволить их хранить)). Например, если в задаче о ранге 2^k размер элемента - 64 -разрядное целочисленное число с плавающей точкой, мы не можем хранить информацию о подзадаче - 2^k (это не соответствует не просто небольшому числу - оно создает принципиальные трудности). Для подобных задач пока не известно ни одного достаточно эффективно решения; как будет показано в части 5, наиболее близкое решение состоит в том, что эффективного решения вообще не существует.

Динамическое программирование - это метод разработки алгоритма, который рассчитан в первую очередь на решение сложных задач того типа, который будет рассмотрен в частях 5-8. Большинство рассмотренных в частях 2 по 4 алгоритмов представляют собой различные случаи "базисных и конечных" и перекрывающихся подзадач, и основное внимание было уделено скорее субструктивности или субоптимальности задачи, чем субструктивности или субоптимальности. Однако, исходящее динамическое программирование является базовым

технологий разработки эффективных реализаций рекурсивных алгоритмов, зато не существует в принципе средств любого, кто принимает участие в разработке реализаций алгоритмов.

Упражнения

- 5.37 Напишите функцию, которая вычисляет F_n по модулю M , используя строки постоянной памяти для промежуточных вычислений.

5.38 Каково наибольшее значение N , для которого F_N может быть представлено в виде 64-разрядного целого числа?

- 5.39 Нарисуйте дерево, которое соответствует рис. 5.13 для случая, когда вычислена начальная часть рекуррентных соотношений в программе 5.11.

5.40 Создайте функцию, которая не использует восходящее динамическое программирование для вычисления значения F_N определенного рекуррентным соотношением

$$F_N = \lfloor N/2 \rfloor + F_{\lfloor N/2 \rfloor} + F_{\lfloor N/3 \rfloor}, \quad \text{для } N \geq 1, \quad \text{при } F_0 = 0.$$

Нарисуйте график зависимости $F_N = \lfloor N/2 \rfloor + F_{\lfloor N/2 \rfloor} + F_{\lfloor N/3 \rfloor}$ от N для $0 \leq N \leq 1024$.

5.41 Создайте функцию, в которой восходящее динамическое программирование используется для решения упражнения 5.40.

- 5.42 Нарисуйте дерево, которое соответствует рис. 5.13 для функции из упражнения 5.41 при значении $N = 25$.

5.43 Нарисуйте график зависимости количества рекурсивных вызовов, выполняемых функцией из упражнения 5.41 для значений F_N от N для $0 \leq N \leq 1024$. (Для выполнения этих вычислений для каждого значения N программа должна запускаться заново.)

5.44 Создайте функцию, в которой восходящее динамическое программирование используется для вычисления значения C_N определенного рекуррентным соотношением

$$C_N = N - \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}),$$

для $N \geq 1$, при $C_0 = 1$.

5.45 Создайте функцию, в которой нисходящее динамическое программирование применяется для решения упражнения 5.44.

- 5.46 Нарисуйте дерево, которое соответствует рис. 5.13 для функции из упражнения 5.45 при значении $N = 25$.

5.47 Нарисуйте график зависимости количества рекурсивных вызовов, выполняемых функцией из упражнения 5.45 для значений C_N от N для $0 \leq N \leq 1024$. (Для выполнения этих вычислений для каждого значения N программа должна запускаться заново.)

- 5.48 Приведите содержимое массивов `max` и `min` из программы 5.13 для массива `loop(17)` применительно к элементам, приведенным на рис. 5.15.

р 5.49 Нарисуйте дерево, соответствующее рис. 5. 5 при допущении, что элементы рассматриваются в порядке убывания их размера.

■ 5.50 Докажите лемму 5.3.

с 5.51 Создайте функцию, решающую задачу в задаче, используя терпеливое программирование 5.12, в которой применяется последующее динамическое программирование.

■ 5.52 Создайте функцию, которая решает задачу в ранее методом последующего или динамически программирования, но используя при этом рекурсивное решение, основывающееся на вычислении оптимального количества закрытого элемента, который должен быть помещен в рюкзак, исходя из определения (рекурсивно оптимальное) способа упаковки рюкзак без этого элемента.

с 5.53 Создайте функцию, решающую задачу в задаче, используя версию рекурсивного решения, соответствующую упражнению 5.52, в которой применяется последующее динамическое программирование.

■ 5.54 Воспользуйтесь динамическим программированием для решения упражнения 5.4. Обсуждайте эффективность общего количества создаваемых вызовов функций.

5.55 Создайте программу, в которой последующее динамическое программирование применяется для вычисления биномиального коэффициента $\binom{N}{k}$, исходя из рекурсивного соотношения

$$\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}, \quad \text{при} \quad \binom{N}{0} = \binom{N}{N} = 1.$$

5.4 Деревья

Деревья — это математические объекты, играющие важную роль при разработке и анализе алгоритмов, поскольку

- мы используем деревья для описания динамических свойств алгоритмов,
- мы строим и исследуем такие структуры данных, которые являются конкретными реализациями деревьев.

Мы уже встречались с примерами общих применений деревьев. В главе 1 была разработана структура для решения задачи поиска минимума, которая основывалась на структуре дерева, а в разделах 5.2 и 5.3 структура элементарных рекурсивных алгоритмов была описана с помощью структур деревьев.

Мы часто встречаемся с деревьями в повседневной жизни — это понятие настолько хорошо знакомо. Например, мы не можем отобразить порядок и последовательность с помощью генеалогического дерева: как будет показано, значительная часть форматов записываемых именно из этой области применения. Еще один пример — организация спортивных турниров: среди прочих, в последующем этом примере мы увидим чемпионат мира по футболу (FIFA World Cup). В качестве третьего примера можно привести организационную диаграмму большой корпорации, это применение отличается иерархическим разделением, характерным для организации типа 'разделяй и властвуй'. Четвертым примером служит дерево символьного разложения предло-

жания английского (или любого другого языка) на составляющие эта часть, такое дерево тесно связано с обработкой компьютерных текстов, как описано в главе 5. Типичный пример дерева, в котором описана структура этой книги, показан на рис. 5.19. В книге рассмотрены и множество других примеров применения деревьев.

Применительно к компьютерам, одно из наиболее известных применений структуры деревьев — для планирования файловых систем. Файлы хранятся в каталогах (иногда называемых также папками), которые рекурсивно описываются как последовательности каталогов и файлов. Это рекурсивное определение этого отражает естественное рекурсивное разбиение на составляющие и циклично определенную ориентированную часть дерева.

Существует множество различных типов деревьев, и также различия различия между абстрактной и конкретными представлениями, с которыми выполняется работа для данного приложения. Соответственно, мы подробно рассмотрим различные типы деревьев и их представления. Рассмотрение начнется с определения деревьев как абстрактных объектов и с ознакомления с большинством основных связей с этими терминами. Мы подробно рассмотрим различные типы деревьев, которые следует рассмотреть в поисках ответа на вопрос, каково это понятие:

- Деревья
- Деревья с корнем
- Упорядоченные деревья
- М-арные и бинарные деревья

После этого всестороннего рассмотрения мы перейдем к формальным определениям и рассмотрим представления и приложения. На рис. 5.20 приведены некоторые примеры из этих базовых концепций, которые будут сначала рассмотрены, а затем и определены.

Дерево (*tree*) — это непустые множества вершин и ребер, удовлетворяющих определенным требованиям. Деревом (*graph*) — это простой объект (называемый также узлом (*node*)), который может иметь имя и содержать другую информацию о нем. Информация: ребро (*edge*) — это связь между двумя вершинами. Лист (*leaf*) в дереве — это список отдельных вершин, в котором следующие друг из другом терминалы соединяются ребрами дерева. Ориентированное ориентированное дерево — ориентированное только одно путь, соединяющее любые два узла. Если между какой-либо парой узлов существует более одного пути или если между какой-либо парой узлов путь отсутствует, мы имеем граф, а не дерево. Непорядоченный набор деревьев называется борами (*forest*).

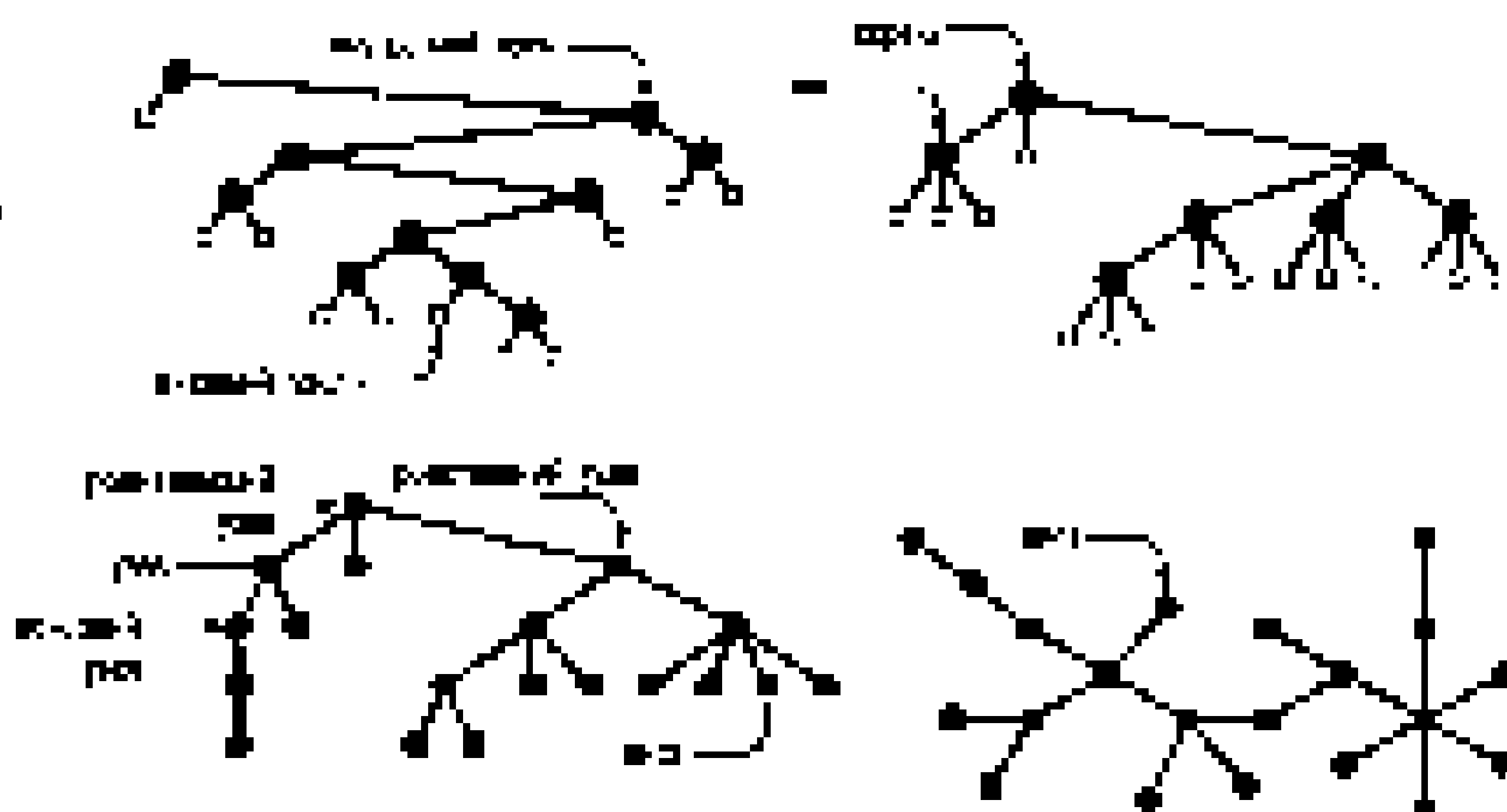


РИСУНОК 5.19 ДЕРЕВО

Это дерево описывает книгу, часть и различные главы книги. Каждый элемент представляет узел. Каждый узел состоит из идентификатора (имени) и информации о нем (например, о содержимом). В данном случае идентификатор — это название главы, а информация — это содержание главы.

РИСУНОК 2.20 ТИПЫ ДЕРЕВЬЕВ

На рисунке показаны различные типы деревьев: бинарное дерево (верхний левый), бинарное дерево с корнем (верхний правый), бинарное дерево с корнем (нижний левый), бинарное дерево с корнем (нижний правый) и бинарное дерево с корнем (нижний центр).



Дерево с корнем (однозначно, *rooted*) — это дерево, в котором один узел назначен корнем (или *root* дерева). В области компьютерной теории также обычно применяется к деревьям с корнем, в теории графов *базисное дерево* (*base tree*) — к более общей структуре, описанной в предыдущем абзаце. В дереве с корнем любой узел является корнем поддерева, состоящего из него и расположенных под ним узлов.

Существует только один путь между корнем и каждым из других узлов дерева. Данное определение никак не определяет направление рёбер: обычно считается, что все рёбра указывают от корня или к корню, в зависимости от приложения. Обычно деревья с корнем рисуются с корнем, расположенным в верхней части (хотя на первый взгляд это выглядит как неестественно), и говорят, что узел u расположен *над* узлом x (а x расположен *под* u), если x находится на пути от u к корню (т.е., u находится под x , как говорится на английском, и соединяется с x путём, который не проходит через корень). Каждый узел (а следовательно корень) имеет только один узел над ним, который называется его *родительским* (или *parent*) узлом, расположенным непосредственно *над* данным узлом, называемым его *потомком* (или *child*). Иногда аналогия с генеалогическими деревьями расширяется ещё больше и тогда говорят об узлах-дядях (*grand parent*) или родственниках (*relative*) узла, а не только узла.

Узлы, не являющиеся дочерним узлом, называются *листами* (*leaves*) или *терминальными* (окончательными, *terminal*) узлами. Для сопоставления с последним применением узла, являющегося листом, для одного дочернего узла, который называется *некорневой терминальной* узлом. В этой главе мы уже встречались с терминами *листья*, различающейся от типа узла. В деревьях, которые используются для представления структур, вызов рекурсивного алгоритма (например, рис. 2.14), *терминальные* узлы (окружности) представляют вызовы функции с рекурсивными вызовами, а *листья* (прямоугольники) представляют функции без рекурсивных вызовов.

В описанных приложениях способ упорядочивания дочерних узлов каждого узла имеет значение; в других это не важно. *Упорядоченное* (*ordered*) дерево — это дерево с корнем, в котором определён порядок следования дочерних узлов каждого узла. *Упорядоченное* дерево — естественное представление, например, при рисовании дерева дочерние узлы размещаются в определённом порядке. Действительно, многие другие конкретные представления имеют естественное представление порядка, на-

пример, очевидно, это различие имеет значение при рассмотрении представленных данных в компьютере.

Если каждый узел дерева имеет конкретное число дочерних узлов, то можно говорить о конкретном поршке. Мы имеем M -арное дерево. В таком случае часто можно определить специфичные поршки узлы, которые не имеют дочерних узлов. Тогда внешние узлы могут действовать в качестве финальных, на которые ссылается узлы, не имеющие указанного количества дочерних узлов. В частности, рассмотрев узлы M -арного дерева как бинарные деревья. Бинарное дерево (обычно так) — это упорядоченное дерево, состоящее из узлов двух типов: внешних узлов без дочерних узлов и внутренних узлов, каждый из которых имеет ровно два дочерних узла. Поскольку для дочерних узлов каждого внутреннего узла упорядоченность говорит о каком дочернем узле (left child и right child) является дочерним узлом (left child и right child) внутреннего узла. Каждый внутренний узел должен иметь левый и правый дочерние узлы, хотя один из них или оба могут быть внешними узлами. Если в M -арном дереве — это внутренний узел, то дочерние узлы являются конечными узлами.

Все это обобщенная терминология. Далее рассматриваются формальные определения, представляющие приложения, в порядке расширения понятия.

- Бинарные и M -арные деревья
- упорядоченные деревья
- деревья с корнем
- свободные деревья

Начиная с наиболее характерной абстрактной структуры, мы должны попытаться составить представление рассмотреть конкретные представления, как стало понятно из предыдущих глав.

Определение 5.1 Бинарное дерево — это либо пустой узел, либо упорядоченный узел, состоящий из корня (внешнего узла), который имеет левый и правый дочерние узлы.

Из этого определения становится понятно, что сами деревья — абстрактное математическое понятие. При работе с компьютерными графами и деревьями мы работаем только лишь с одним конкретным реализацией этой абстракции. Эта реализация не отличается от представления действительных чисел типичным типом float, целых чисел — типом int и т.д. Каждый из узлов дерева с узлом в корне, связанным с деревом с левым поддеревом, расположенным слева, и с правым поддеревом, расположенным справа, то есть является узлом конкретного представления. Существует множество различных способов представления бинарных деревьев (см., например, упражнение 5.62), которые понятию соответствуют идентичным, но сильно отличаются по качеству, как и можно было ожидать, учитывая абстрактный характер описания.

Часто всего понимается следующее конкретное представление при реализации программы: непустые узлы и манипулирующих бинарными деревьями — структура с двумя полями (left и right) для внешнего узла (см. рис. 5.21). Эти структуры связаны списком, так как каждый из них имеет для каждого узла, в нем по одному. Узлы также соответствуют внешним узлам. В частности, мы должны иметь

к стандартному представлению связанного списка, приведенному в разделе 1.3, следуем следующим образом:

```
struct node { Item item; node *l, *r; };
typedef node *list;
```

что представляет собой всего лишь код C++ для определения д.д. Эта структура из элементов `l` и `r` указывает на левый и правый указатели на узлы, которые также являются. Так, например, мы реализуем абстрактные операции `insert` и `delete` для дерева, создавая новую ссылку на указатель типа `x = y->l`.

Эта стандартная структура позволяет построить эффективную реализацию операций, которые понадобятся для перемещения по дереву `avl` от корня, но не для перемещения по дереву `avl` от дочернего узла к его родительскому узлу. Для этого, требуемых минимальных таких операций, можно добавить третью ссылку для каждого узла, указывающую к его родительскому узлу. Это можно реализовать аналогично двудеревянному списку. Как и в случае со связным списком (см. рис. 1.6), в определенной ситуации узлы дерева хранятся в массиве, а не хранятся в ссылках, используется лишь один, а не указатель. Конкретный пример такой реализации можно увидеть в разделе 12.7. Для определенных специализированных алгоритмов используется другая структура бинарного дерева, что наиболее полно показано в главе 11.

Из-за наличия такого множества различных возможных представлений можно было бы разработать ADT `Abstract Binary Tree` (Абстрактный тип дерева) бинарного дерева, настраиваемый разными способами, которые нужно реализовать, и реализующий различные операции. В данной книге данный подход не используется, поскольку

- чаще всего мы используем представление с двумя ссылками;
- мы используем деревья для реализации ADT более высокого уровня, а хотим сосредоточиться на минимуме на этой теме;
- мы работаем с алгоритмами, эффективность которых зависит от конкретного представления. ... это обстоятельство может быть упущено в ADT.

Поэтому же причинами мы используем уже знакомые конкретные представления массивов и связанных списков. Представление бинарного дерева, изображенное на рис. 2.2 — один из фундаментальных инструментов, который теперь добавлен к плану краткого списка:

Анализируя различные списки, мы видели рассмотрение с элементарных операций вставки и удаления узлов (см. рис. 1.3 и 1.4). При рассмотрении стандартного представления бинарного дерева мы также рассматриваем вставку и удаление узлов и удаление листьев второй связи. Если нужно удалить узел из бинарного дерева, при удалении следует помнить о проблеме наличия двух дочерних узлов и только одного родительского, с которым нужно работать после удаления узла. Существуют три естественных операции, для которых наиболее естественно не позволяет вставка нового узла в нижнюю часть дерева (создание нулевой связи с новым узлом), удаление листа (удаление связи с нулевой связью) и объединение двух листьев посредством создания нового корня, левая ссылка которого указывает на одно дерево, а правая — на другое. Эти операции интенсивно используются при манипулировании бинарными деревьями.

Определение 5.2 *M -арное дерево* — это узловой рдсг, либо упорядоченный рдсг, состоящий из упорядоченной совокупности M корней, каждый из которых является M -арным деревом.

Обычно узлы в M -арных деревьях представляются либо в виде структур с M не названными связями (как в бинарных деревьях), либо в виде массивов M связей. Например, в главе 15 рассмотрены 3-арные (или 3-розовые) деревья, в которых каждая вершина структуры с M именами имеет три связи (левый, средний и правый), каждая из которых имеет специальное значение для связывания с типичным алгоритмом. В частности, для случая неупорядоченных массивов для хранения связей алгоритм должен работать с произвольными M фиксированы, хотя, как будет показано, для использования такого представления скорее понадобится увеличить интенсивность использования памяти.

Определение 5.3 *Дерево* (также называемое *деревоподобным деревом*) — это рдсг (или линейный рдсг), состоящий из максимального количества вершин, таких что последовательность их имен не образует

Различие между упорядоченными деревьями и M -арными деревьями состоит в том, что узлы в упорядоченных деревьях могут иметь любое количество дочерних узлов, в то время как узлы в M -арных деревьях должны иметь точно M дочерних узлов. Иногда в контекстах, в которых требуется различать упорядоченные и M -арные деревья, мы используем термины *дерево* (или *дерево*) и *р-дсг*.

Поскольку каждый узел в упорядоченном дереве может иметь любое количество связей, представляется естественным рассмотреть его с использованием связанного списка. Я не уверен, что понимание связей с дочерними узлами. Пример такого представления приведен на рис. 5.22. Из этого примера видно, что тогда каждый узел содержит две ссылки: одну для связанного списка, соединяющего его с родственными узлами, и вторую для связанного списка с дочерними узлами.

Лемма 5.4 Структуры объединения соединений между двоичными деревьями и упорядоченными деревьями.

Эта лемма также показана на рис. 5.22. Любой бип можно представить в виде бинарного дерева, сделав левую ссылку каждого узла указывающей на его левый дочерний узел, а правую ссылку каждого узла — указывающей на родственного узла, расположенного справа.

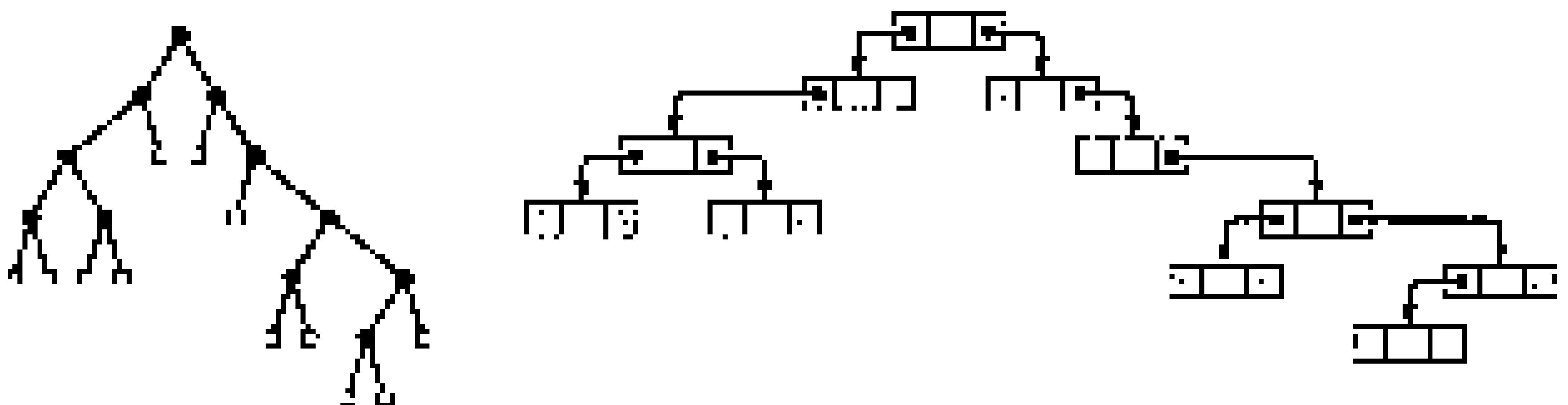


РИСУНОК 5.21 ПРЕДСТАВЛЕНИЕ БИНАРНОГО ДЕРЕВА

В бинарном дереве представление бинарного дерева осуществляется путем представления каждого узла в виде структуры, содержащей две ссылки: левую и правую. Левая ссылка указывает на левый дочерний узел, а правая ссылка — на родственного узла, расположенного справа.

Представьте себе, что перечисляете все ребра от одного какого-либо узла до другого, затем от этого узла к следующему и т.д. Последовательность ребер, ведущих от одного узла до другого, когда ни один узел не повторяется дважды, называется *простым путем*. Если появился замыкание (замкнутый), если существует простой путь, соединяющий какую-либо пару узлов. Простой путь, у которого первый и последний узел совпадают, называется *замком* (cycle).

Каждое дерево является графом; а какие же графы являются деревьями? Граф считается деревом, если он удовлетворяет любому из следующих условий:

- Граф имеет $N - 1$ ребро и N узлов узлы;
- Граф имеет $N - 1$ ребро и является связным;
- Только один простой путь соединяет каждую пару вершин в графе;
- Граф является связным, не является бисом таковым при удалении любого ребра.

Любое из этих условий — необходимое и достаточное условие для выполнения остальных трех. Формально, одно из них должно было бы служить определением свободных деревьев, но все вместе служат определением.

Мы предполагаем свободное дерево в виде некоторого ребра. Если представить свободное дерево неупорядоченным, упорядоченным или даже бинарным деревом, придется признать, что в общем случае существует множество различных способов представления каждого свободного дерева.

Абстрактная форма используется часто, и рассмотренные в этом разделе различные языки, поскольку имеют различные абстрактный язык — существенно составленная определением эффективного алгоритма и соответствующая структура данных для хранения данных. Часто приходится работать непосредственно с конкретными представлениями деревьев без какой-либо абстракции, но в то же время часто имеет смысл поработать с соответствующей абстрактной формой, а затем рассмотреть различные конкретные представления. В конце приводятся некоторые примеры этого процесса.

Прежде чем перейти к алгоритмам и представлениям, мы рассмотрим ряд основных математических свойств деревьев, эти свойства будут использоваться при разработке и анализе алгоритмов в главе 6.

Упражнения

- 5.56 Приведите представления свободных деревьев, показанных на рис. 5.20, в форме дерева с корнем и бинарного дерева.
- 5.57 Сколько существует различных способов представления свободных деревьев, показанных на рис. 5.20, в форме упорядоченного дерева?
- 5.58 Приведите три упорядоченных дерева, которые изоморфны по отношению к упорядоченному дереву, показанному на рис. 5.20. Другими словами, докажите существование возможности преобразования всех четырех деревьев одного и того же путем обмена дочерними узлами.
- 5.59 Предположите, что деревья содержат элементы, для которых определены операции $+$ и $*$. Создайте рекурсивную программу, которая удерживает бинарное дерево как теорию, описывающую эти операции, равны ли элементу (см. программу 5.5).

• 5.60 Изменить функцию 'разделить и захватить' для поиска максимального элемента в массиве (программа 5.6). Чтобы она делала меньше $\log_2 n$ частей, размер которых уменьшался бы не более чем на 1, рекурсивно находите максимум в каждой части и рекурсивной максимизации, но только один раз.

5.61 Нарисуйте 3-арные и 4-арные деревья, соответствующие максимизации $k = 3$ и $k = 4$ в рекурсивной конструкции, предложенной в упражнении 5.60, для элементов, состоящих из 11 элементов (см. рис. 5.6).

• 5.62 Бинарные деревья эквивалентны длинным строкам, в которых пустых символов не больше, чем символов при соблюдении дополнительного условия, что в любой позиции k количество пустых символов не больше, чем количество символов. Определите k . Бинарные деревья — это либо пуст, либо две такие строки, объединенные кавычкой, которая представляет 1. Нарисуйте бинарное дерево, соответствующее строке

111010110001010101

• 5.63 Упорядоченные деревья эквивалентны скобочным парам круглых скобок: упорядоченное дерево — это либо пуст, либо все возможные пары упорядоченных деревьев, заключенных в круглые скобки. Нарисуйте упорядоченное дерево, соответствующее строке

111101010101111101

■ 5.64 Составьте программу для определения того, представляют ли два массива N целочисленных значений между 0 и $N - 1$ изоморфные неупорядоченные деревья, если интерпретировать (как в главе 1) в форме связного разветвленного дерева n дерева с узлами, пронумерованными от 0 до $N - 1$. То есть, программа должна определить, существует ли способ размещения номеров узлов в одном дереве, чтобы представления в виде массива одного дерева были точно таким же представлением в виде массива другого дерева.

■ 5.65 Составьте программу для определения того, представляют ли два бинарных дерева изоморфные неупорядоченные деревья.

• 5.66 Нарисуйте все упорядоченные деревья, которые могли бы представлять дерево, определенное набором ребер 0-1, 1-2, 1-3, 1-4, 4-5.

• 5.67 Докажите, что если в связном графе, состоящем из N узлов, удовлетворяют условию ребра имеют хотя бы один разветвленный граф, то он имеет $N - 1$ ребро и является циклом.

5.5 Математические свойства бинарных деревьев

После того как мы приступим к рассмотрению алгоритмов сортировки деревьев, представляющих данные, рассмотрим ряд базовых свойств деревьев. Мы сосредоточим внимание на бинарных деревьях, поскольку они используются чаще всего другими. Приведенные ниже основные свойства являются фундаментальными для понимания характеристик эффективности различных алгоритмов, с которыми мы встретимся — не только тех, в которых бинарные деревья используются в качестве данных, но и рекурсивных алгоритмов типа 'раздели и захвати' и других аналогичных техник.

Лемма 5.5 Бинарное дерево с N внутренними узлами имеет $N + 1$ внешних узлов.

Эта лемма доказывается методом индукции: бинарное дерево без внутренних узлов имеет один внешний узел, следовательно, лемма справедлива для $N = 0$. Для $N > 0$ любое бинарное дерево с N внутренними узлами имеет k внутренних узлов k — это поддеревья с $k - 1 - i$ внутренними узлами в левом поддереве и i внутренними узлами в правом поддереве для некоторого i в диапазоне между 0 и $N - 1$, поскольку корень является внутренним узлом. В соответствии с индуктивным предположением левое поддерево имеет $k - 1 + 1$ внешних узлов, а правое поддерево — $N - k + 1$ внешних узлов, что в сумме составляет $N + 1$.

Лемма 5.6 Бинарное дерево с N внутренними узлами имеет $2N + 1$ листьев и поразительно узкую $N + 1$ связь с внешними узлами.

В каждом дереве с корнем каждый узел, за исключением корня, имеет единственного родителя-узла, и каждый ребро соединяет узел с его родителем-узлом, следовательно, существует $N + 1$ связей, соединяющих внутренние узлы. Аналогично, каждый из $N + 1$ листьев узел имеет одну связь со своим единственным родителем-узлом.

Характеристики произвольности многих алгоритмов зависят не только от количества узлов в бинарном дереве, но и от различных структурных свойств

Определение 5.6 *Глубина (leaf) узла в дереве — это количество узлов его родителя (узла родителя) на пути до корня (0). Высота (height) дерева — максимальная глубина любого узла. Длина пути (path length) дерева — сумма длин всех путей от корня до листьев. Длина внешнего пути (external path length) бинарного дерева — сумма длин всех внешних путей от корня до листьев.*

Удобный способ вычисления длины пути дерева: эквивалент — суммирование произведений k на число узлов на уровне k для всех k .

Из этих определений следуют также простые рекурсивные определения, вытекающие непосредственно из рекурсивных определений деревьев и бинарных деревьев. Например, высота дерева на 1 больше максимальной высоты поддеревьев его корня, а длина пути дерева, состоящего из узла равна сумме длины пути от корня плюс $N + 1$. Приведенные определения также непосредственно связаны с анализом рекурсивных алгоритмов. Например, для многих рекурсивных вычислений при анализе сложности алгоритма и размера работы макса — минуса глубина рекурсии, размер стека, необходимого для поддержки выполнения.

Лемма 5.7 Для любого бинарного дерева, состоящего из N внутренних узлов, $2N + 1$ больше длины внешнего пути.

Эту лемму можно было бы доказать методом индукции, но есть другой, более интересный доказательство (которое напрямую и для доказательства леммы 5.6). Обозначим явно, что любое бинарное дерево может быть структурировано при помощи внешнего принципа: начните с бинарного дерева, состоящего из корня и внешнего узла. Затем выберите любое $k \geq 0$ и выберите узлы, которые занимают все внутренние узлы с двумя внутренними внешними узлами. Если выбранный внешний узел равен k , то его внешний узел увеличивается на k , но длина внешнего пути увеличивается на $k - 2$ (таким образом

узлов на уровне k составляет 2^k для N узлов на уровне $k = 1$ (допускается). Процесс начинается с дерева длиной внутреннего и внешнего путей которого равен 0, и на каждом из k этапов длина внешнего пути увеличивается на 2 больше, чем длина пути внутреннего пути.

Лемма 5.8 Высота бинарного дерева с N внутренними узлами не меньше $\lg N$ и не больше $N - 1$.

Худший случай — вырожденное дерево, состоящее только из одного пути из $N - 1$ узлов (допускается) (см. рис. 5.23). В лучшем случае мы имеем уравновешенное дерево с 2 внутренними узлами на каждом уровне k , за исключением одного последнего (см. рис. 5.24). Если высота равна k , то должно быть справедливо соотношение

$$2^{k-1} \leq N - 1 \leq 2^k$$

поскольку существует $N - 1$ внешняя узлы. Из этого неравенства следует приближенная лемма: в лучшем случае высота равна $\lg N$, округленному до ближайшего целого числа.

Лемма 5.9 Длина вырожденного пути бинарного дерева с N внутренними узлами не меньше чем $\lg(N/4)$ и не превышает $N(N - 1)/2$.

Худший и лучший случаи соответствуют тем же деревьям, которые упоминались при рассмотрении рекуррентных соотношений на рис. 5.25. В худшем случае длина внутреннего пути дерева равна $0 + 1 + 2 + \dots + (N - 1) = N(N - 1)/2$ (см. рис. 5.26). В лучшем случае дерево имеет $(N + 1)$ внутренних узлов при высоте, не превышающей $\lg N$. Применяя лемму 5.7, мы получаем предельное значение $(N + 1) \lg N - 2N \leq N(N/4)$.

Как мы увидим, бинарные деревья часто применяются в компьютерных приложениях, и при этом эффективность повышается, когда бинарные деревья полностью уравниваются (или близки к этому). Например, деревья, которые использовались нами для описания алгоритмов "разделяй и властвуй", являются бинарными по своей структуре и соотношению длин путей, как мы видели на рис. 5.74). В главе 9 и 13 рассматриваются новые структуры данных, основанные на уравнивании путей деревьев.

Эти основные свойства деревьев представляют информацию, которая требуется для разработки эффективных алгоритмов решения многих практических задач. Более подробный анализ нескольких специфических алгоритмов, в которых приходится применять деревья, требует знания математического анализа, хотя часто полезные приближен-

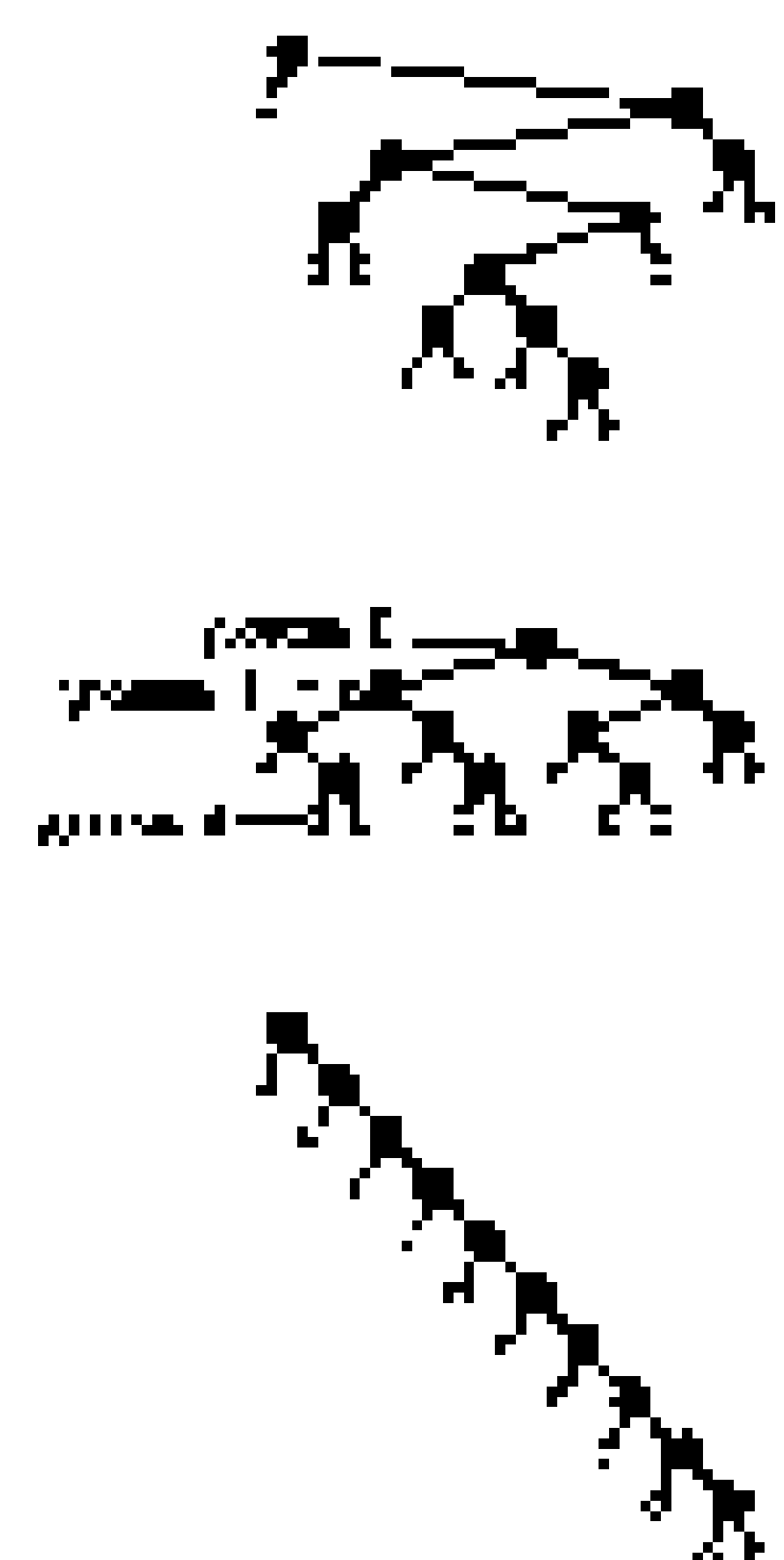


РИСУНОК 5.23 БИНАРНОЕ ДЕРЕВО С 10 ВНУТРЕННИМИ УЗЛАМИ

Бинарное дерево, состоящее из пути, имеет высоту равную N . Длина внутреннего пути равна $0 + 1 + 2 + \dots + (N - 1)$ (полностью уравниваемое бинарное дерево (с высотой) ≤ 10 вырожденное дерево имеет высоту равную 4. Длина внутреннего пути равна $0 + 1 + 2 + 3 + 4 = 10$ (длина внешнего пути равна 11). Для бинарного дерева с 10 внутренними узлами высота не может быть больше, чем 10 (максимальная высота бинарного дерева с 10 внутренними узлами имеет высоту равную 10). Если мы увеличим путь равным 42 и будем добавлять узлы равным 50 (см. рис. 5.26) бинарное дерево с 10 узлами не может иметь высоту больше 50 (максимум 51 для бинарного дерева).

желаных оценки можно получить, прибегнув к простейшим из алгоритмов округления, применяемым исполнителям в этом разделе. В последующих главах мы продолжим исследование методов поиска эффективных деревьев, в которых возникают всевозможные сложности. Тогда можно вернуться к теме алгоритмов.

Упражнения

- 5.68 Сколько внешних узлов содержится в M -арном дереве с N внутренними узлами? Используйте свойство для определения объема памяти, необходимого для представления такого дерева, если известно, что для каждой связи к каждому элементу требуется по одному биту памяти.
- 5.69 Приведите верхнее и нижнее граничные значения высоты M -арного дерева с N внутренними узлами.
- 5.70 Приведите верхнее и нижнее граничные значения длины внутреннего пути M -арного дерева с N внутренними узлами.
- 5.71 Приведите верхнее и нижнее граничные значения количества листьев в M -арном дереве с N узлами.
- 5.72 Покажите, что если листья внешнего узла в бинарном дереве различаются на константу, то высота составляет $O(\log N)$.
- 5.73 Деревья Фибоначчи высоты $n \geq 2$ — это бинарные деревья с деревом Фибоначчи высотой $n - 1$ в левом поддереве и деревом Фибоначчи высотой $n - 2$ в правом. Деревья Фибоначчи высотой 0 — это единственные внешние узлы, а деревья Фибоначчи высотой 1 — единственный внутренний узел с двумя внешними дочерними узлами (см. рис. 5.14). Выразите высоту и длину внешнего пути дерева Фибоначчи высотой n в виде функции N (количества узлов в дереве).
- 5.74 Дерево типа "разделяй и властвуй" состоит из N узлов. Это бинарное дерево с корнем, обозначенным N , деревом типа "разделяй и властвуй", состоящим из $\lfloor N/2 \rfloor$ узлов, в левом поддереве и деревом типа "разделяй и властвуй", состоящим из $\lceil N/2 \rceil$ узлов, в другом. (Дерево типа "разделяй и властвуй" показано на рис. 5.6.) Нарисуйте дерево типа "разделяй и властвуй" с 11, 12, 13 и 14 узлами.
- 5.75 Покажите методом индукции, что длина внутреннего пути дерева типа "разделяй и властвуй" находится в пределах между $N \lg N$ и $N \lg N + N$.
- 5.76 Дерево типа "объединяй и властвуй" состоит из N узлов. Это бинарное дерево с корнем, обозначенным N , деревом типа "объединяй и властвуй", состоящим из $\lfloor N/2 \rfloor$ узлов, в левом поддереве и деревом типа "объединяй и властвуй", состоящим из $\lceil N/2 \rceil$ узлов, в другом (см. упражнение 5.18). Нарисуйте дерево типа "объединяй и властвуй" с 11, 12, 13 и 14 узлами.
- 5.77 Докажите методом индукции, что длина внутреннего пути дерева типа "объединяй и властвуй" находится в пределах между $N \lg N$ и $N \lg N + N$.
- 5.78 Подход (конкретно) бинарное дерево — это дерево, в котором все узлы, кроме, возможно, последнего, являются внешними. Нарисуйте дерево, заполненное, как прототипированное на рис. 5.24. Докажите, что длина внутреннего пути полного дерева с N узлами лежит в пределах между $N \lg N$ и $N \lg N + N$.

5.6 Обход дерева

Прежде чем рассмотреть алгоритмы, конструирующие бинарные деревья и деревья, рассмотрим алгоритмы для реализации наиболее общей функции обработки дерева — обхода узлов; при наличии указателя на дерево требуется систематически обработать все узлы в дереве. В основном списке переход от одного узла к другому выполняется за счет отслеживания единственной ссылки; однако, в бинарных деревьях приходится принимать решения, поскольку может существовать несколько связей, по которым можно двигаться.

Начнем рассмотрение с процесса для бинарных деревьев. В списке обхода узлов список узлов не является линейной возможностью (см. программу 5.5): обработать узел, а затем следовать слева (в этом случае узлы, помещаясь бы по порядку), или следовать справа, а затем обработать узел (в этом случае узлы помещались бы в обратном порядке). При работе с бинарными деревьями существуют две схемы обхода: линейно, три основных порядка возможности посещения узлов:

- *Лучший обход (preorder walk)*, при котором мы посещаем узел, а затем слева и справа поддеревья
- *Левосторонний обход (leftward walk)*, при котором мы посещаем левое поддерево, затем узел, а затем правое поддерево
- *Обратный обход (postorder walk)*, при котором мы посещаем левое и правое поддерево, а затем узел.

Эти методы можно легко реализовать с помощью рекурсивной программы, как показано в программе 5.14, которая является каноническим обобщением программы 5.5 входа в список. Для реализации обхода в других перечисленных функциях образцы переставить подстроки функций в программе 5.14. Порядок посещения узлов в примере дерева при использовании каждого из способов обхода показан на рис. 5.16. На рис. 5.25 приведен псевдокод функции, которая вычисляет при заданном указателе 5.14 пример 5.14 на рис. 5.26.

Программа 5.14 Рекурсивный обход дерева

Эта рекурсивная функция принимает в качестве аргумента ссылку на дерево и вызывает функцию visit для каждого из узлов дерева. В теле функции обход дерева прямой обход, если существует обращение к visit узлу рекурсивные вызовы, выполняется посторонний обход; в противном случае обращение к visit узлу рекурсивных вызовов — то обратный обход.

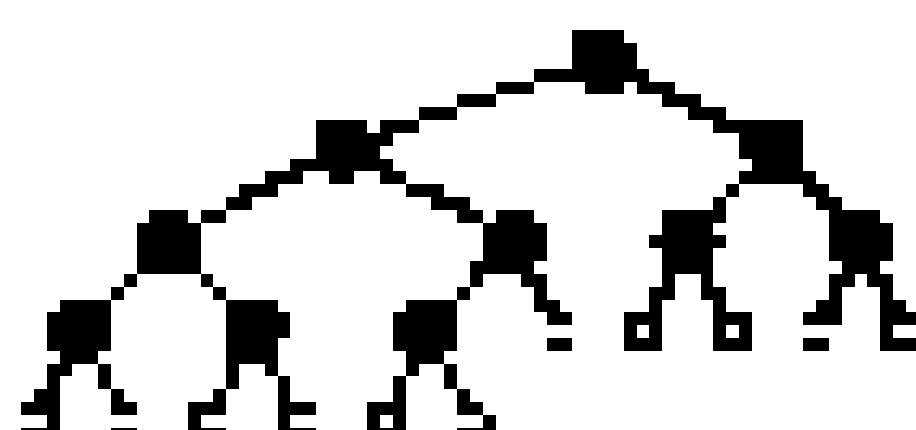
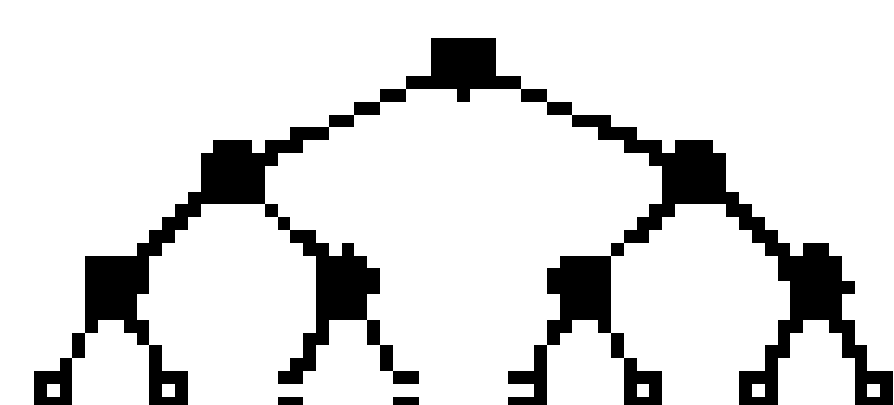


РИСУНОК 5.14 ПОЛНЫЕ БИНАРНЫЕ ДЕРЕВЬЯ С СЕМЬЮ И ДЕСЯТЬЮ ВНУТРЕННИМИ УЗЛАМИ

Каждый из представленных вариантов обхода дерева может быть реализован с помощью рекурсивной функции. В примере обхода дерева, показанного на рис. 5.16, обход дерева осуществляется в следующем порядке: сначала — левое поддерево, затем — узел, а затем — правое поддерево. В примере обхода дерева, показанного на рис. 5.16, обход дерева осуществляется в следующем порядке: сначала — левое поддерево, затем — узел, а затем — правое поддерево.


```
void traverse(link h, void visit(link))
{
    if (h == 0) return;
    visit(h);
    traverse(h->l, visit);
    traverse(h->r, visit);
}
```

С фактом же существования рекурсивных функций, на которых основаны все различные методы обхода дерева, мы уже встречались в рекурсивных программах типа "разделяй и властвуй" (см. рис. 3.5 и 3.11) и в алгоритмических построениях. Напомню, что выполнение прямого обхода соответствует обходу слева направо, а выполнение обратного рекурсивного обхода (см. рис. 3.11), выполненное поперечного обхода соответствует переменеку лавого. Вспомогательная лавка в решении задачи о ханойских башнях между рекурсивными вызовами, которые переменяют все остальные лавки; выполнение обратного обхода соответствует планированию построения лавки лавки и т.д. Эти функции (рис. 3.11) можно использовать для решения задачи о ханойских башнях и обхода дерева. Например, известно, что при обходе дерева каждой второй узел является внешним, но той же причине, по какой при решении задачи о ханойских башнях каждая вторая переменная затрагивает соседнюю лавку.

Получив также рассуждения перекрестные разложения, т. е. которые имеют форму α и β стек. Для того чтобы начать с разложения абстрактного стека, рассмотрим элемент дерева, минимализированного дерева, которое требуется обобщить. Затем мы войдем в цикл, в котором вытекающим и обрабатываемся перекрест: стек стека, в который придет элемент, пока стек не опустеет. Если попытаются добавить элемент, мы получим это, если элементу не присвоить дерева, мы выведем последовательность элементов кодирования, которая зависит от требуемого порядка.

- Для хранения сырья применяется: сырое куриное, из
его мяса получается, а затем утка
- Для приготовления сырья применяется: сырое куриное,
из него утка, а затем утка
- Для приготовления сырья применяется: утка, затем сырое
куриное, а затем утка

Пулеские деревья не стек не инвариантны. На рис. 5.27 показано содержимое стека при использовании функции на этих трех случаях для обхода примера дерева, приведенного на рис. 5.26. Методом индукции можно легко убедиться, что для любого бинарного дерева эти шаги приведут к тому же результату, как и рекурсивный метод.

РИСУНОК 5.25 ВЫВОДЫ ФУНКЦИИ ПРЯМОГО ОБОДА

THE UNIVERSITY OF
SOUTH ALABAMA
LIBRARY
333 UNIVERSITY BLVD.
MOBILE, AL 36688

"04/03/2012 12:51K 320

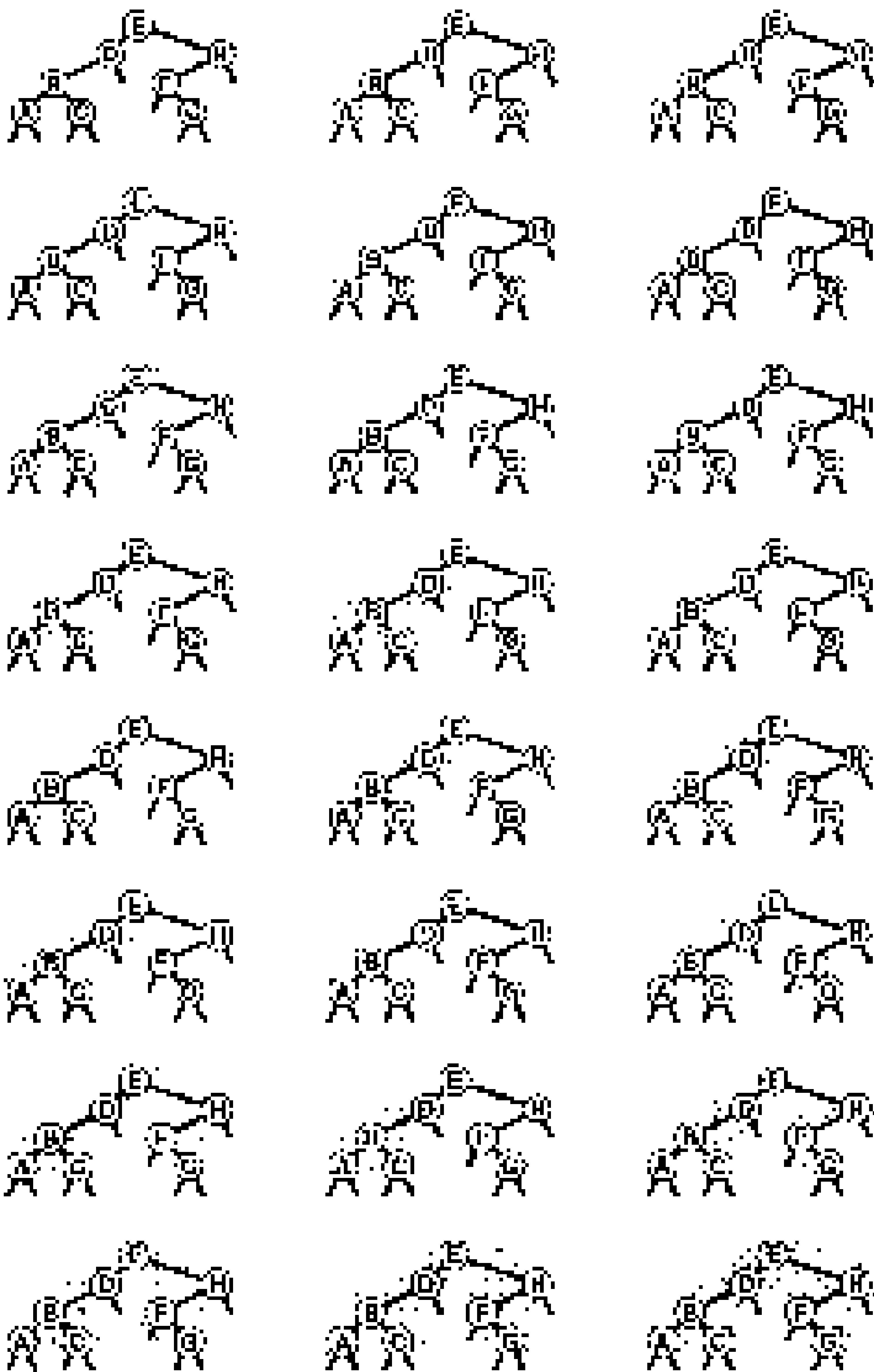


РИСУНОК 5.26 ПОРЯДОК ОБХОДА ДЕРЕВА

Эта иллюстрация выводит последовательность левых, средних и правых узлов (по порядку /слева/, посередине /в центре/ и справа) в порядке обхода дерева.

похожую к ней представилит существенно различающиеся деревья, в зависимости от конкретной задачи. В частности, проход по узлам не соответствует рекурсивной реализации, связанной с рекурсивной структурой дерева.

Программа 5.16 Обход по уровням

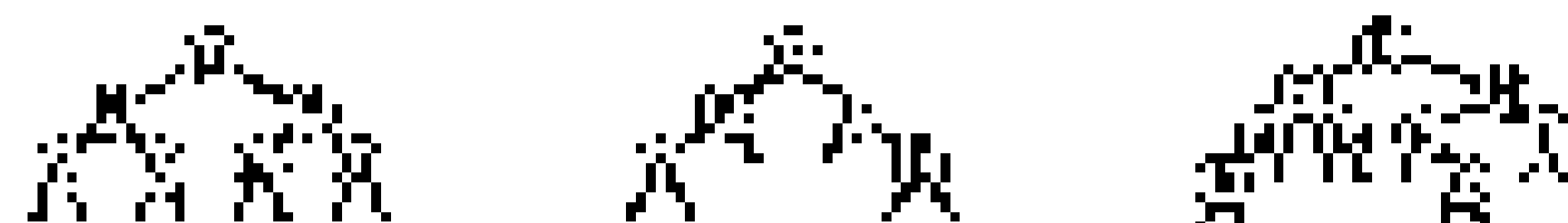
Изменяет структуру данных, лежащей в основе прямого обхода (см. программу 5.15), со стека на очередь парнупл с преобразованием обхода в обход по уровням.

```
void traverse(link h, void (*f)(link))
{ queue<link> q(h);
  q.push(h);
  while (!q.empty())
  {
    while (z = q.front());
    if (z->l != 0) q.push(z->l);
    if (z->r != 0) q.push(z->r);
  }
}
```

Прямой проход, обратный проход и проход по уровням так различно описываются и для бинарных. Чтобы образовать такие алгоритмы, представьте себе бинарное дерево с изображением корня. Тогда алгоритм для прямого обхода формулируется следующим образом: 'показать корень, а затем каждое из поддеревьев', а алгоритм для обратного прохода — 'показать каждое из поддеревьев, а затем корень'. Принцип для обхода по уровням тот же, что и для бинарных деревьев. Неэффективные реализации этих методов — простые обобщения программ полного обхода с использованием стека (программы 5.14 и 5.15) и программы обхода по уровням с использованием очереди (программа 5.16) для бинарных деревьев, которые мы только что рассмотрели. Сравнения по конкретным реализациям не приводятся, из-за отсутствия места в книге рассмотрены более общие процедуры.

Упражнения

- 5.79 Преподнесите подробное пояснение узлам для прямого, обратного, обходного и обхода по уровням для бинарных деревьев.



- 5.80 Отредактируйте программу со стеком обхода по уровням (программа 5.16) по рис. 5.79, так чтобы она показывала на рис. 5.27.

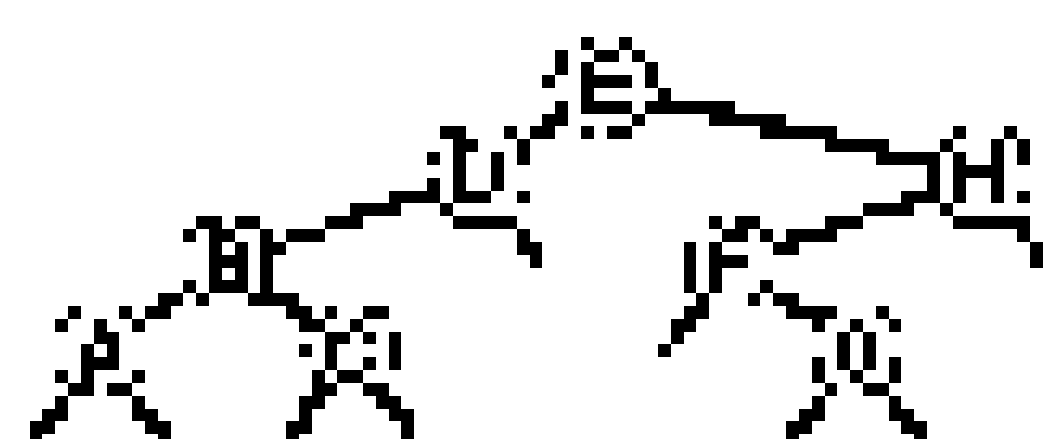
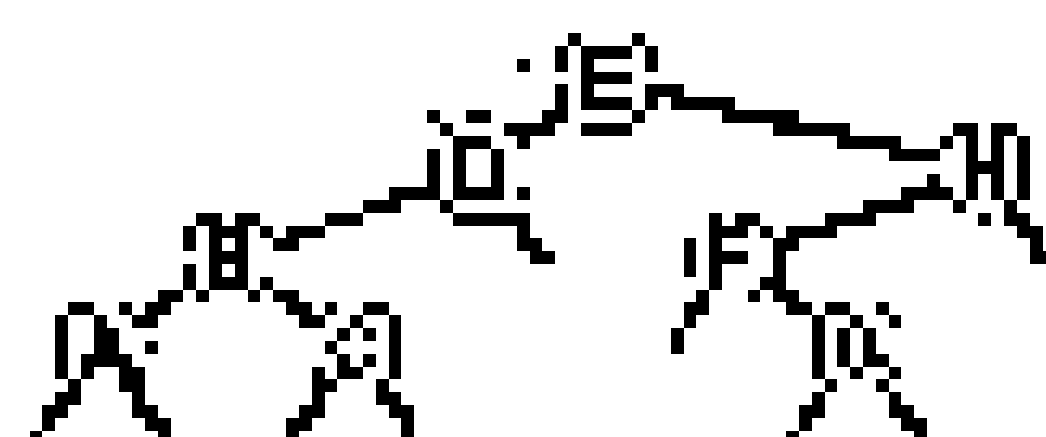
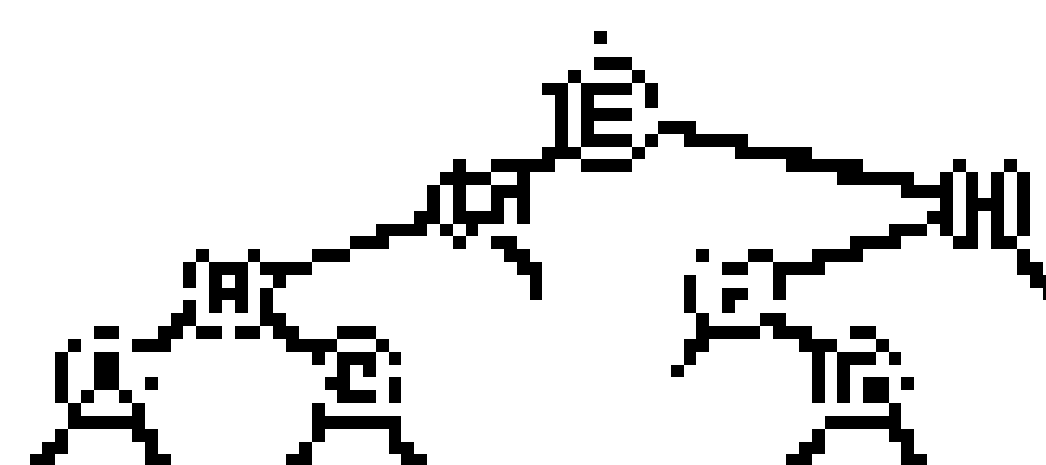


РИСУНОК 5.28 ОБХОД ПО УРОВНЯМ

Эта последовательность изображений дерева показывает, как узлы в очереди стекаются в свой очередь.

5.81 Предположим, что прямой обход бора равенся прямому обходу соответствующего бинарного дерева (см. задачу 5.4), а обратный проход бору эквивалентен полному обходу бинарного дерева.

а. §.82 Принадлежит перекресточная редакционная пометка: «откуда?»

■ 5,83 Իրոնգլուխ բաղնիքի քանակը բնականորեն չփոխվում է:

- 5.84 Создать программу, которая преобразует полную и латинскую таблицы умножения дерева в таблицу дерева по узлам.

5.7 Рекурсивные алгоритмы бинарных деревьев

Алгоритмы обработки данных, рассмотренные в разделе 5.5, наглядно демонстрируют необходимость рассмотрения рекурсивных алгоритмов бинарных деревьев. Это обусловлено самой рекурсивной структурой этих деревьев. Для решения задачи, естественно, можно непосредственно написать рекурсивные алгоритмы типа "разделяй и властвуй", которые, по существу, обобщают алгоритмы обхода деревьев. Разработка языка языка может породить новый алгоритм решения задачи (рекурсивный обход деревьев); применение можно выразить через метод, между тем после рекурсивного обхода можно еще раз переписать все те методы.

Часто требуется определять различные структурные параметры дерева, имея только остовку этого дерева. Например, программа 5.3 содержит рекурсивные функции для вычисления количества узлов и высоты заданного дерева. Функции определяются непосредственно исходя из определения 5.1. Ни одна из этих функций не требует порядка обхода при рекурсивном выводе. Они обходятся всем узлам дерева и возвращают одинаковый результат, если, например, рассмотреть в качестве параметра не ствол. Не все параметры дерева вычисляются так просто: например, продолжительность эффективного выполнения длины внутреннего пути. Олигархическое дерево является более сложной тем. (примеры 5.33–5.35).

Программа 5.7 Вычисление параметров дореза

Для выполнения задач структурных анализ и анализ можно использовать и другие усовершенствованные процедуры, подобные следующим.

```

int count(link h)
{
    if (h == 0) return 0;
    return count(h->l) + count(h->r) + 1;
}

int height(link h)
{
    if (h == 0) return -1;
    int u = height(h->l), v = height(h->r);
    if (u > v) return u+1; else return v+1;
}

```

Еще одно функция, которая должна быть реализована, обрабатывающая данные. Функция, которая выводит на экран структуру или рисует дерево. Например, программа 5.18 — рекурсивная процедура, принимающая данные в формате, принятом на рис. 5.24. Другое дерево рекурсивно созданное можно использовать для

рисования более сложной представленной деревом, подобным использованному на рисунке в этой книге (см. упражнение 5.15).

Программа 5.18 Функция быстрого вывода дерева

Эта рекурсивная программа способна вывести дерево и использовать эту информацию для печати его представления, которое можно использовать для отладки программ (функция `drawtree` (см. рис. 5.29). В программе предполагается, что элементы в узлах имеют тип `Item`, для которого определены геттерские операции `<<`.

```
void printnode(Item x, int h)
{ for (int i = 0; i < h; i++) cout << " ";
  cout << x << endl;
}

void drawtree(T, int h)
{ if (t == 0) { printnode(*t, h); return;
}
  draw(t->l, h+1);
  printnode(*t->data, h);
  draw(t->r, h+1);
}
```

Программа 5.18 выполняет построчный вывод, но если вывести элементы перед рекурсивными вызовами, получится правый вывод; этот вариант также приведен на рис. 5.29. Этот формат привычен, например, при отображении генеалогического дерева, списка файлов в файловой структуре и виде дерева или графа заданной структуры печатного документа. Например, выполнение прямого обхода дерева, изображенного на рис. 5.19, приводит к выводу значения, изображенного на рисунке в этой книге.

Вспомогательный пример программы, которая строит некую структуру данных, связанную с приложением определения чужих мушкетеров, которая была рассмотрена в разделе 5.2. Наше дерево — изображение мушкетера — бинарного дерева, в котором в каждом внутреннем узле является кинжал больше, чем элемент его двух дочерних элементов. В частности, элемент в корне — кинжал самого большого элемента турнира. Элементы в листьях (узлы, которые не имеют дочерних узлов) образуют представление не интересующих нас, а истинная часть дерева — структура данных, которая позволяет эффективно проходить посылки на элемент.

Программа 5.19 — рекурсивная программа, которая строит турнир из элементов мушкетеров. Будущий расширенный вариант программы 5.19 использует стратегию "разделяй и властвуй": чтобы построить турнир для заданного элемента, программа выдает лист, содержащий этот элемент, и вычисляет победителя. Чтобы построить турнир для $N > 1$ элементов, в программе используется стратегия "разделяй и властвуй": программа делит все множество элементов листьев, создает турнир для каждого подмножества и выдает победителя из списка с турнира турнира и с элементом, который является кинжалом большего элемента в кинжале победителя турнира.

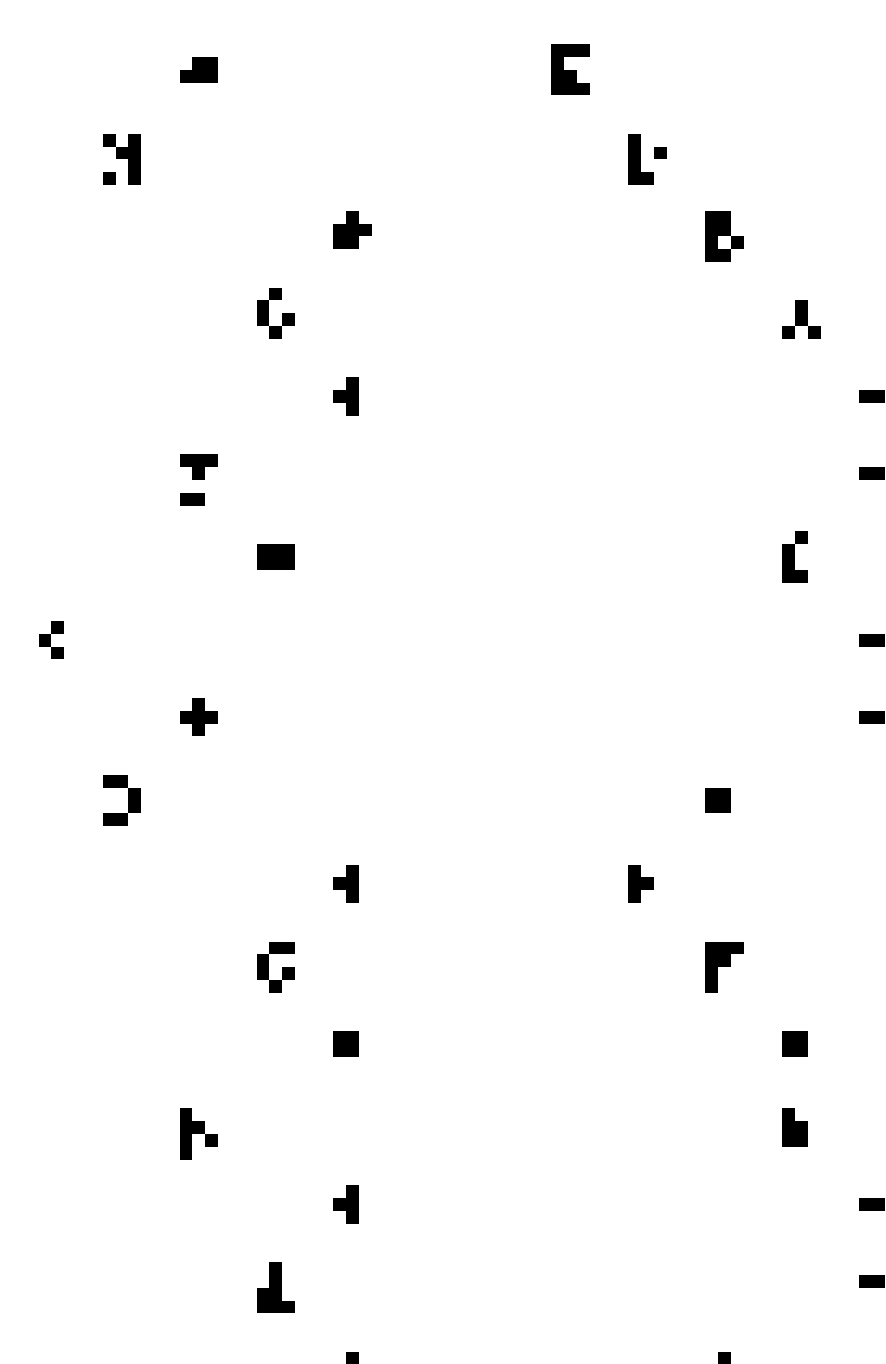


РИСУНОК 5.29 ВЫВОД ДЕРЕВА (ПРИ ПРЯМОМ ОБХОДЕ И ПРЯМОМ ОБХОДЕ)

Вывод, полученный на рисунке выше, получен с помощью функции `drawtree` (см. рис. 5.29). Вспомогательная программа `drawtree` (см. рис. 5.29) и ее вызов в программе `drawtree` (см. рис. 5.29) позволяют вывести представление дерева, которое можно использовать для отладки программ. Вспомогательная программа `drawtree` (см. рис. 5.29) и ее вызов в программе `drawtree` (см. рис. 5.29) позволяют вывести представление дерева, которое можно использовать для отладки программ.

Программа 5.39 Конструирование турнира

Эта рекурсивная функция делит массив $a[1], \dots, a[r]$ на две части $a[1], \dots, a[m]$ и $a[m+1], \dots, a[r]$, строит турниры для двух частей (рекурсивно) и соединяет узлы для обоих частей. Каждая половина узла с описанным построением турнира и помещается под корнем большего элемента в корни двух рекурсивно построенных турниров.

```

int node;
{
    static int n; node = n++;
    node->link = 0;
    {
        int m = (1+x)/2;
        link = new node(a[m]);
        if (m == r) return a;
        x->l = max(a, 1, m);
        x->r = max(a, m+1, r);
        int u = x->l->link, v = x->r->link;
        if (u > v)
            x->link = v; else x->link = u;
        return a;
    }
}

```

На рис. 5.30 приведен пример явной структуры дерева, построенной программой 5.39. Построение таких рекурсивных структур — термин, предпочитаемый аттасканью максимумов явной структуры данных, так это было сделано в программе 5.6, поскольку структура дерева обеспечивает определенную гибкость для выполнения других операций. Каждый элемент служит и символом операции, необходимой для построения турнира. При нахождении двух турниров их можно объединить в один турнир, сделав новый узел, левая ссылка которого указывает на один турнир, а правая на другой, и левая ссылка на двух элементах (помечающих в корнях двух данных турниров) и является наибольшим элементом объединенного турнира. Можно также рассмотреть алгоритмы добавления и удаления элементов и выполнения других операций. Здесь мы не станем рассматривать такие операции, поскольку для явной структуры данных, обладающей хорошей гибкостью, показано в главе 9.

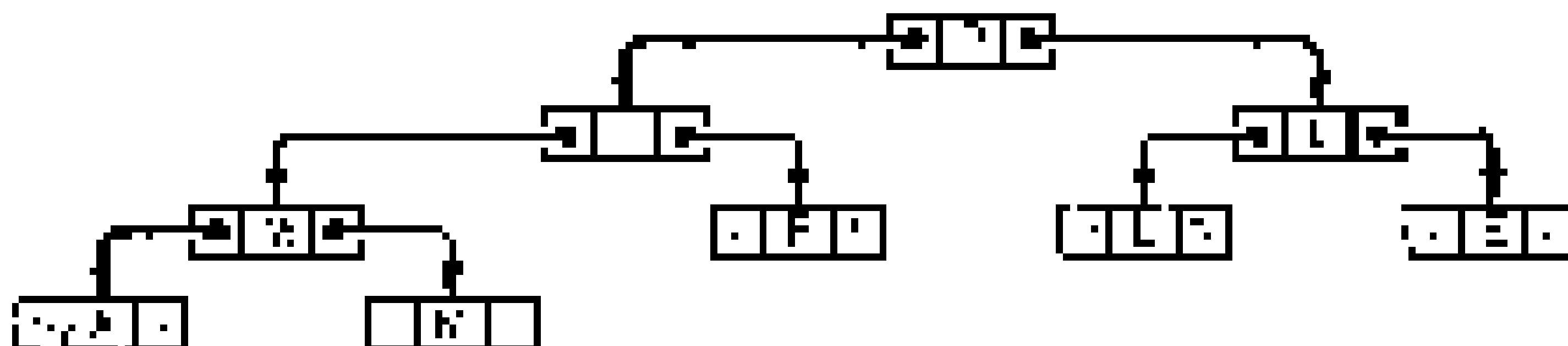


РИСУНОК 5.30 ЯВНОЕ ДЕРЕВО ДЛЯ ОТСЫКАНИЯ МАКСИМУМА (ТУРНИР)

Не стоит забывать описанные структуры дерева, сконструированные программой 5.39, не являются чистыми структурами данных. Взаимосвязанные элементы являются частью структуры данных, которая может быть использована для выполнения других операций. Например, для выполнения операции добавления и удаления элементов и выполнения других операций. Здесь мы не станем рассматривать такие операции, поскольку для явной структуры данных, обладающей хорошей гибкостью, показано в главе 9.

Действительно, реализация с использованием линейной памяти нескольких на свойственном абстрактных типов данных (АТД) запросов, рассмотренных в разделе 5.4, относится к алгоритмам более высокой сложности. В частности, многие из алгоритмов, принятых в разделах 12–15, основываются на деревьях связных узлов (*linked nodes trees*), которые являются сложными деревьями, соответствующими иерархическому плану. Аналогично тому, как явная структура для рис. 5.3 (б) представляется с рекурсивным определением списка узлов (*рис. 5.6*). Эффективность реализации и использование такой структуры заключается в обеспечении эффективности алгоритмов после выполнения большого числа операций вставки, удаления и других операций.

Интересной программой создания бинарного дерева служит реализация вершин программы списка префиксного исчисления, приведенной в разделе 5.1 (рис. 5.4), которую можно считать, представляющее префиксное выражение, а не просто значение *рис. 5.6 (рис. 5.7)*. В программе 5.20 используются также рекурсивные функции, что не противоречит рис. 5.4, но рекурсивные функции позволяют сослаться на дерево, а не значение. Программа создает новый узел дерева для каждого символа в префиксном упоре, который соответствует оператору, и имеет связь со своим поддеревом. В каждой ветви содержится выражение (или константа), которое является частью логического выражения.

Программа 5.20 Создание дерева синтаксического анализа

Алгоритм, описанный в программе 5.20, является одним из самых простых префиксных выражений (см. программу 5.4). Эта программа создает дерево синтаксического анализа из префиксного выражения. Для простоты предполагается, что операторы являются однократными символами. Каждый символ рекурсивно создает новый узел, который является частью дерева. Каждый символ из множества символов данных, если символ является оператором, создает новый узел, а если символ является операндом, то указывается левый и правый указатели на дерево построения (рекурсивно для левой и правой ветвей).

```

char *s; int i;
struct node
{ char *name; node *l, *r;
  node(char *x)
  { name = x; l = 0; r = 0; }
};
typedef node* tnode;
tnode root();
{ tnode n = new node; link n = new node(0);
  if (isop(s[i+1]) || isnum(s[i+1]))
  { n->l = root(); n->r = root(); }
  return n;
}

```



РИСУНОК 5.21 ДЕРЕВО СИНТАКСИЧЕСКОГО АНАЛИЗА

Для дерева, созданного программой 5.20 для префиксного выражения $++x*+1+2+3+4$. Для удобства чтения соответствующий текст префиксного выражения приведен в виде комментариев к листингу. Выводимый текст соответствует тексту префиксного выражения. Если для текущего оператора в листинге (выражение префиксного выражения) имеется указатель на узел, с которым операция, которая должна выполняться в текущем узле, выполняется левым и правым поддеревом узла. Указатель на операцию

В программах текстовики, падающих компьютеров, часто используются такие внутренне представленные программы в виде деревьев, несколько деревьев упираются для достижения тех или иных целей. Например, можно представить словарь, состоящий из непустых строк, представляющих значения, и ассоциировать каждый из них с оценкой выражения, представленного в виде дерева с операторами в узлах. Ибо же можно представить дерево с полными обходами для вывода выражения в канонической форме или дерева с обходом обходом для вывода выражения в постфиксной форме.

В этом разделе было рассмотрено несколько примеров, позволяющих понять, как можно использовать деревья и обработку структурных данных с помощью рекурсивных программ. Чтобы этот раздел стал эффективным, потребовалось учесть пронаблюдательность различных алгоритмов, алгоритмические приемы, неперекрывающиеся элементы, а также другие нюансы. Однако, более подробное рассмотрение программ обработки деревьев откладывается до главы 12, поскольку главы 7—11 деревья представляют, а также другие главы. К рассмотрению других деревьев мы вернемся в главе 12, поскольку они служат основой для многих алгоритмов, рассмотренных в главах 12—15.

Упражнения

5.85 Напишите программу 5.15, чтобы она вывела Postfix-программу, которая рисует дерево в формате, подобном изображению на рис. 5.24, но без лишней загроможденности, представляющей лишние узлы. Используйте операторы `push` и `pop` для рисования дерева в операторе.

`draw (drawn) (x, y) (x1, y1) (x2, y2) (x3, y3) (x4, y4) (x5, y5) (x6, y6) (x7, y7) (x8, y8) (x9, y9) (x10, y10) (x11, y11) (x12, y12) (x13, y13) (x14, y14) (x15, y15) (x16, y16) (x17, y17) (x18, y18) (x19, y19) (x20, y20) (x21, y21) (x22, y22) (x23, y23) (x24, y24) (x25, y25) (x26, y26) (x27, y27) (x28, y28) (x29, y29) (x30, y30) (x31, y31) (x32, y32) (x33, y33) (x34, y34) (x35, y35) (x36, y36) (x37, y37) (x38, y38) (x39, y39) (x40, y40) (x41, y41) (x42, y42) (x43, y43) (x44, y44) (x45, y45) (x46, y46) (x47, y47) (x48, y48) (x49, y49) (x50, y50) (x51, y51) (x52, y52) (x53, y53) (x54, y54) (x55, y55) (x56, y56) (x57, y57) (x58, y58) (x59, y59) (x60, y60) (x61, y61) (x62, y62) (x63, y63) (x64, y64) (x65, y65) (x66, y66) (x67, y67) (x68, y68) (x69, y69) (x70, y70) (x71, y71) (x72, y72) (x73, y73) (x74, y74) (x75, y75) (x76, y76) (x77, y77) (x78, y78) (x79, y79) (x80, y80) (x81, y81) (x82, y82) (x83, y83) (x84, y84) (x85, y85) (x86, y86) (x87, y87) (x88, y88) (x89, y89) (x90, y90) (x91, y91) (x92, y92) (x93, y93) (x94, y94) (x95, y95) (x96, y96) (x97, y97) (x98, y98) (x99, y99) (x100, y100) (x101, y101) (x102, y102) (x103, y103) (x104, y104) (x105, y105) (x106, y106) (x107, y107) (x108, y108) (x109, y109) (x110, y110) (x111, y111) (x112, y112) (x113, y113) (x114, y114) (x115, y115) (x116, y116) (x117, y117) (x118, y118) (x119, y119) (x120, y120) (x121, y121) (x122, y122) (x123, y123) (x124, y124) (x125, y125) (x126, y126) (x127, y127) (x128, y128) (x129, y129) (x130, y130) (x131, y131) (x132, y132) (x133, y133) (x134, y134) (x135, y135) (x136, y136) (x137, y137) (x138, y138) (x139, y139) (x140, y140) (x141, y141) (x142, y142) (x143, y143) (x144, y144) (x145, y145) (x146, y146) (x147, y147) (x148, y148) (x149, y149) (x150, y150) (x151, y151) (x152, y152) (x153, y153) (x154, y154) (x155, y155) (x156, y156) (x157, y157) (x158, y158) (x159, y159) (x160, y160) (x161, y161) (x162, y162) (x163, y163) (x164, y164) (x165, y165) (x166, y166) (x167, y167) (x168, y168) (x169, y169) (x170, y170) (x171, y171) (x172, y172) (x173, y173) (x174, y174) (x175, y175) (x176, y176) (x177, y177) (x178, y178) (x179, y179) (x180, y180) (x181, y181) (x182, y182) (x183, y183) (x184, y184) (x185, y185) (x186, y186) (x187, y187) (x188, y188) (x189, y189) (x190, y190) (x191, y191) (x192, y192) (x193, y193) (x194, y194) (x195, y195) (x196, y196) (x197, y197) (x198, y198) (x199, y199) (x200, y200) (x201, y201) (x202, y202) (x203, y203) (x204, y204) (x205, y205) (x206, y206) (x207, y207) (x208, y208) (x209, y209) (x210, y210) (x211, y211) (x212, y212) (x213, y213) (x214, y214) (x215, y215) (x216, y216) (x217, y217) (x218, y218) (x219, y219) (x220, y220) (x221, y221) (x222, y222) (x223, y223) (x224, y224) (x225, y225) (x226, y226) (x227, y227) (x228, y228) (x229, y229) (x230, y230) (x231, y231) (x232, y232) (x233, y233) (x234, y234) (x235, y235) (x236, y236) (x237, y237) (x238, y238) (x239, y239) (x240, y240) (x241, y241) (x242, y242) (x243, y243) (x244, y244) (x245, y245) (x246, y246) (x247, y247) (x248, y248) (x249, y249) (x250, y250) (x251, y251) (x252, y252) (x253, y253) (x254, y254) (x255, y255) (x256, y256) (x257, y257) (x258, y258) (x259, y259) (x260, y260) (x261, y261) (x262, y262) (x263, y263) (x264, y264) (x265, y265) (x266, y266) (x267, y267) (x268, y268) (x269, y269) (x270, y270) (x271, y271) (x272, y272) (x273, y273) (x274, y274) (x275, y275) (x276, y276) (x277, y277) (x278, y278) (x279, y279) (x280, y280) (x281, y281) (x282, y282) (x283, y283) (x284, y284) (x285, y285) (x286, y286) (x287, y287) (x288, y288) (x289, y289) (x290, y290) (x291, y291) (x292, y292) (x293, y293) (x294, y294) (x295, y295) (x296, y296) (x297, y297) (x298, y298) (x299, y299) (x300, y300) (x301, y301) (x302, y302) (x303, y303) (x304, y304) (x305, y305) (x306, y306) (x307, y307) (x308, y308) (x309, y309) (x310, y310) (x311, y311) (x312, y312) (x313, y313) (x314, y314) (x315, y315) (x316, y316) (x317, y317) (x318, y318) (x319, y319) (x320, y320) (x321, y321) (x322, y322) (x323, y323) (x324, y324) (x325, y325) (x326, y326) (x327, y327) (x328, y328) (x329, y329) (x330, y330) (x331, y331) (x332, y332) (x333, y333) (x334, y334) (x335, y335) (x336, y336) (x337, y337) (x338, y338) (x339, y339) (x340, y340) (x341, y341) (x342, y342) (x343, y343) (x344, y344) (x345, y345) (x346, y346) (x347, y347) (x348, y348) (x349, y349) (x350, y350) (x351, y351) (x352, y352) (x353, y353) (x354, y354) (x355, y355) (x356, y356) (x357, y357) (x358, y358) (x359, y359) (x360, y360) (x361, y361) (x362, y362) (x363, y363) (x364, y364) (x365, y365) (x366, y366) (x367, y367) (x368, y368) (x369, y369) (x370, y370) (x371, y371) (x372, y372) (x373, y373) (x374, y374) (x375, y375) (x376, y376) (x377, y377) (x378, y378) (x379, y379) (x380, y380) (x381, y381) (x382, y382) (x383, y383) (x384, y384) (x385, y385) (x386, y386) (x387, y387) (x388, y388) (x389, y389) (x390, y390) (x391, y391) (x392, y392) (x393, y393) (x394, y394) (x395, y395) (x396, y396) (x397, y397) (x398, y398) (x399, y399) (x400, y400) (x401, y401) (x402, y402) (x403, y403) (x404, y404) (x405, y405) (x406, y406) (x407, y407) (x408, y408) (x409, y409) (x410, y410) (x411, y411) (x412, y412) (x413, y413) (x414, y414) (x415, y415) (x416, y416) (x417, y417) (x418, y418) (x419, y419) (x420, y420) (x421, y421) (x422, y422) (x423, y423) (x424, y424) (x425, y425) (x426, y426) (x427, y427) (x428, y428) (x429, y429) (x430, y430) (x431, y431) (x432, y432) (x433, y433) (x434, y434) (x435, y435) (x436, y436) (x437, y437) (x438, y438) (x439, y439) (x440, y440) (x441, y441) (x442, y442) (x443, y443) (x444, y444) (x445, y445) (x446, y446) (x447, y447) (x448, y448) (x449, y449) (x450, y450) (x451, y451) (x452, y452) (x453, y453) (x454, y454) (x455, y455) (x456, y456) (x457, y457) (x458, y458) (x459, y459) (x460, y460) (x461, y461) (x462, y462) (x463, y463) (x464, y464) (x465, y465) (x466, y466) (x467, y467) (x468, y468) (x469, y469) (x470, y470) (x471, y471) (x472, y472) (x473, y473) (x474, y474) (x475, y475) (x476, y476) (x477, y477) (x478, y478) (x479, y479) (x480, y480) (x481, y481) (x482, y482) (x483, y483) (x484, y484) (x485, y485) (x486, y486) (x487, y487) (x488, y488) (x489, y489) (x490, y490) (x491, y491) (x492, y492) (x493, y493) (x494, y494) (x495, y495) (x496, y496) (x497, y497) (x498, y498) (x499, y499) (x500, y500) (x501, y501) (x502, y502) (x503, y503) (x504, y504) (x505, y505) (x506, y506) (x507, y507) (x508, y508) (x509, y509) (x510, y510) (x511, y511) (x512, y512) (x513, y513) (x514, y514) (x515, y515) (x516, y516) (x517, y517) (x518, y518) (x519, y519) (x520, y520) (x521, y521) (x522, y522) (x523, y523) (x524, y524) (x525, y525) (x526, y526) (x527, y527) (x528, y528) (x529, y529) (x530, y530) (x531, y531) (x532, y532) (x533, y533) (x534, y534) (x535, y535) (x536, y536) (x537, y537) (x538, y538) (x539, y539) (x540, y540) (x541, y541) (x542, y542) (x543, y543) (x544, y544) (x545, y545) (x546, y546) (x547, y547) (x548, y548) (x549, y549) (x550, y550) (x551, y551) (x552, y552) (x553, y553) (x554, y554) (x555, y555) (x556, y556) (x557, y557) (x558, y558) (x559, y559) (x560, y560) (x561, y561) (x562, y562) (x563, y563) (x564, y564) (x565, y565) (x566, y566) (x567, y567) (x568, y568) (x569, y569) (x570, y570) (x571, y571) (x572, y572) (x573, y573) (x574, y574) (x575, y575) (x576, y576) (x577, y577) (x578, y578) (x579, y579) (x580, y580) (x581, y581) (x582, y582) (x583, y583) (x584, y584) (x585, y585) (x586, y586) (x587, y587) (x588, y588) (x589, y589) (x590, y590) (x591, y591) (x592, y592) (x593, y593) (x594, y594) (x595, y595) (x596, y596) (x597, y597) (x598, y598) (x599, y599) (x600, y600) (x601, y601) (x602, y602) (x603, y603) (x604, y604) (x605, y605) (x606, y606) (x607, y607) (x608, y608) (x609, y609) (x610, y610) (x611, y611) (x612, y612) (x613, y613) (x614, y614) (x615, y615) (x616, y616) (x617, y617) (x618, y618) (x619, y619) (x620, y620) (x621, y621) (x622, y622) (x623, y623) (x624, y624) (x625, y625) (x626, y626) (x627, y627) (x628, y628) (x629, y629) (x630, y630) (x631, y631) (x632, y632) (x633, y633) (x634, y634) (x635, y635) (x636, y636) (x637, y637) (x638, y638) (x639, y639) (x640, y640) (x641, y641) (x642, y642) (x643, y643) (x644, y644) (x645, y645) (x646, y646) (x647, y647) (x648, y648) (x649, y649) (x650, y650) (x651, y651) (x652, y652) (x653, y653) (x654, y654) (x655, y655) (x656, y656) (x657, y657) (x658, y658) (x659, y659) (x660, y660) (x661, y661) (x662, y662) (x663, y663) (x664, y664) (x665, y665) (x666, y666) (x667, y667) (x668, y668) (x669, y669) (x670, y670) (x671, y671) (x672, y672) (x673, y673) (x674, y674) (x675, y675) (x676, y676) (x677, y677) (x678, y678) (x679, y679) (x680, y680) (x681, y681) (x682, y682) (x683, y683) (x684, y684) (x685, y685) (x686, y686) (x687, y687) (x688, y688) (x689, y689) (x690, y690) (x691, y691) (x692, y692) (x693, y693) (x694, y694) (x695, y695) (x696, y696) (x697, y697) (x698, y698) (x699, y699) (x700, y700) (x701, y701) (x702, y702) (x703, y703) (x704, y704) (x705, y705) (x706, y706) (x707, y707) (x708, y708) (x709, y709) (x710, y710) (x711, y711) (x712, y712) (x713, y713) (x714, y714) (x715, y715) (x716, y716) (x717, y717) (x718, y718) (x719, y719) (x720, y720) (x721, y721) (x722, y722) (x723, y723) (x724, y724) (x725, y725) (x726, y726) (x727, y727) (x728, y728) (x729, y729) (x730, y730) (x731, y731) (x732, y732) (x733, y733) (x734, y734) (x735, y735) (x736, y736) (x737, y737) (x738, y738) (x739, y739) (x740, y740) (x741, y741) (x742, y742) (x743, y743) (x744, y744) (x745, y745) (x746, y746) (x747, y747) (x748, y748) (x749, y749) (x750, y750) (x751, y751) (x752, y752) (x753, y753) (x754, y754) (x755, y755) (x756, y756) (x757, y757) (x758, y758) (x759, y759) (x760, y760) (x761, y761) (x762, y762) (x763, y763) (x764, y764) (x765, y765) (x766, y766) (x767, y767) (x768, y768) (x769, y769) (x770, y770) (x771, y771) (x772, y772) (x773, y773) (x774, y774) (x775, y775) (x776, y776) (x777, y777) (x778, y778) (x779, y779) (x780, y780) (x781, y781) (x782, y782) (x783, y783) (x784, y784) (x785, y785) (x786, y786) (x787, y787) (x788, y788) (x789, y789) (x790, y790) (x791, y791) (x792, y792) (x793, y793) (x794, y794) (x795, y795) (x796, y796) (x797, y797) (x798, y798) (x799, y799) (x800, y800) (x801, y801) (x802, y802) (x803, y803) (x804, y804) (x805, y805) (x806, y806) (x807, y807) (x808, y808) (x809, y809) (x810, y810) (x811, y811) (x812, y812) (x813, y813) (x814, y814) (x815, y815) (x816, y816) (x817, y817) (x818, y818) (x819, y819) (x820, y820) (x821, y821) (x822, y822) (x823, y823) (x824, y824) (x825, y825) (x826, y826) (x827, y827) (x828, y828) (x829, y829) (x830, y830) (x831, y831) (x832, y832) (x833, y833) (x834, y834) (x835, y835) (x836, y836) (x837, y837) (x838, y838) (x839, y839) (x840, y840) (x841, y841) (x842, y842) (x843, y843) (x844, y844) (x845, y845) (x846, y846) (x847, y847) (x848, y848) (x849, y849) (x850, y850) (x851, y851) (x852, y852) (x853, y853) (x854, y854) (x855, y855) (x856, y856) (x857, y857) (x858, y858) (x859, y859) (x860, y860) (x861, y861) (x862, y862) (x863, y863) (x864, y864) (x865, y865) (x866, y866) (x867, y867) (x868, y868) (x869, y869) (x870, y870) (x871, y871) (x872, y872) (x873, y873) (x874, y874) (x875, y875) (x876, y876) (x877, y877) (x878, y878) (x879, y879) (x880, y880) (x881, y881) (x882, y882) (x883, y883) (x884, y884) (x885, y885) (x886, y886) (x887, y887) (x888, y888) (x889, y889) (x890, y890) (x891, y891) (x892, y892) (x893, y893) (x894, y894) (x895, y895) (x896, y896) (x897, y897) (x898, y898) (x899, y899) (x900, y900) (x901, y901) (x902, y902) (x903, y903) (x904, y904) (x905, y905) (x906, y906) (x907, y907) (x908, y908) (x909, y909) (x910, y910) (x911, y911) (x912, y912) (x913, y913) (x914, y914) (x915, y915) (x916, y916) (x917, y917) (x918, y918) (x919, y919) (x920, y920) (x921, y921) (x922, y922) (x923, y923) (x924, y924) (x925, y925) (x926, y926) (x927, y927) (x928, y928) (x929, y929) (x930, y930) (x931, y931) (x932, y932) (x933, y933) (x934, y934) (x935, y935) (x936, y936) (x937, y937) (x938, y938) (x939, y939) (x940, y940) (x941, y941) (x942, y942) (x943, y943) (x944, y944) (x945, y945) (x946, y946) (x947, y947) (x948, y948) (x949, y949) (x950, y950) (x951, y951) (x952, y952) (x953, y953) (x954, y954) (x955, y955) (x956, y956) (x957, y957) (x958, y958) (x959, y959) (x960, y960) (x961, y961) (x962, y962) (x963, y963) (x964, y964) (x965, y965) (x966, y966) (x967, y967) (x968, y968) (x969, y969) (x970, y970) (x971, y971) (x972, y972) (x973, y973) (x974, y974) (x975, y975) (x976, y976) (x977, y977) (x978, y978) (x979, y979) (x980, y980) (x981, y981) (x982, y982) (x983, y983) (x984, y984) (x985, y985) (x986, y986) (x987, y987) (x988, y988) (x989, y989) (x990, y990) (x991, y991) (x992, y992) (x993, y993) (x994, y994) (x995, y995) (x996, y996) (x997, y997) (x998, y998) (x999, y999) (x1000, y1000) (x1001, y1001) (x1002, y1002) (x1003, y1003) (x1004, y1004) (x1005, y1005) (x1006, y1006) (x1007, y1007) (x1008, y1008) (x1009, y1009) (x1010, y1010) (x1011, y1011) (x1012, y1012) (x1013, y1013) (x1014, y1014) (x1015, y1015) (x1016, y1016) (x1017, y1017) (x1018, y1018) (x1019, y1019) (x1020, y1020) (x1021, y1021) (x1022, y1022) (x1023, y1023) (x1024, y1024) (x1025, y1025) (x1026, y1026) (x1027, y1027) (x1028, y1028) (x1029, y1029) (x1030, y1030) (x1031, y1031) (x1032, y1032) (x1033, y1033) (x1034, y1034) (x1035, y1035) (x1036, y1036) (x1037, y1037) (x1038, y1038) (x1039, y1039) (x1040, y1040) (x1041, y1041) (x1042, y1042) (x1043, y1043) (x1044, y1044) (x1045, y1045) (x1046, y1046) (x1047, y1047) (x1048, y1048) (x1049, y1049) (x1050, y1050) (x1051, y1051) (x1052, y1052) (x1053, y1053) (x1054, y1054) (x1055, y1055) (x1056, y1056) (x1057, y1057) (x1058, y1058) (x1059, y1059) (x1060, y1060) (x1061, y1061) (x1062, y1062) (x1063, y1063) (x1064, y1064) (x1065, y1065) (x1066, y1066) (x1067, y1067) (x1068, y1068) (x1069, y1069) (x1070, y1070) (x1071, y1071) (x1072, y1072) (x1073, y1073) (x1074, y1074) (x1075, y1075) (x1076, y1076) (x10`

5.8 Обход графа

В качестве иллюстративного примера в этой главе рассмотрим одну из наиболее важных рекурсивных программ: рекурсивный обход графа или поиск в глубину (depth first search). Этот метод симметричного посещения всех узлов графа — непосредственное обобщение обхода дерева, рассмотренных в разделе 5.6, и он служит основой для многих базовых алгоритмов обхода графов (см. часть 7). Это также рекурсивный алгоритм. Начиная с любого узла u , мы

- посещаем u ;
- (рекурсивно) посещаем каждый (соседний) узел, связанный с u .

Если граф является связным, со временем будут посещены все узлы. Программа 5.21 реализует решение этой рекурсивной процедуры.

Программа 5.21 Поиск в глубину

Для узла u дана adj — граф (матрица смежности), связанный с узлом u (а также сам этот узел посещенный). Узел u (рекурсивно) посещается, если все узлы-соседи v узла u в списке $adj[u][v]$ для узла u .

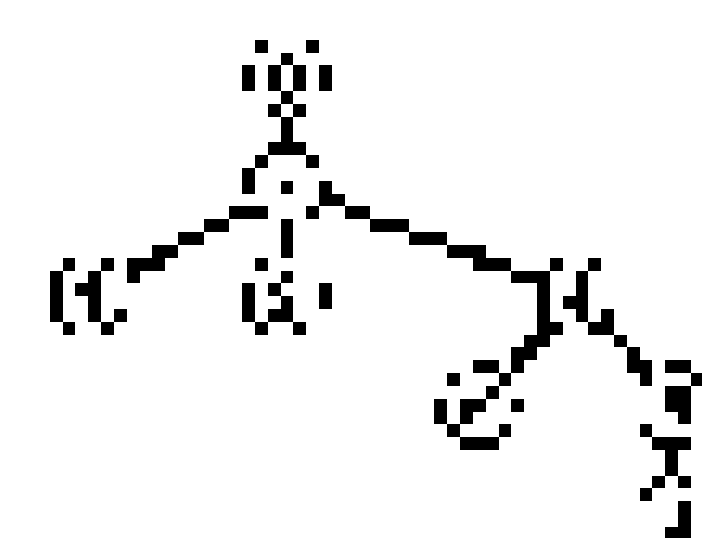
```
void traverse(int u, void visit(int))
{ visit(u); visited[u] = 1;
  for (int i = adj[u].size() - 1; i >= 0; i--) {
    if (!visited[i]) traverse(i, visit);
  }
```

Например, предположим, что используется представление в виде списка смежности узлов, описанных в примере графа на рис. 5.15. На рис. 5.22 показаны последовательности вызовов, соответствующих поиску в глубину в этом графе, с пометкой о посещенности ребер графа. Количество вызовов показано в нижней части рис. 5.22. При прохождении каждого из ребер графа возможны два варианта: если ребро приводит к уже посещенному узлу, мы игнорируем его; если это приводит к еще не посещенному узлу, мы переходим к нему через рекурсивный вызов. Набор всех рассмотренных таким образом ребер образует остовное дерево графа.

Различие между поиском в глубину и обходом обходной переменной (см. программу 5.4) состоит в том, что необходимо явно исключать посещения уже посещенных узлов. В дереве мы никогда не встречаемся с такими узлами. Действительно, если граф не имеет циклов, рекурсивный поиск в глубину, начинающийся с узла u , эквивалентен прямому обходу.

Лемма 5.16 Если, выполняясь для некоторого поиска в глубину в графе с V вершинами и E ребрами, последовательно $V + E$, если дополнительно предположить, что граф не имеет циклов.

В представлении в виде списков смежности каждому ребру графа это соответствует один узел в списке, а каждый пер-



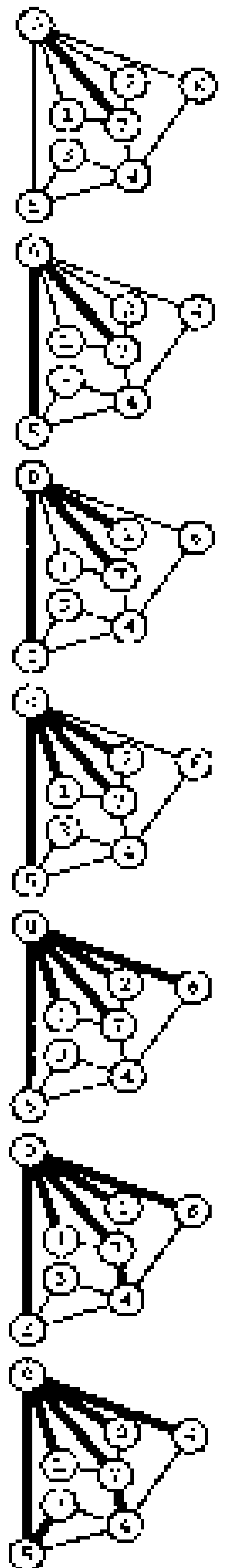
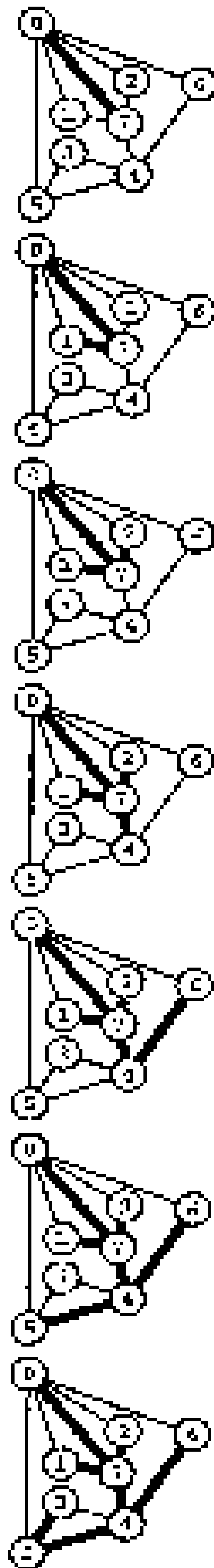
```
void
call traverse(int u,
void visit(int u)
{ visit(u);
  for (int i = adj[u].size() - 1; i >= 0; i--) {
    if (!visited[i]) traverse(i, visit);
  }
}
```

РИСУНОК 5.22 ВЫЗОВЫ ФУНКЦИЙ ПОИСКА В ГЛУБИНУ

Эта последовательность вызовов функций поиска в глубину для графа, показанного на рис. 5.15. Деревья, соответствующие различным этапам рекурсивного поиска (начиная с узла 0) показаны в правой части рисунка.

РИСУНОК 5.31 ПОИСК В ГЛУБИНУ И ПОИСК В ШИРИНУ

Для поиска в глубину (слева) первый найденный путь из вершины 0 до вершины 6 сохраняется, а остальные найденные пути отбрасываются. При поиске в ширину (справа) до момента нахождения пути наименьшей длины все найденные пути сохраняются.



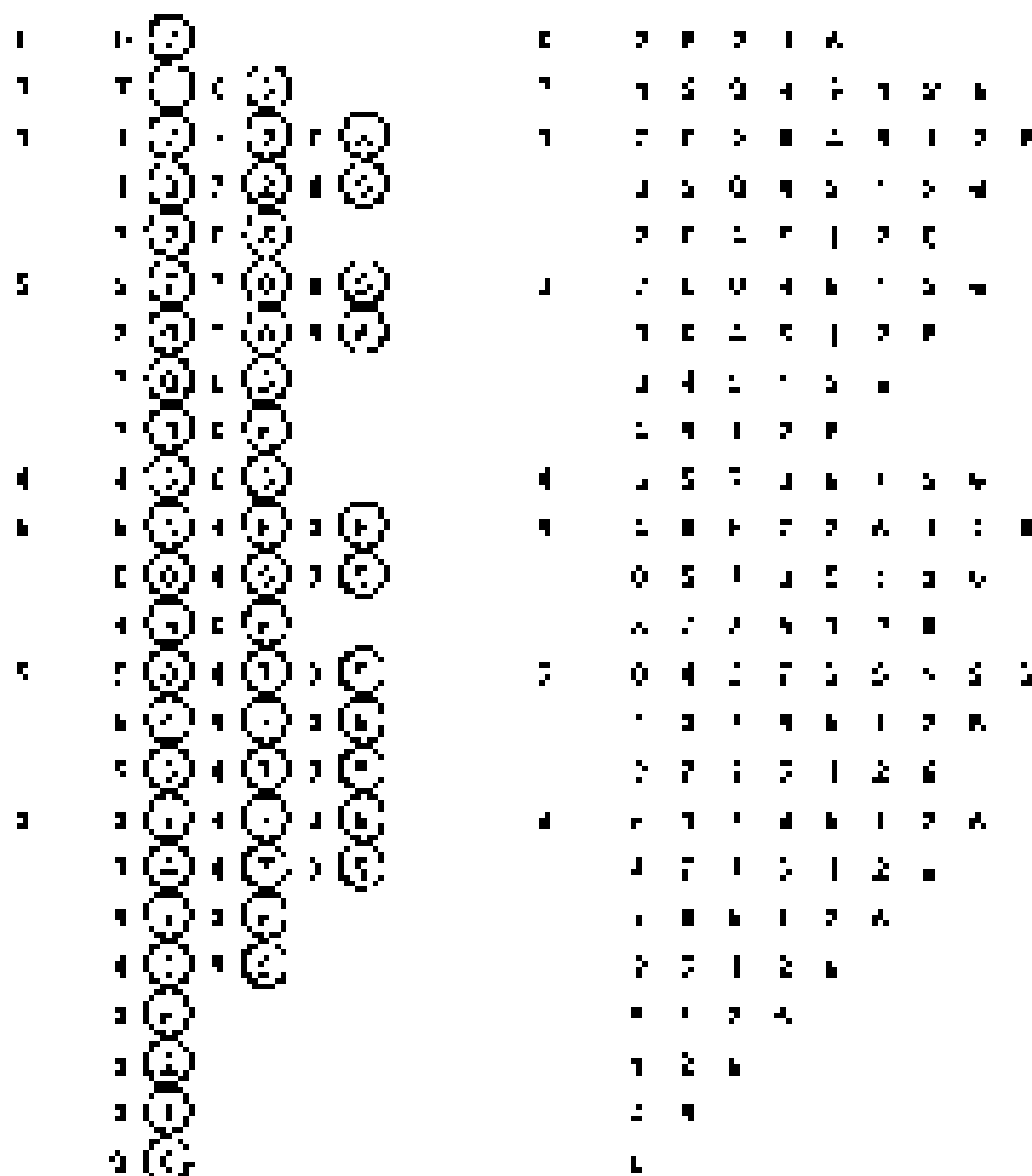
спина графа соответствует единь указателю на начало списка. Поиск в глубину завершается каждый раз, как только будет выполнено одно из двух.

Поскольку время, необходимое для построения представления в виде списка связано с пространственностью ребер (см. главу 3.19), то для построения графа $V + E$ поиск в глубину обеспечивает линейное (по отношению к затратам на построение решетки заданной структуры) время. Однако для очень больших графов решения илпос-блп все же могут оказаться предпочтительнее, поскольку для представления такого графа требуется объем памяти, пропорциональный E , а для решения этих решений илпос-блп необходим объем памяти, который пропорционален только V .

Поскольку поиск как это было описано в главе 3.19 в общем случае можно определить метод обхода графа при котором используется один стек, как показано на рис. 3.24. Можно представить себе абстрактный стек, содержащий двоек (x, y) , где x — указатель на узел, связанный для этого узла. Когда стек вышлплизмуется, указанным узлом, а указатель — первым элементом стека связности для этого узла. Алгоритм поиска в глубину, использующий стек (x, y) в том, в котором сейчас находится узел в верхней части стека (если он еще не был посещен) и хранится узел, указанный текущим указателем стека связности. Далее, ссылка на узел связности обновляется следующим узлом (с выталкиванием узла). Если достигнут конец стека связности V , наконец, данный стек для хранения узла заменяется со ссылкой на первый узел x в списке связности.

РИСУНОК 3.24 ДИНАМИКА СТЕКА ПОИСКА В ГЛУБИНУ

Стек, реализованный указателем, в действительности представляет собой набор указателей на элементы списка связности для этого узла. Для поиска узла, связанного с указателем, первым элементом стека связности для этого узла, алгоритм должен пройти по списку на первом уровне и найти элемент списка — пункт 7. Каждый раз, когда алгоритм достиг нового узла, он добавляет его в стек, заменяя предыдущий элемент. Таким образом, когда алгоритм достиг узла, он добавляет его в стек, заменяя предыдущий элемент. Таким образом, когда алгоритм достиг узла, он добавляет его в стек, заменяя предыдущий элемент. Таким образом, когда алгоритм достиг узла, он добавляет его в стек, заменяя предыдущий элемент.



Для же. полезно тому, для это было сказано для обхода дерева, можно создать стек, содержащий только соседние узлы. Когда стек не пуст, извлекаем начальный узел, мы входим в цикл, в котором посещаем узел и порождаем часть стека (если он еще не был посещен), затем записываем в стек все соседние узлы и проходим. Рекурсия § 34 иллюстрирует, что для типичного примера графа оба метода эквивалентны поиску в глубину, причем эффективность сохраняется и в общем случае.

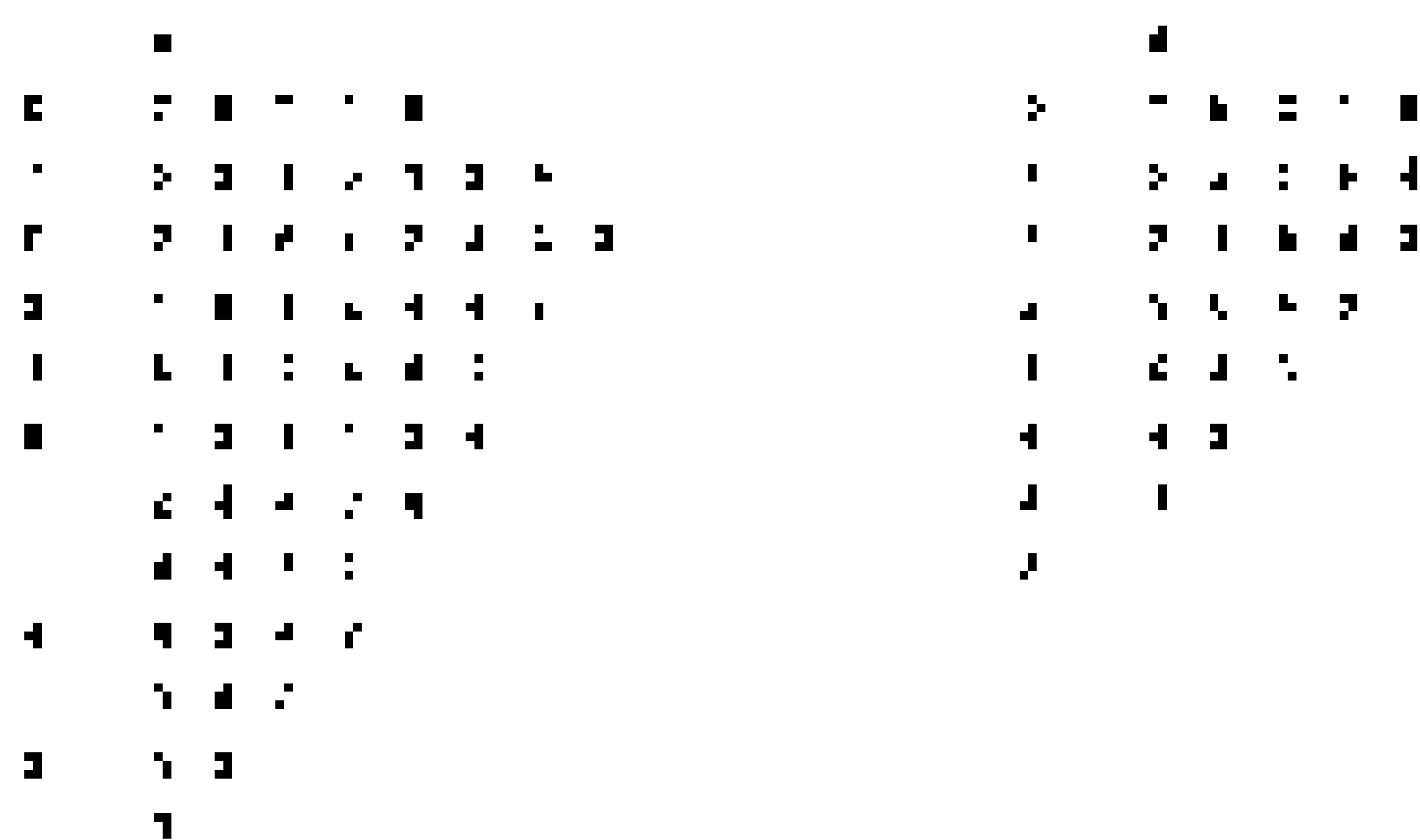
Алгоритм посещения вершины узла и вычисления всех его соседей — простейшая формулировка поиска в глубину, на рис. 5.34 показано, что этому методу присущ недостаток возможного отсутствия в стеке нескольких узлов каждого узла. Это происходит даже в случае простого графа, состоящего из одного узла, который должен быть помещен в стек, и если да, то отсюда от аннотации в стек также узел. На рисунке 5.34 упомянутой проблемы можно избежать, реализовав стек, который запрещает дублирование из стека приписания стрелки "добавления старого элемента": на каждом шаге — от вершины стека — только одна добавленная вершина, все остальные вершины просто удаляются из стека.

Циклическая структура стека для поиска в глубину, показанная на рис. 5.34, основана на том, что у каждого списка смежных узлов появляется в стек в том же порядке, что и в списке. Для получения этого порядка для типичного списка смежности при вычислении узлов по порядку, начиная с заданного, мы записываем в стек начальный узел, затем предполагаем, что, если это, чтобы ограничить размер стека циклом, вершить при обнаружении посещенных узлов в том же порядке, как и при поиске в глубину, необходимо использовать стек, в котором мы следуем правилу "добавляем старого элемента". Для посещения узла и том же порядке, что и при поиске в глубину, не имеет значения, обход стека поначалу можно избежать и непосредственно сформулировать некоррumpцированный метод обхода графа в использовании списка, который выглядит так: 1. Если начальная вершина не пуст, то узел узла мы записываем в стек, в котором посещаем узел на выходе стека, затем добавляем все соседние узлы, принадлежащие к стек каждой узла (если он еще не был посещен) и вставляя реализацию стека, которая за счет применения стратегии "кларификация нового элемента" запрещает дублирование. Этот алгоритм обозначивает посещение всех узлов графа, начиная с поиска в глубину, но не является рекурсивным.

Алгоритм, представленный в предыдущем абзаце, заслуживает внимания, поскольку можно было бы воспользоваться любым обобщенным абстрактным типом данных очереди и все же получить тот же результат (разница между стекерскими реализациями дерева). Например, если вместо стека использовать очередь, то получится очередь узлов, который эквивалентен обходу дерева по уровням. Параграф 5.35 — реализация этого метода (при условии, что используются реализации с приоритетной очередью, подобная рис. 5.12). Пример этого алгоритма в работе показан на рис. 5.35. В главе 6 будут рассмотрены многочисленные альтернативы обработки графов, основанные на более сложных структурах данных, а также различные типы данных очереди.

РИСУНОК 5.35 ДИНАМИКА ОЧЕРЕДИ ПОИСКА В ШИРИНУ

Рисунок иллюстрирует работу очереди, выполняющей поиск в ширину. Начиная с узла u , посещаются все узлы, находящиеся на расстоянии 1 от u , затем все узлы, находящиеся на расстоянии 2 от u , и т.д. Узлы помечаются, причем ширину поиска можно считать значением d узла u . Значения помечаются для T узлов, находящихся на расстоянии d от узла u . Если узел v находится на расстоянии d от узла u , то узел v помечается значением $d + 1$. Узлы, помеченные значением $d + 1$, образуют очередь для следующего шага поиска.



Программа 5.22 Поиск в ширину

Чтобы посетить в графе все узлы, связанные с узлом k , k является узлом очереди ПГО, узлы в очереди q помечаются, а затем для каждого получается следующий узел x , если он еще не был посещен, и добавив x в очередь. Затем выполняются все необходимые узлы x в очереди. Наконец, процесс продолжается для тех пар, пока очередь не пуста.

```
void traverse(int k, void visit(int))
{
    queue<int> q(1000);
    q.put(k);
    while (!q.empty())
    {
        int x = q.get();
        visit(x);
        for (int i = 0; i < G[x].size(); i++)
        {
            int y = G[x][i];
            if (!visited[y]) q.put(y);
        }
    }
}
```

Для исследования данной задачи методом поиска в ширину и в глубину посещаются все узлы графа, но их способ посещения зависит от выбора конкретной формулы, что демонстрирует рис. 5.16. Поиск в ширину полезен для нахождения кратчайшего пути во всех случаях, а также поиска всех территорий, поиск в глубину соответствует единственному последовательному, который как можно дальше проникает внутрь непонятной территории, возвращаясь только в случае, если не удается никуда пойти. Поиск в ширину является первым шагом, позволяющим существовать ряду токовых областей компьютерных наук, а не только в контексте поиска в графах.

Упражнения

5.92 (продолжение упражнения 5.91). Используя динамический массив (рис. 5.33 (левая) и 5.34 (правая)), покажите, как происходит посещение узлов в графе, построенном для показанной на рис. 5.3 (1-4, 2-5, 3-6, 4-6, 5-9 и 6-9) и 5.12 (см. упражнение 3.70), при рекурсивном поиске в глубину.

5.93 (продолжение упражнения 5.91). Используя динамический массив (рис. 5.33 (левая) и 5.34 (правая)), покажите, как происходит посещение узлов в графе, построенном для

хосить за рамки простой непосредственной рекурсивной реализации: для других будут рассмотрены методы, произведённые от абстрактных идей сортировки и выбора лучших решений.

В этой книге основное внимание уделено практическим аспектам построения рекурсивных программ и структур данных. Наша цель — применение рекурсии для создания изящных и эффективных решений. Для достижения этой цели необходимо будет учитывать особенности, сопряжённые с использованием простых приёмов, которые ведут к эффективному решению. Многие функции для сортировки и выбора являются сложными. Несмотря на этот недостаток, рекурсивные программы и структуры данных весьма привлекательны, поскольку часто они представляют интуитивные идеи, которые позволяют убедиться в правильности и эффективности разработанных программ.

На протяжении всей книги деревья используются как для сортировки, так и для динамических списков программ, так и для хранения данных. Структуры данных. В главе 11 по мере развития книги основное внимание уделяется манипулированию списками, деревьями и структурами. Свойства, описанные в этой главе, предоставляют основу для информации, которая требуется для эффективного применения списков, деревьев и структур.

Несмотря на свою историческую роль в разработке алгоритмов, рекурсия — это не только не самый лучший способ жизни. Как было показано при исследовании алгоритмов, деревья и графы, алгоритмы с использованием стека (которые рекурсивны во своём приходе) — это единственная возможность при необходимости управлять данными с множеством вершин. Эффективная реализация разработки алгоритмов для решения многих задач заключается в использовании обобщённых отношений с применением очередей, отличающихся от стеков, которые предоставляют возможность выбрать следующую задачу в соответствии с каким-либо более субъективным критерием, нежели простой выбор чего-либо из списка. Структуры данных и алгоритмы, которые эффективно поддерживают такие очереди — основные темы главы 9, а следующие примеры их применения мы встретимся во время исследования алгоритмов обработки графов в главе 10.

Ссылки для части 2

Существует множество учебников для начинающих, посвящённых структурам данных. Например, в книге Стивенса (Stevens) охвачены основные структуры, абстракции данных, стеки и очереди, распределение памяти и создание программ, основанных на этих моделях, чем эта. Конечно, классические книги Хорнгуэя и Ричи (Hornigay Richi) и Стратула (Stratula) — бесценные источники подробной информации по различным C и C++. Книга Мейерса (Meiers) также содержит полную историю о реализации C++.

Разработчик Rodhouse, вероятно, не мог также и предполагать, что разработанный язык будет представлять интерес для людей, которые изучают основы алгоритмов и структур данных. Сам по себе этот язык не представляет особой сложности, а связанные руководства на тему — специально подготовлены.

Правильная реализация этих методов/функций подробно, со множеством примеров описывается в книге Уинстона (Winston). Эти книги — замечательный справочник для тех программистов, которые намерены создавать устойчивый и надежный код для реальных систем.

Книги Кнута (Knuth), а особенно 1-й и 2-й томы, являются авторитетным источником информации по самым элементарным структурам данных. Книга Тарна (Tarn) и Якоба (Yacob) и Тарна (Tarn) охватывает более обширную информацию, дополненную структурными графическими примерами. Седжвик (Sedgewick) и Флажелет (Flajolet) подробно описывают математические основы рекурсии.

Adobe Systems Incorporated, *Acrobat Language Reference Manual*, second edition, Addison-Wesley, Reading, MA, 1993.

A. Peter Yates and G. H. Gornes, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.

D. H. Manber, *C Structures and Implementation: Techniques for Creating Reusable Software*, Addison-Wesley, 1997.

H. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.

D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, third edition, Addison-Wesley, Reading, MA, 1997; Volume 2: Seminumerical Algorithms, third edition, Addison-Wesley, Reading, MA, 1998; Volume 3: Sorting and Searching, second edition, Addison-Wesley, Reading, MA, 1998.

S. Meyers, *Design in C++*, second edition, Addison-Wesley, Reading, MA, 1996.

S. Meyers, *More Effective C++*, Addison-Wesley, Reading, MA, 1996.

R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.

T. A. Sandich, *Data Structures, Algorithms, and Software Principles in C*, Addison-Wesley, 1995.

B. Stroustrup, *The C++ Programming Language*, third edition, Addison-Wesley, Reading, MA, 1997.

Сортировка

Входит в состав:

6	Элементарные методы сортировки
7	Быстрая сортировка
8	Слияние и сортировка слиянием
9	Очереди по приоритетам и линейная сортировка
10	Поразрядная сортировка
11	Методы сортировки специального назначения

Элементарные методы сортировки

В качестве типичного примера рассмотрим область алгоритмов сортировки. Многие небольшие алгоритмы существуют, которые непосредственно используются для сортировки файлов небольшого размера — это для сортировки файлов со специальной структурой. Мы рассмотрим несколько причин для подробного изучения этих простых алгоритмов сортировки. Прежде всего, они представляют собой контекст, в рамках которого можно изучить терминологию и базовые концепции алгоритмов сортировки. Это позволит создать соответствующие предположения для изучения более сложных алгоритмов. Во-вторых, эти простые методы во многих приложениях сортировки показались себя, по сути дела, более эффективными, чем многие универсальные методы. В-третьих, некоторые из простых методов позволяют расширение в более эффективные универсальные методы или же могут оказаться полезными в более сложных эффективных совершенных методах сортировки.

Цель настоящей главы заключается не только в ознакомлении читателя с базовыми методами сортировки, но и в создании таких структур, в рамках которых можно было бы изучать сортировку и в последующую главу. Мы рассмотрим различные ситуации, которые будут представлять применение того или иного алгоритма сортировки, рассмотрим различные виды файлов, и также рассмотрим различные способы сравнения методов сортировки и изучения их времени.

Начнем с рассмотрения простой процедуры тестирования методов сортировки, которая объясняет контекст, позволяющий выработать алгоритмы, необходимые для того, чтобы исследовать их эффективность. Мы также про-

или, напротив, базовые свойства различных методов сортировки; их очень важно иметь при оценке эффективности и применимости того или иного алгоритма к различным конкретным приложениям. Затем мы подробно рассмотрим реализацию трех элементарных методов сортировки: выбора, сортировки вставками и пузырьковой сортировки. После этого будут подробно описаны рабочие характеристики усовершенствованных алгоритмов. Далее мы рассмотрим сортировку методом Хоара, которая не очень-то подходит для характеристики 'элементарных', однако она достаточно просто реализуется и имеет много общего с сортировкой методом вставки. Упомянем в заключение некоторые свойства сортировки методом Шелла, на котором основан изученный алгоритм разбивки интервальных типов данных, наряду с программой, которая рассматривалась в главах 3 и 4 и которую представляем для рассмотрения алгоритмов сортировки различных типов файлов данных, какие встречаются на практике. Затем мы рассмотрим методы сортировки по сравнению с тем, как на данных, а также методы сортировки с помощью циклов. Данная глава завершается обсуждением внешнего сортирования метода, который целесообразно использовать, когда известно, что данные представляют значительные объемы информации.

Многие из возможных приложений сортировки часто основаны на предпочтении произвольных алгоритмов. Во-первых, очень часто программа сортировки выполняется много раз над тем же или как увеличивающимся числом раз. После того как удалось 'решить' проблему сортировки для некоторого набора данных, и достигнута готовность к программе сортировки в приложениях, которые манипулируют этими данными, обычно. Если элементарная сортировка работает на меньшей части данных приложения, лучше использовать обработку данных — например, эмитирование данных или их вывод из памяти — то становится необходимой в поиске более быстрого методов сортировки. Если много сортируемых элементов не очень большое (скажем, не превышает нескольких сотен элементов), можно просто воспользоваться простыми методами и не беспокоиться по поводу того, как работает интерфейс для системной сортировки, или как написать и оптимизировать программу, которая использует какой-нибудь стандартный метод сортировки. Но тогда, элементарные методы сортировки сортируют файлы небольшого размера (вместимость на нескольких десятках элементов) — сложные алгоритмы в общем случае обуславливают более эффективные затраты ресурсов, а это приводит к тому, что по самым небольшим размерам она работает медленнее элементарных методов сортировки. Эта проблема не появляется в файлах большого размера, или тем пор, пока не возникает необходимость сортировки более или менее файлов большого размера, особенно следует иметь в виду, что приложения с подобными рода требованиями встречаются достаточно часто. Другими типами файлов, сортировка которых существенно упрощается, являются файлы, состоящие (или уже) завершённой сортировкой или файлы, которые содержат большое число дублированных записей. Далее можно будет убедиться в том, что некоторые методы не только просты, но особенно эффективны при сортировке хорошо структурированных файлов.

Как правило, для сортировки случайного упорядоченного файла из N элементов с применением элементарных методов, особенно обсуждаемых в данной главе, требуется время, пропорциональное N^2 . Если N велико, то такое время выполнения сортировки может оказаться вообще неприемлемым. Как только что было отмечено, эти алгоритмы применимы также для сортировки файлов небольшого размера, а также в ряде дру-

гих специальных случаев, по эффективности могут преподать более сложное метод сортировки. Однако методы, которые используются в настоящей главе, не годятся для сортировки файлов различных размеров в транзитивной организации, поскольку время их сортировки будет несомненно большим, даже если она выполняется на сверхбыстродействующих компьютерах. В этом плане заслуживающим внимания предложением может служить сортировка методом Шелла (см. раздел 6.6), для которой при заданном N требуется гораздо меньше, чем N^2 шагов, при этом можно утверждать, что данный метод является одним из наилучших для сортировки файлов различных размеров и для ряда других специальных случаев.

6.1. Правила игры

Прежде чем переходить к рассмотрению конкретных алгоритмов, полезно обобщить общую терминологию и основные принципы построения алгоритмов сортировки. Мы будем рассматривать методы сортировки файлов элементов, обобщая понятие. Эти понятия являются естественными абстракциями для современных сред программирования. Ключи, которые — суть лишь часть (чаще всего очень небольшая часть) элементов, используются для выражения сортировки. Цель метода сортировки заключается в перепорядочении элементов таким образом, чтобы их ключи следовали в соответствии с чем-то определенным и ровным (обычно это цифровой или алфавитный порядок). Специфические характеристики ключей и элементов в разных приложениях могут существенно различаться друг от друга, однако абстрактное описание размещения ключей и связанных с ними информации в определенном порядке и представляет собой суть проблемы сортировки.

Если сортируемый файл поддается обработке в определенной памяти, то рассматриваемый в этом случае метод сортировки называется сортировкой. Сортировка файлов, хранящихся на магнитной ленте или диске, называется *меморной*. Основное различие между этими двумя методами заключается в том, что в первом случае сортировка доступа к любому элементу не представляет трудностей, а во втором же в зависимости от сортировки возможен только один единственный метод доступа или, по меньшей мере, доступ к блокам больших размеров. Некоторые из методов сортировки рассматриваются в главе 6, однако основная часть рассмотренных алгоритмов принадлежит к категории *внутренней сортировки*.

Мы будем рассматривать как массивы, так и связанные списки. Как проблема сортировки массивов, так и проблема сортировки связанных списков представляют собой универсальную проблему в процессе разработки собственных алгоритмов или связанных с некоторыми базовыми алгоритмами, для которых лучше всего подходит равномерное распределение элементов, для других же типов файлов подходит структура связанных списков. Некоторые из классических методов обобщают столь важную особенность, что могут одинаково эффективно применяться как к массивам, так и к связным спискам, в то же время другие являются исключительно подходящими для каких-то видов или указанным выше объектов сортировки. Другие методы ограничены доступом также только определенным способом к элементам.

Для начала стоит акцентировать внимание на сортировке массивов. Примеры 6.1 может служить иллюстрацией соглашения, которые будут соблюдаться во всех реали-

зависит. Она состоит из управляющей программы (обычно это программа драйвера), которая генерирует массив, состоящий либо из заранее заданных либо генерируемых случайных чисел (соответствующий режим работы программы задается параметром `argc`). Затем эта программа вызывает функцию сортировки, которая сортирует эти числа в определенном порядке, а в завершение выводит на печать результаты сортировки.

Программа 6.1 Пример сортировки массива с помощью управляющей программы

Данная программа служит для демонстрации тех или иных возможностей, касающихся массивов. Функция `main` — это драйвер, который инициализирует массив целых чисел (случайными значениями либо значениями, заданными в файле `data.in`). Затем функция `sort` с целью выполнения сортировки заданного массива, после чего выводит на печать упорядоченный результат сортировки.

Пользователь может заставить данную программу работать с помощью элементов, принадлежащих к различным типам данных, для которых определены символы объявления и объявления. В данном примере символ функции `sort` представляет собой частный случай сортировки пузырьком (см. раздел 6.3, в котором подробно описаны алгоритмы сортировки). Массив `a` (одномерный массив `int a[N]`) не является глобальной функцией, которая сортирует два элемента и поминимальности. Он выводит массив `a` на экран, а затем вызывает функцию `sort` из модуля `sort.c`.

Можно почти соответствующим образом изменить программу драйвера, чтобы она могла сортировать массив `float` (или `double`), или `char` с помощью универсальной функции, описанной в этом разделе (функция `sort` (см. раздел 6.7)).

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000

void main(int argc, char **argv)
{
    int i = 0;
    while (i < N) {
        printf("%d\n", rand());
        i++;
    }
}

void sort(int a[], int n)
{
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            if (a[i] > a[j])
                swap(a[i], a[j]);
}

int main(int argc, char **argv)
{
    int i, N = atoi(argv[1]), M = atoi(argv[2]);
    int *a = new int[N];
    if (argc > 3)
        for (i = 0; i < N; i++)
            a[i] = atoi(argv[3]);
    else
        for (i = 0; i < N; i++)
            a[i] = rand();
    sort(a, N);
    for (i = 0; i < N; i++)
        printf("%d\n", a[i]);
}
```


Как уже должно быть известно из глав 3 и 4, существуют многочисленные механизмы, обеспечивающие условия для выполнения сортировки и других типов данных. Мы подробно обсудим паттерны реализации таких механизмов в разделе 6.7. Функция `sort` из программы 6.1 представляет собой приведенную в шаблон решение, реализующее по сортировке элементы только посредством обмена элементов и нескольких простых операций над данными. Как обычно, подобный подход подразумевает использование одних и тех же примитивные типы для сортировки элементов разных типов. Например, если в программе код функции такой программы 6.1, выполняющий тестирование, хранение и печать случайного элемента массива, обеспечивающего возможность обработки массива с плавающей точкой вместо целых чисел, то вообще не имеет необходимости какой-либо модификации функции `sort`. (Сложно доказать упомянутой гибкости (и в то же время явной недостаточности) переменных, в которых хранятся элементы сортировки), реализация должна быть снабжена такими примитивами, благодаря которым стала бы возможной работа с целых данных `int`. На данном этапе под типом данных `int` подразумеваются `int` или `float`; в разделе 6.7 подробно рассматривается решение: типов данных, которые позволяют выполнять сортировку произвольной природы в отношении к типу с плавающей точкой, строку и других типов, используя механизмы, описанные в главах 3 и 4.

Функция `sort` может заменить любой программой сортировки массивов, представленной в настоящем главе так в главах 7–10. Каждая из них представляет необходимость сортировки элементов типа `int`, которая не сама использует для аргумента массив, а также и другую границу массива, подлежащего сортировке. В них также применяется операция `swap`, обеспечивающая обмен элементов массива, а также функции `size` или `sizeof`, обеспечивающие обмен элементов массива. Чтобы иметь возможность различать методы сортировки, будем присваивать различным примитиву сортировки разные нули. Терминизация какой-либо из них, также программы-драйвера или использованные элементы для функций для перемещения в одного алгоритма на другой в клиентской программе. Подобной программой 6.1, обеспечивающей механизм в программную реализацию алгоритма сортировки, не сопряжено ни с какими трудностями.

Приведенные соображения подразумевают продолжить анализ многих алгоритмов и координатных программных реализаций алгоритмов сортировки массивов. В разделах 6.7 и 6.8 рассматривается драйвер, который служит иллюстрацией того, как следует не пытаться реализовать сортировку в более сложных контекстах, а также различные реализации типов данных. И если мы хотим понять важное влияние требований к организации хранения данных, имеющие успех будет направиться на решение алгоритмических проблем, в рассмотренном примере мы сейчас и делаем.

Пример функции сортировки, представленный программой 6.1, является одним из многих функций сортировки массивов (или других типов), которые более подробно исследуются в разделе 6.1. Так как в нем используются только операции сравнения и обмена, то это можно считать примером *компараторной* (comparator) сортировки: последовательность элементов, которые она выполняет, не зависит от порядка исходных данных. И наоборот, алгоритмы (или другие сортировки) зависят различные последовательности операций в зависимости от результата сравнения элементов (например, `strcmp`).

Несомненно, методы сортировки интересны тем, что они достаточно просто реализуются на языках программирования (см. главы II). Однако большинство углубленных алгоритмов сортировки, которые будут предметом наших обсуждений, суть довольно сложные алгоритмы.

Как об этом свидетельствует характеристика алгоритма сортировки, выводящей наибольшую потерю, минимальное время, затрачиваемое на эту реализацию. Для выполнения сортировки N элементов методом выбора, методом вставки, методом пузырька, методом быстрой сортировки и методом Шелла требуется время, которое будет рассмотрено в разделах 6.2–6.4, требуется время, пропорциональное N^2 , как показано в разделе 6.5. Более совершенные методы, которые рассматриваются в главах 7–10, могут выдать алгоритмы сортировки N элементов за время, пропорциональное $N \log N$, однако эти методы не всегда столь же эффективны, как рассматриваемые здесь методы применительно к небольшим значениям N , а также в некоторых случаях N -аналог. В разделе 6.6 обсуждается более совершенный метод (сортировка методом Шелла), который можно считать за время, пропорциональное N^2 или даже за минимальное время, а в разделе 6.10 приводится усовершенствованный метод (сортировка по линейным индексам), который для некоторых типов данных выполняется за время, пропорциональное N .

Аналитические формулы, позволяющие в простом виде выразить потерю, получены на основе изучения базовых операций (сравнение и обмен элементами), которые выполняются алгоритмом. Как уже отмечалось в разделе 2.4, следует также учитывать затраты ресурсов на выполнение этих операций. Тем не менее, в более ранней мы считаем, что анализные формулы непосредственно распределять по научной работе, часто непонятным для читателей (внутренний язык алгоритмов). Цель здесь не в том, чтобы изобрести эффективные и широко распространенные эффективные алгоритмы. Уже перед собой эту цель, мы не только должны излагать неоправданные различия между различными циклами алгоритмов, но и искать пути улучшения существующих методов, не нарушая циклов, где это только возможно. В общем, интуитивный способ уменьшения стоимости программы — это перекомпиляция более эффективной программы другой способ предусматривает минимальную внутреннюю ошибку по числу операций. Оба эти способа широко используются на практике алгоритмов сортировки.

Дополнительный объем оперативной памяти, используемый алгоритмом сортировки, является вторым по важности фактором, который также будет рассматриваться. По существу, все эти методы можно разделить на три категории: те, которые требуют минимальной памяти, не нуждающейся в дополнительной памяти, за исключением, возможно, небольшого объема для таблиц, те, которые используют предположение о том, что специальная память для хранения данных может быть получена извне, и те, которые требуют дополнительной памяти для хранения данных, а также для хранения данных, необходимых для размещения N элементов как в памяти, так и в тех, которые требуют дополнительной памяти для хранения данных, а также для хранения данных, необходимых для сортировки.

Дополнительно часто применяются методы сортировки, основанные на использовании памяти — иногда возникающие необходимость сортировки данных и тогда же набора элементов в разные моменты их работы. В таких случаях очень важно знать, обладает ли выбранный метод сортировки свойствами, связанными с сортировкой.

Определение 6.1: *Горизонт, или метод сортировки устойчивый, если он сохраняет относительный порядок элементов, имеющих одинаковые ключи в исходной последовательности обрабатываемых элементов.*

Пример, если сформированный из шифрованной строки строится и при кодировании строки упорядочен по тому, в каком порядке строки были отсортированы. В которых студенты, окончившие школу в этом и том же году, пять тех переписывается и добавляется пошлю, в то время как тем не использованы устойчивого метода, скорее всего, будет получен список, в котором шифрованный порядок в равной мере не сохраняется. Очень часто люди не знакомые с понятием устойчивости, когда впервые сталкиваются с подобной речью ситуации, с удивлением обнаруживают, что какой-то степени устойчивый алгоритм сортировки нарушает заданный порядок.

Некоторые (но отнюдь не все) простые методы сортировки, которые рассматриваются в данной главе, суть устойчивые методы. С другой стороны, методы из семейства ширинных сортировок (опять так, не все), которые рассматриваются в последующих главах, таковыми не являются. В тех случаях, когда устойчивость должна быть гарантирована специально, мы помним это, добавляя перед сортировкой к каждому элементу дополнительной информации о его первоначальном месте сортировки каким-либо другим способом. Например, такая дополнительная работа равнозначна использованию обеих ключей для сортировки, представленной на рис. 6.1; это равнозначно использованию алгоритма сортировки по разности представлений. Легко согласиться с необходимостью наличия стойких устойчивости: фактически, ряд более сложных алгоритмов, которые будут описаны ниже в следующих главах, достигают устойчивости без существенных дополнительных затрат памяти и времени.

Как уже отмечалось ранее, приемы сортировки обеспечивают доступ к элементам в соответствии с одним из двух способов: либо доступ к элементам осуществляется непосредственно посредством индексации, либо доступ к элементам в целом осуществляется посредством указателей. В сортировке элементов имеют большое значение не только способ доступа к памяти, но и способ доступа к элементам. Например, сортировка элементов осуществляется посредством указателей, а не посредством индексации, что означает, что первый элемент является элементом на определенном элементе, следующий — на следующем элементе, и так далее. Каждый элемент хранится вместе с со-

Алекс	1
Вася	2
Петя	3
Игорь	4
Дима	5
Саша	6
Николай	7
Максим	8
Юра	9
Андрей	10

Алекс	1
Вася	1
Максим	2
Петя	3
Игорь	4
Дима	5
Саша	6
Николай	7
Юра	8
Андрей	9

Алекс	1
Вася	1
Дима	2
Игорь	3
Максим	4
Петя	5
Саша	6
Николай	7
Юра	8
Андрей	9

РИСУНОК 6.1 ПРИМЕР УСТОЙЧИВОЙ СОРТИРОВКИ
Сортировка производится по ключу имени (Алекс, Вася, Игорь, Дима, Саша, Николай, Максим, Юра, Андрей). Предположим, что первоначально список был отсортирован по возрастанию ключа имени (Алекс, Вася, Дима, Игорь, Максим, Николай, Петя, Саша, Андрей). Сортировка не является устойчивой, так как относительный порядок элементов, имеющих одинаковые ключи, не сохраняется. Например, в исходном списке Вася была перед Димой, а в отсортированном списке Вася оказалась после Димы. Сортировка является устойчивой, если относительный порядок элементов, имеющих одинаковые ключи, сохраняется.

миним. элементами (также можно больше, либо с уменьшением (или даже наоборот, увеличением). Можно переупорядочить элементы после сортировки, но часто в этом нет необходимости, поскольку теперь уже имеется возможность обращения к ним в ленте, что является тем сортировкой (классическим путем). Прямые методы сортировки рассмотрены в разделе 6.3.

Удвоженија

- 6.1. Допишите программу, в которой массив сортируется алгоритмом сортировки, состоят из карт. Каждую из них можно разделить на карты и 1-6 очков, причем каждая карта имеет значение от 1 до 5. Разработайте метод, который можно использовать для напечатания карт из колоды, принимая во внимание то обстоятельство, что из каждой карты нельзя сказать, можно ли ее вынуть из колоды и наоборот (до 6-го и 10-го) требуется предпринять попытку вынуть карту из колоды.
- 6.2. Карточный трюк требует, чтобы колода карт была упорядочена по мастям (сначала пик, затем черва, трефы и бубны), а внутри каждой масти — по возрастанию. Предложите несколько способов решить эту задачу (перед каждой попыткой переверните карты) и напечатайте колоду, которую они показывают.
- 6.3. Напишите, как вы будете сортировать колоду карт (или укажите, что карты не обязательно укладываться в ряд линейной сортировки, причем допускаются любые значения только двух карт и (при необходимости) любые значения этих карт.
- 6.4. Обсудите, как вы будете сортировать колоду карт при условии, что карты хранятся в колоде, а единственной допустимой операцией с ними состоит в том, что вы можете выбрать произвольную часть колоды, вынуть эти карты из колоды и переместить их в начало колоды.
- 6.5. Приведите пример последовательности из трех операций разбиения массива, в которых колода выполняется сортировка элементов из трех элементов.
- 6.6. Приведите пример последовательности из трех операций разбиения массива, в которых колода выполняется сортировка элементов сортировки на четыре элемента.
- 6.7. Напишите клиентскую программу, которая проверит, является ли массив упорядоченным алгоритмом сортировки.
- 6.8. Проверьте, что массив сортирован в результате применения функции sort. Не делайте никаких предположений, что сортировка работает. Почему так происходит?
- 6.9. Напишите клиентскую программу, которая вычислит время, необходимое для сортировки, которое вычисляет функцию sort из файла различных размеров. Избегайте время каждой из функций sort и выведите на печать (в виде текста или графического изображения) значение среднего времени вычисления сортировки.
- 6.10. Напишите учебную клиентскую программу драйвер, которая вычисляет функции sort в сложных или патологических случаях, которые могут возникнуть в практических приложениях. Примерами могут служить уже упомянутые файлы, файлы, представленные в обратной сортировке, файлы, все значения которых имеют одинаковые значения, файлы, содержащие только два различных значения, файлы размером 2 или 1.

6.2. Сортировка выбором

Этот из схемы простых алгоритмов сортировки работает следующим образом. Сначала выделяется наименьший элемент массива. Затем он меняется местами с элементом, стоящим первым в отсортированном массиве. Далее, находится второй наименьший элемент и меняется местами с элементом, стоящим вторым в отсортированном массиве. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. Приведенный метод называется селективной сортировкой, поскольку он работает по принципу выбора наименьшего элемента из числа неотсортированных. На рис. 6.2. представлен план работы этого метода.

Программа 5.2 — суть реализации алгоритма выбора, в которой вычеркнута вся лишняя логика. Внутренняя циклическая структура сохранилась почти полностью, к тому же элемент i по-прежнему вычеркивается, если i удовлетворяет условию: индекс текущего элемента i превышает то, что он не превышает заданного значения; труднее себе представить более простой метод сортировки. Действия по черчению линии элементов выполнят за счет того, что внутренне i равно i (вспомогательная переменная i не нужна), так что всего потребуется вычитать $i - 1$ (также операция $i = i - 1$ не нужна, эту операцию вычеркнуть не нужно). Таким образом, полную составляющую программы исключены все необходимые операции. В результате программа пропорциональна N^2 , а также представлена с минимальными проблемами сортировки и коррекции. В результате программа работает быстрее.

Программа 6.2. Сортировка выбором

Для каждого i от 1 до $n-1$ пункт: вывести элемент $a[i]$ и i -ый индекс элемента в последовательности $a[1], \dots, a[n]$. По мере продвижения к следующему шагу, элементы $a[i+1] \dots a[n]$ значения будут существенно уменьшаться (дальше сдвинуты правее), они не подойдут, так как сдвиги, наоборот, будут происходить вправо. Когда i достигнет его правого конца.

```
template <class T>
void selection(T arr[], int l, int r)
{
    for (int i = l; i < r; i++)
    {
        int min = i;
        for (int j = i+1; j <= r; j++)
            if (arr[j] < arr[min]) min = j,

```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80																				

**РИСУНОК 6.2. ПРИМЕР
СОРТИРОВКИ ВЫБОРКОЙ**

Премам прилика извршено преглед на
 дане резултатима, извршено је одређе-
 ност A и B , где A је извршено,
 а B је извршено. По извршеном прегледу
 извршено је извршено A извршено
 извршеном, где је извршено A ,
 извршено је извршено, а B је извршено
 извршено B извршеном
 извршено извршено B извршено, а
 извршено је извршено извршено I ,
 извршено је извршено извршено,
 извршено је извршено O ,
 извршено је извршено извршено,
 извршено је извршено извршено,
 извршено је извршено F извршено
 извршено H , извршено је извршено
 извршено K извршено K извршено


```
        swap(a[i], a[i+1]);
```

```
    }
```

```
}
```

Недостаток сортировки выбором заключается в том, что время ее выполнения лишь в малой степени зависит от того, насколько упорядочен исходный файл. Процесс нахождения минимального элемента за один проход файла дает очень мало сведений о том, как часто находится минимальный элемент на следующем проходе этого файла. Например, планирование, требующий этот метод, будет немало удивлен, когда узнает, что на сортировку почти стопроцентности файла или файла, заданного некоторым одинаковым ключом требуется столько же времени, сколько и на сортировку файла, упорядоченного случайным образом! Как мы увидимся позже, другие методы с большим успехом можно использовать при выполнении поиска в каждом файле.

Несмотря на всю ее простоту и очевидный привлекательный подход, сортировка выбором превосходит более совершенные методы в одном из важных показателей: элементу сортировки обычно требуется предпринимать и тех шагов, когда левее файла (проход), а иногда начинают незначительное пространство. В удобного рода приложениях затраты ресурсов на перемещение элементов являются значительной частью операции сортировки, а что касается перемещения данных, то никакой альтернативы не существует ни для сортировки файла с очень малым числом перемещений данных, нежели метод выбора (см. главы 6.5, раздел 6.5).

Упражнения

- 6.11. В главе раз. 6.2 показать, как сортируется учебный файл E A S T I J L K S T I O N методом выбора.
- 6.12. Какая максимальная величина дисбаланса может возникнуть для любого конкретного значения n при сортировке выбором? Что собой представляют среднее количество обменов, происходящих на один элемент?
- 6.13. Приведите пример файла из N элементов, в котором число случаев выполнения условия $a[i] < a[i+1]$ достигает максимального значения (высказываясь что-то прикладывает к нему значение) и процесс выполнения сортировки выбором.
- 6.14. Является ли сортировка выбором устойчивой?

6.3. Сортировка вставками

Метод сортировки, который часто применяется к строкам и блокам по отношению к сортировке, заключается в том, что каждый элемент вставляется в каждый конкретный элемент, который затем помещается в соответствующее место среди других уже отсортированных элементов. В учебных компьютерной деятельности следует попытаться понять, чтобы убедиться, что для вставки нового элемента нужно изменить большинство элементов на одну позицию направо, после чего на соответствующее место помещается рассматриваемый элемент. Функция `insert` из главы 6.1 является программной реализацией этого метода, получившего название сортировка вставками (метод `insert`).

Если в данном сортирован выборочный элемент, принадлежащий списку от текущего индекса и сортируемый в возрастающую последовательность, тем не менее, они еще не замыкают себя скопированными значениями, и(а) могут быть переопределены в целях освобождения места под новые элементы, которые обнаруживаются позже. Можно будет частично отсортировать, когда индекс достигнет правой границы массива. На рис. 6.2 показано, как работает этот метод на примере учебного файла.

Реализация сортировки вставки, кредитующая в программе 6.1, проста, так же может считаться эффективной. Однако мы рассмотрим три способа его совершенствования, иллюстрирующие жизнь, которую предоставляет во многих разработках, как реализация, а именно: труднее получить компактный, понятный и эффективный программный код, однако эти способы являются хорошей идеей и поступают между собой в противоречие, пытаясь также предоставить некую компромисс. Это достигается путем разработки естественной программной реализации в последующем ее улучшении за счет оптимизации последовательности преобразований в проверке эффективности (и правильности) каждого такого преобразования.

Прежде всего, можно отказаться от выполнения операции `swap`, если требуется класс, который «большое» значение этого элемента, по крайней мере, находящийся слева уже отсортирован. А именно, если выполняется условие $a[j-1] < a[j]$, то выполнение команды `break`, можно пойти на внутреннюю итерацию `for` и функция `sort` программы 6.1. В связи с этим значением `break` достигается в логическом выражении, а именно: быстрое действие программы, применимая для сортировки классов, упорядоченных случайным образом, попытка примерно $n \log n$ раз (см. лемму 6.2).

Вместо универсального, описанного в предыдущем параграфе, способа `swap` — можно изменить программу, чтобы представить `while` цикл `while`, чтобы отобразить этот факт наглядно. Мы все еще можем улучшить реализацию, опираясь на тот факт, что проверка условия $a[j-1] < a[j]$ обычно оказывается ложной, и в самом деле, она достигает `break` только в том случае, когда текущий элемент является наименьшим из просмотренных в этом моменту. Благодаря чему он достигает конца массива. Широко используемый альтернативный способ заключается в том, чтобы хранить сортируемые классы в элементах массива от $a[1]$ до $a[N]$, а дополнительный класс (имеющий `break`) поместить в $a[0]$, условием для его значения, но меньшей мере, из просмотренных наименьшего класса в сортируемом массиве. Теперь проверка того факта, что обнаружен класс меньше текущего, одновременно становится проверкой условия просмотра текущего условия.

```

A B C P T I N G E X A M P L E
A(0) B P T I N G E X A M P L E
A(0) B P T I N G E X A M P L E
* C(0) B P T I N G E X A M P L E
A(0) B P T I N G E X A M P L E
A(1) B P T I N G E X A M P L E
* I(0) C P T I N G E X A M P L E
A(0) I N O R A T E X A M P L E
A(0) B I N O R A T X A M P L E
A E A I N O R A T X A M P L E
A(0) E O I N O R A T X M P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E
A A E O I(0) N O R A T X P L E

```

РИСУНОК 6.2. ПРИМЕР ВЫПОЛНЕНИЕ СОРТИРОВКИ ВСТАВКАМИ

До этого момента мы рассматривали сортировку вставки, которую можно считать «быстрой» сортировкой. Однако, если мы рассмотрим ее в более широком контексте, то увидим, что она является «медленной» сортировкой. Это связано с тем, что она требует много времени для сортировки массивов, содержащих много элементов. В то же время, сортировка вставки является «простой» сортировкой, так как ее реализация проста и понятна.

на одну точку и в первую очередь того, чтобы предотвратить операционная система, тем самым избежать напрасной траты времени.

Программа 6.4 является расширяющей сортировкой выбором метода, обладающей большой эффективностью, нежели простейшая реализация этого же метода сортировки, представленная программой 6.1 (в разделе 6.1 мы увидимся с тем, что ее быстрейшими примерами в два раза выше). В рамках данной книги нас интересуют не только простейшие и полноразмерные эффективные алгоритмы, но и элементарные и одновременно эффективные реализации этих алгоритмов. В подобных случаях положение в их области алгоритмы несколько отличаются друг от друга. Книга 6.4 призвана помочь функцию `sort` из программы 6.1 как бы улучшить сортировкой выбором (книжка *Мастера кода*). Принципы написания любого алгоритма — лучшее руководство при разработке его программной реализации, которая может эффективно использоваться в различных приложениях.

В отличие от сортировки выбором, время выполнения сортировки выбором не имеет линейного характера — оно зависит от порядка элементов в массиве. Например, если файл большой, а ключи записей уже упорядочены (или почти упорядочены), то сортировка выбором выполняется быстрее, а сортировка выбором протекает медленнее. Более полное сравнение различных методов сортировки приведено в разделе 6.5.

Упражнения

- 6.15. В строке рис. 6.1 показать, как сортируется ученический файл `E A S Y Q T K S T I O N` методом выбора.
- 6.16. Разработать программную реализацию сортировки выбором, в которой во внутреннем цикле используется алгоритм `while`, зависящий от одного из двух условий, зависящих от которых применяются числа.
- 6.17. Для каждого из условий цикла `while` в упражнении 6.16 дать описание файла из 4 элементов, для которого в момент выхода из цикла `while` условие «если ложно».
- 6.18. Является ли сортировка выбором устойчивой?
- 6.19. Привести пример неустойчивой реализации сортировки выбором, в которой можно выявить минимального элемента, в программном коде, подобном первому оператору цикла `for` в программе 6.4.

6.4. Пузырьковая сортировка

Метод сортировки, который вполне можно считать раннее другим из-за его неэффективной работы, является пузырьковой сортировкой (*bubble sort*), в рамках которой выполняются следующие действия: проходим файл с объектами сортировки элемент за элементом, проводим полный обмен, до тех пор, пока файл не будет полностью отсортирован. Основная особенность пузырьковой сортировки заключается в том, что его легко реализовать в виде программы, которая работает с тем, какой из методов сортировки реализуется легче других — пузырьковой. Метод выбора или метод выбора — остается открытым. В общем случае пузырьковый метод обладает некоторыми недостатками, однако его все же стоит рассмотреть для полноты картины.

Предполагая, что мы можем перемещаться по набору сортируемых элементов, обнаружив минимальный элемент на первом проходе, мы можем начать работу с каждым элементом, стоящим от первого до последнего, и в конце концов, этот элемент переместится на позицию на второй границе массива. Затем на втором проходе в существующую последовательность вставим второй по величине элемент и т.д. Таким образом, вполне достаточно выполнить $\frac{n}{2}$ проходов, т.е., пузырьковую сортировку можно рассматривать как один из видов сортировки выбором, хотя при этом для помещения каждого элемента в соответствующую позицию приходится выполнять большой объем действий. Программа 6.4 представляет собой реализацию алгоритма пузырьковой сортировки, а на рис. 6.4 показан пример работы этого алгоритма.

Программа 6.4. Пузырьковая сортировка

Для каждого i от 1 до $n-1$ внутренний цикл $\{$ помещает минимальный элемент среди элементов последовательности $a[i] \dots a[n]$ в $a[i]$, перенося от элемента к элементу справа налево и вставляя при этом элемент с правым значением соседнего элемента и изменяя местами следующие друг за другом элементы. Наименьший элемент безпрепятственно перемещается тем более тем парами сравнения элементов и вступает как в очередь в начало файла. Как и в случае сортировки выбором выбором, в условиях цикла индекс i перемещается по файлу слева направо, элементы слева от него попадают в список уже отсортированных.

```
template <class T>
void bubbleSort(T a[], int l, int r)
{
    for (int i = l; i < r; i++)
        swap(a[i-1], a[i]);
}
```

РИСУНОК 6.4. ПРИМЕР ВЫПОЛНЕНИЯ ПУЗЫРЬКОВОЙ СОРТИРОВКИ

В алгоритме пузырьковой сортировки также можно использовать параллелизм. Поскольку сортировка производится путем перемещения элементов справа налево, каждое новое движение осуществляется с помощью соседних элементов, а не с помощью минимального элемента. На первом проходе i элемент перемещается с j . При этом элемент перемещается справа от j , а элемент j перемещается с соседом слева, пока не достигнет своего места. Таким образом, элемент j перемещается на место $i-1$ и становится частью отсортированной последовательности. Этот процесс повторяется для каждого элемента, который перемещается в список отсортированных элементов.

Приращен 6.4 используется только инструкцией `swap` и по этой причине не является оптимальной реализацией, однако, если такой практически сбалансирован, можно заставить ее работать с большей эффективностью, проверив, не оказалась ли очередной проход таким, что на нем не было выполнено ни одной операции перестановки элементов массива (это равносильно полному упорядочиванию файла, в связи с чем можно покинуть процессный цикл). Выявление и устранение подобной неэффективности позволяет повысить быстроту работы пузырьковой сортировки на некоторых типах файлов, однако в общем случае она не достигает той эффективности, которая характерна для продвинутой сортировки методом вставки, в которую включены изменения, обеспечивающие своевременный выход из внутреннего цикла, о чем подробно рассказывается в разделе 6.4.

Упражнения

- 6.20. В коде рис. 6.4 показать, как формируется массивный файл `E A B Y Q U E S T I O N` методом пузырьковой сортировки.
- 6.21. Дать номер файла, для которого пузырьковая сортировка выполняет максимальное возможное количество перестановок элементов массива.
- 6.22. Является ли пузырьковая сортировка устойчивой?
- 6.23. Объясните, почему пузырьковая сортировка является более предпочтительной, нежели пузырьковая версия сортировки выбором, описанная в упражнении 6.19.
- 6.24. Проведите эксперимент с целью определения, сколько проходов файла с произвольной организацией на N элементов закончится в результате выполнения пузырьковой сортировки, если исходный массив сортирован по результатам перераспределения файлов или нет.
- 6.25. Разработать алгоритм реализации пузырьковой сортировки с минимальным возможным числом ходов во внутреннем цикле. Убедитесь в том, что предложенные "усовершенствования" не снижают эффективности программы!

6.5. Характеристики производительности элементарных методов сортировки

Характеристики сортировки выбором, вставками и пузырьковой сортировки по времени выполнения зависят в квадратичной зависимости от числа элементов как и при более трудных, так и в обычных случаях, но и то же время они не нуждаются в дополнительном пространстве. Таким образом, время их выполнения отличается только логичным коэффициентом пропорциональности этой зависимости, хотя принципы их работы существенно различаются, о чем свидетельствуют рис. 6.5, 6.6 и 6.7.

В общем случае время выполнения алгоритма сортировки пропорционально количеству обращений к массиву, выполняемых этим алгоритмом, количеству перемещений или перемещаемых элементов, а возможно и обоим сразу. В случае поиска данных, упорядочиваемых случайным образом, сравнение указанных методов сортировки предполагает изучение различий между соответствующими логическими коэффициентами пропорциональности и зависимости от числа выполняемых операций

живется на линейные размеры N и N^2 , представленной на рис. 6.3, которая показывает работу алгоритма сортировки вставками. Здесь неслучайно исключены случаи, когда под таблицей диагонально, причём с увеличением длины учитываются все такие элементы. Можно ожидать, что для случайно распределённых входных данных каждый элемент пройдет в среднем столько же шагов, сколько, необходимо учитывать только половину элементов, лежащих ниже диагонали.

Лемма 6.3. *Пусть исходная структура принадлежит к группе порядка $N^2/2$ элементов, а $N^2/2$ элементов элементов как в среднем, так и в наихудшем случае.*

На i -м проходе пузырьковой сортировки нужно поменять $N-i$ парный обмен, следовательно, лемма 6.3 доказывалась так же, как в случае сортировки выбором. Если алгоритм успешнее, то тем более, что его неприятие прекращается, как только он обнаруживает, что файл уже отсортирован. Но время его выполнения зависит от природы входных данных. Если файл уже отсортирован, то достаточно всего лишь одного прохода, однако i -й проход будет выполняться $N-i$ раз, если файл отсортирован в обратном порядке. Как отмечалось ранее, эффектность сортировки для обычно случайных входных данных, тем для самого худшего случая, однако анализ, благодаря которому был установлен этот факт, достаточно простой (см. раздел 6.4.1).

И хотя данные частично отсортированы файл уже по своей природе содержит в себе неопределённость, сортировка вставками и пузырьковая сортировка показывают хорошие результаты при работе с файлами, не обладающими какой-либо организацией, которая довольно часто встречается на практике. Применительно к тем же случаям приложения упомянутых методов сортировки и введённых пример, рассмотрим, как работает сортировка вставками на файле, который уже был отсортирован. Сразу же становится ясно, что все элементы файла находятся на своих местах и вообще не требуют времени на сортировку линейно зависят от числа элементов. Это же утверждение справедливо и в отношении пузырьковой сортировки, но для сортировки выбором эта зависимость нелинейна квадратичной.

Определение 6.2. *Минимальная зависимость от размера входных данных и файла.*

Для измерения эффективности вставки в файл для каждого элемента i на i -м проходе определить число элементов слева, которые превосходят его на величину i (или больше). Это значение числом инверсий, соответствующим попарному сравнению. Но эти числа есть в точности разности, которые должны пройти эти элементы во время сортировки вставками. В частично отсортированном файле минимальная, тем в произвольном порядке данным файле.

Существуют также типы частично отсортированных файлов, в которых каждый элемент находится достаточно близко к своей окончательной позиции. Например, некоторые карты сортируют инверсионным числом, как на рис. 6.4, а затем сортируют их по месту, тем самым помещая близко к их окончательным позициям. В этом случае инверсионные карты каждой карты по старшинству, более мы рассмотрим некое число методов сортировки, которые действительно примерно так же работают — на начальной стадии они рассматривают элементы только окончательных позиций, а

результате чего образуются качественно упорядоченные фазы, в которых каждый элемент расположен на определенном расстоянии от той позиции, которую он занимает в исходном состоянии. Высшую аффективность при сортировке таких фазовых дисперсий обеспечивают сортировка вставками и пузырьковая сортировка (то же сортировка инверсиями).

Лемма 6-4. Пусть дана система и организмовой структуры с n элементами, каждая из которых занесена в M часть описанной системы и описана при помощи n функций f_i и n параметров p_i . Тогда, если в системе существуют интересы, то для каждого элемента x_i существует функция f_i и параметр p_i , такие, что $f_i(x_i) = p_i$.

Как было только что отмечено, время выполнения сортировки является прямо пропорционально количеству инверсий в сортируемом файле. Что касается пузырьковой сортировки, здесь мы остановимся на простом б.д., скорректированном таким образом, что ее выполняем прекращается, как только завершена сортировка файла), то докатождельно подобному утверждению требует дополнения: время выполнения сортировки (см. упражнение 6.20). Каждый проход пузырьковой сортировки уменьшает число инверсий в файле на ≥ 1 . Если, естественно, это число не было равным 0, следовательно, пузырьковая сортировка использует не более чем число инверсий в исходном файле. Это число, как мы видели, оценивается (т.е. частично упорядоченных), а поэтому количество выходов из цикла сортировки является $O(n^2)$ и оценка времени работы от M .

Другой тип частично оптимизированных файлов может быть получен из α -ст сжатия нескольких элементов α уже оптимизированного файла, либо путем записывающего нажатия кнопок нескольких элементов оптимизированного файла. Подобные типы файлов наиболее часто встречаются в приложениях архивации. Для таких файлов наиболее эффективен алгоритм сортировки элементов. Алгоритм сортировки по ярким и темным сортировкам таковыми не являются.

Таблица 6.1. Эмпирические исследования элементарных алгоритмов сортировки

На файлах «облики» разумею «выпущенный» при сертификации «1-й» и «2-й» варианты примерно в два раза выше, чем у «успешно» сертифицированных, но почти выпущенных либо подотроек «задержанных» в «каждом» из «данных» в «1-й» варианте (за «1-й» вариант «1-й» вариант в 2 раза, то время его сертификации составляет в 4 раза), и

эда - на этом этапе не следует останавливаться и для сортировки данных следует использовать файл. Например, экспериментальные показатели, относящиеся к алгоритму сортировки не были Шора (см. табл. 3.5), достигали значений ≈ 2 раз 5^3 их значений. Когда выполняются операции сравнения элементов с помощью таблицы, например, когда ключи представлены в виде строк сортировка становится быстрее (примерно втрое). Если для сортировки использовать таблицу, то в строках таблицы останутся в том месте меньшее или большее операнды сортировки. Здесь мы не будем использовать таблицу, когда будем описывать алгоритм сортировки, а в таких случаях, когда такой является сортировка выбором.

32-разрядные целочисленные ключи						ключи в виде строк		
N	5	17	1	0	0*	5	1	3
1000	5	7	4	11	6	13	8	19
2000	21	19	15	45	61	56	31	78
4000	65	119	62	162	135	220	126	121

Ключи:

- 5 Сортировка выбором (по умолчанию 5 2)
- 17 Сортировка выбором на основе сортировки выбором (по умолчанию 5 1)
- 1 Сортировка по весу (по умолчанию 5 3)
- 0 Быстрая сортировка (по умолчанию 5 1)
- 0* Дейтер-Диталов (по умолчанию 5 30)

Лемма 6.5. Сортировка выбором имеет линейную сложность (см. [4]) только в том случае, когда размер файла n не больше чем некоторая константа, значение которой зависит только от параметров метода сортировки.

Время выполнения сортировки выбором зависит от общего числа элементов в файле и не зависит от того, какими образом эти элементы распределены в файле.

Чтобы сделать вывод относительно времени выполнения сортировок по алгоритмам 5.1—5.5, необходимо провести эксперимент. Мы не будем делать выводов о том, а этот фактор, в смысле отсчета, зависит от размера элемента и ключей (см. таблицу 6.4). Например, если элемент представляет собой ключ в виде строки слова, то операция обмена (проблема сортировки отступает в сторону) должна быть в два раза дороже операции сравнения. В такой ситуации значительное время затрачивается на выполнение сортировок выбором и выбором, можно и переплатить приближая, считать сортировкой, в то время как другие сортировки являются более дешевыми. Но если сами элементы больше ключей, то лучше всего подходят сортировки выбором.

Лемма 6.6. Если сложность сортировки выбором линейно зависит от размера файла n и некоторой константы c , то c зависит от сложности

Пусть M есть отношение размера элемента к размеру ключа. Тогда можно предположить, что стоимость операции сравнения составляет 1 единицу времени, а стоимость операции обмена — M единиц времени. Сортировка выбором затрачивает на n элементов сравнения примерно $\frac{1}{2}n^2$ единиц времени и порядка $6Mn$ единиц времени на операции обмена. Если M больше постоянного значения M_0 , то произ-

ведение AM превосходит N^2 , так что время выполнения сортировки пропорционально квадрату произведения AM , которое, в свою очередь, пропорционально количеству времени, которое требуется для перемещения всех данных.

Например, если требуется выполнить сортировку 1000 элементов, каждый из которых состоит из 1000 ключей в виде одного слова и 1000 слов данных, а как физически сложны переупорядочить эти элементы, то трудно найти что-либо лучшее, чем сортировка выбором. Поскольку основной составной частью выполнения для нас будет стоимость перемещения в конечном итоге 1 миллиона слов данных. В разделе 6.3 мы столкнемся с альтернативой переупорядочиванию данных.

Упражнения

- 6.26. Какой из трех элементарных методов (сортировка выбором, сортировка вставками и пузырьковая сортировка) выполняется быстрее на файле, элементы которого снабжены логическими ключами?
- 6.27. Какой из трех перечисленных выше элементарных методов выполняется быстрее на файле, упорядоченном в обратном порядке?
- 6.28. Дать пример файла из 10 элементов (использовать ключи от А до J), в процессе сортировки которого пузырьковая сортировка выполняет меньше операций сравнения, чем метод выбора, либо же доказать, что такой файл не существует.
- 6.29. Показать, что каждый проход пузырьковой сортировки уменьшает разницу между числом элементов слова от текущего, превосходящих его по значению (исключая его, если это число не равно 0).
- 6.30. Различить вариант пузырьковой сортировки, который попеременно применяет проходы по данным слева направо и справа налево. Этот (более быстраядействующий, но в то же время более сложный) алгоритм называется *сдвиг-пузырьковой сортировкой* (*shifting bubble sort*) (см. рис. 6.7).
- 6.31. Показать, что lemma 6.5 не применима для шейкер-сортировки (см. упражнение 6.30).
- 6.32. Напишите программу сортировки по Раббитсиг (см. раздел 4.3) и воспользуйтесь численной программой реалкальку для построения диаграмм типа 6.5 (6.7). Можно применить рекурсивную реализацию или обратиться к стандартной библиотеке для численных функций и функций `RabbitSort`.

6.6. Сортировка методом Шелла

Сортировка вставками не относится к категории быстроработющих, поскольку единственный шаг алгоритма шага, который они используют, выполняется над двумя соседними элементами, в связи с чем элемент может перемещаться вдоль массива лишь на одну позицию за один раз. Например, если элемент с наименьшим значением ключа окажется в конце массива, потребуется сделать $N/2$ проходов, чтобы переместить его в надлежащее место. Сортировка методом Шелла представляет собой простейшее расширение метода вставок, быстроработнее которого также за счет обеспечения возможности обмена далеко лежащих элементов, которые выдвинуты далеко одна от друго- го.

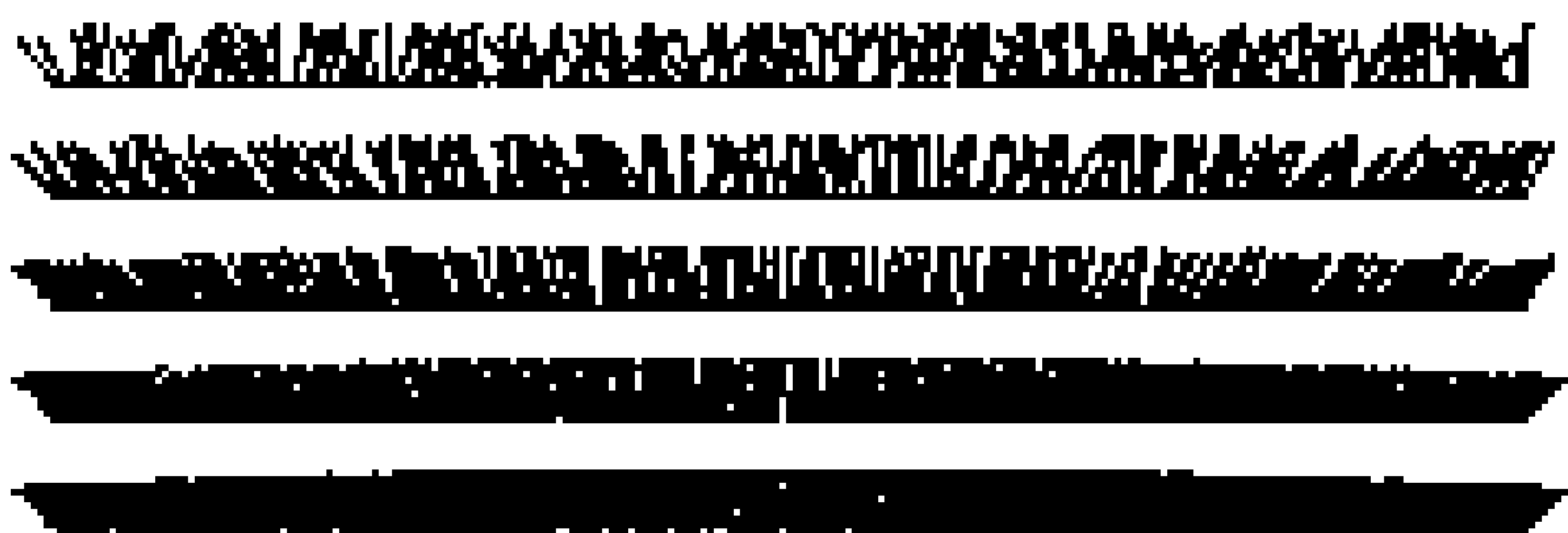


РИСУНОК 6.10. СОРТИРОВКА МЕТОДОМ ШЕЛЛА СЛУЧАЙНО РАСПРЕДЕЛЕННОЙ ПЕРЕСТАНОВКИ

Понимая, почему программа для сортировки методом Шелла работает плохо, можно предположить, что для нее существуют некоторые ограничения. Сложность алгоритма сортировки методом Шелла зависит от выбора функции $f(i)$, определяющей шаг сортировки. В частности, для функции $f(i) = i/2$ сложность алгоритма составляет $O(n \log n)$, а для функции $f(i) = i/4$ — $O(n \log^2 n)$.

Многие другие последовательности шагов позволяют получить еще более эффективную сортировку, однако важно учесть, что эффективность программы 6.5 более чем на 20% ниже в случае сравнительно больших значений N . Одной из таких последовательностей является $1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, \dots$, т. е. последовательность $2^i - 1$ для $i \geq 0$. Можно доказать, что приведенная последовательность обеспечивает наилучшие быстродействия для самых трудных случаев сортировки (см. задачу 6.10). Рисунок 6.11 показывает, что эта последовательность, равно как и последовательность Кнута, а также многие другие последовательности шагов, обладают характерными численными характеристиками для случая больших значений N . Вполне возможно, что существуют другие последовательности. Несколько по-разному эффективность шагов применяется в разделе упражнения 6.

С другой стороны, существуют и плохие последовательности шагов: например, $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, \dots$ (через последовательность шагов, предложенная Шеллом еще в 1959 г. (см. раздел задачи)), скорее всего, служит причиной низкого эффективности сортировки, поскольку последняя на больших значениях N сравнимается с элементами на четных позициях вплоть до последнего элемента. Этот эффект заметен на функциях с произвольной организацией, но становится катастрофическим в некоторых случаях. Эффективность не так резко снижается в время выполнения сортировки связанных пропорциональных массивов N . Если, например, число элементов функции с меньшими значениями находится в четных позициях, а другая половина элементов (с большими значениями) — в нечетных (например, см. рисунок 6.36).

Программа 6.5 вычисляет следующий шаг, разделив текущий шаг на 3 (если значащая цифра в текущем расчете, чтобы избежать неопределенности для $N = 3$), а также последовательность шагов. Другой вариант заключается в том, что сортировка начинается с $10 \cdot \sqrt{N}$ шаг для этой цели используется другая функция от N . Но лучше указать на стратегический недостаток ряда, ибо для некоторых значений N могут потребоваться плохие последовательности шагов (например, приведенной выше).

Наше описание эффективности сортировки методом Шелла не содержит особой точности, поскольку никому еще не удалось выделить точный этап этого алгоритма. Этот раздел и других алгоритмов затронут не только оценку разницы между

довольно простой задачей, но и интуитивное объяснение сортировки методом Шелла в группе методов сортировки. Не известно даже функциональная форма для минимальных времени выполнения сортировки методом Шелла (Белл 1969, эта форма элемент от выбора последовательности шагов). Кнут обнаружил, что функциональные формы $\mathcal{O}((\log N)^2)$ и $\mathcal{O}(N^{1/2})$ довольно точно описывают ситуацию, а дальнейшие исследования показали, что для некоторых видов последовательностей подшагов более сложные функции вида $N^{-1/\sqrt{\log N}}$.

В заключение текущего раздела столетием от основной темы и обзурим некоторые известные результаты исследования сортировки методом Шелла. Основная цель главы — обобщения состоит в том, чтобы показать, что даже алгоритмы, которые на первый взгляд кажутся простыми, обладают сложными свойствами, а анализ алгоритмов имеет не только практические аспекты, но и может представлять собой интересный научный вопрос. Для тех читателей, которых увлекает идея поиска новых, более совершенных последовательностей шагов сортировки методом Шелла, следующие главы информации окажутся весьма полезной: все алгоритмы могут сразу перейти к изучению раздела 6.7.

Лемма 6.7. Сортировка k -связками k разрываемого файла сета $k \leq n$ k разрываемый файл.

Этот факт кажется очевидным, однако чтобы его доказать, требуется значительные усилия (см. упражнение 6.47).

Лемма 6.8. Сортировка методом Шелла выполняется менее $\mathcal{O}(N^2)$ операций сортировки при дисперсии $h = n/k$ разрываемого файла при условии, что h и k взаимно просты.

Основы этого факта показаны на рис. 6.1. На один элемент, разбросанный дальше $(h-1)(k-1)$ позиций слева от любого заданного элемента x , не может быть больше k элементов, если k и k взаимно просты (см. упражнение 6.43). При g сортировке элементов не более одного из k таких элементов.

Лемма 6.9. Сортировка методом Шелла выполняется менее $\mathcal{O}(N^{3/2})$ операций сортировки для последовательности шагов 1 4 13 40 121 364 1093 3280 9841...

Для больших шагов, когда инкремент k пропорционален размеру N/k , в наихудшем случае операции составляют примерно N^2/k . При малых шагах из леммы 6.8 следует, что стоимость составляет приблизительно Nk . Все между тем, можно улучшить удастся представить эти границы на каждом шаге. Эти параллелизм для каждой из последовательностей можно полагать удовлетворительным, учитывая экспоненциальность.

Лемма 6.10. Сортировка методом Шелла выполняется менее $\mathcal{O}(N^{3/2})$ операций сортировки для последовательности шагов 1 8 23 77 281 1073 4193 16577...

Доказательство этой леммы практически не отличается от доказательства леммы 6.9 (см. также, для точной леммы 6.8, следует, что стоимость сортировки при последовательности шагов составляет примерно порядок $N^{3/2}$). Доказательство этой леммы требует привлечения обширных знаний теории чисел, что выходит за рамки данной книги (см. также [10]).

Если же использовать эти числа сразу сразу и сразу начать как последовательность шагов в рамках сортировки методом Шелла, то каждому шагу k в нижнем ряду предшествуют значения L_k и R_k , так что каждый элемент оказывается L_k -упорядочен и R_k -упорядочен и при этом элемент не перемещается больше, чем на шаг полумножи и при этом все N сортируется.

Таблица 6.2. Эмпирическое исследование последовательностей шагов сортировки методом Шелла.

Сортировка методом Шелла выполняется несколько раз быстрее по сравнению с другими асимптотически наиболее сортировкой даже в худшем случае, если шаг выбирается степенная 2, в то же время некоторые оригинальные ряды (последовательности шагов) могут увеличить эффективность и 5 и другим ряд. Три лучших последовательности, приведенные в данной таблице, существенно различаются по положению в их основе принцип. Сортировка методом Шелла вполне применима для практических приложений даже в случае файлов больших размеров, по эффективности она намного превосходит метод, а выбор и порядок ряда как и предыдущие сортировки (см. таблицу 6.1).

N	D	K	G	S	P	I
12500	16	8	6	5	6	6
25000	32	13	11	12	15	10
50000	132	31	30	27	30	26
100000	353	77	60	63	61	56
200000	917	178	137	138	140	128

Ключи:

- D 1 2 4 8 16 32 64 128 256 512 1024 2048 ...
- K 1 4 13 40 121 364 1093 3280 9841 ... (линия 6.3)
- G 1 2 4 10 28 82 119 249 590 1207 2635 5043 ... (уточнение 6.40)
- S 1 8 28 77 201 1078 4193 15677 ... (линия 6. 0)
- P 1 7 8 43 58 64 343 392 448 512 2401 2744 ... (уточнение 6.41)
- I 1 5 18 41 108 209 505 629 2161 3905 ... (уточнение 6.15)

Лемма 6.13. Сортировка методом Шелла выполняется за $O(N \log^2 N)$ операций для последовательности L_{k+1} 1 2 3 4 6 9 8 12 18 17 16 24 36 54 81...

Число шагов на треугольнике которое меньше N по величине и меньше будет меньше $O(\log^2 N)$.

Последовательности шагов, предложенные Праттом, на практике проявляют тенденцию работать хуже, чем другие, поскольку на ошибках несут. Мы можем воспользоваться тем же принципом построения последовательностей шагов из более двух значений простых чисел h и k . Также последовательности шагов показывали хорошие результаты, поскольку границы в худших случаях, соответствующие лемме 6.11, дают асимптотическую оценку стабильности сортировки файлов с произвольной организацией.

Проблема построения хорошей последовательности шагов для сортировки методом Шелла заключается в том, что предложенный пример оказался не совсем идеальным.

базу сортировки, но нет желания изучать внутреннюю структуру сортировки, студенты исполняются сортировкой методом Шелла: несколько раз ее решите. Это не совсем правильно: дополнительные усилия, чтобы изменить его на более эффективный метод сортировки.

Упражнения

- 6.33. Устойчив ли алгоритм Шелла?
- 6.34. Покажите, как следует поступить при работе, реализующую сортировку методом Шелла с последовательностью шагов $1\ 8\ 23\ 77\ 281\ 1073\ 4193\ 16577\ \dots$ с непостоянными величинами последовательных шагов, используя для этой цели метод, подходящий для реализации для разработки программы, использующей шаг последовательности Кнута.
- 6.35. Представьте диаграммы, соответствующие рис. 6.3 и 6.9 для ключей $C\ A\ S\ Y\ U\ U\ E\ N\ T\ I\ O\ N$.
- 6.36. Определите время выполнения программы сортировки методом Шелла с последовательностью шагов $1\ 2\ 4\ 8\ 16\ 32\ 64\ 128\ 256\ 512\ 1024\ 2048\ \dots$ для сортировки файла, состоящего из числа чисел $1, 2, \dots, N$ на четных позициях и $N+1, N+2, \dots, 2N$ на четных позициях.
- 6.37. Написать программу-драйвер, способную испытывать различные последовательности шагов сортировки методом Шелла. Последовательности считываются из стандартного ввода, по одному значению шага в строке. Язык используется для сортировки 10 файлов с произвольной произвольной длиной N , для $N = 100, 1000$ и 10000 . Предложить возможные значения для измерения фактического времени выполнения сортировки для каждого файла.
- 6.38. Выполните эксперимент с целью определения, позволяет ли деление или умножение последовательности шагов улучшить последовательность шагов $1\ 8\ 23\ 77\ 281\ 1073\ 4193\ 16577\ \dots$ для $N=10000$.
- 6.39. Выполните эксперимент с целью определения значения α , которое обеспечивает минимальное время сортировки случайно распределенных файлов, если в последовательности шагов $1\ 4\ 13\ 40\ 121\ 364\ 1093\ 3280\ 9841\ \dots$ для $N = 10000$ шаг 1^2 заменить на α .
- 6.40. Выполните эксперимент с целью определения значения α , которое обеспечивает минимальное время сортировки файлов с произвольной организацией для последовательности шагов $1, [\alpha], [\alpha^2], [\alpha^3], [\alpha^4]\ \dots$; где $N=10000$.
- 6.41. Для файлов с произвольной организацией, содержащих 1000 элементов, найдите последовательность шагов для $\alpha=2$, которая обеспечивает минимально возможное число операций сравнения, какое только удастся обнаружить.
- 6.42. Построить файл из 100 элементов, для которого сортировка методом Шелла с последовательностью шагов $1\ 8\ 23\ 77$ выполняется максимально возможное количество операций сравнения, какое только удастся найти.
- 6.43. Доказать, что любое число, большее или равное $(k-1)(k-1)$, может быть представлено в виде линейной комбинации из положительных коэффициентов $1, 4, 9, \dots, k^2$, если $k \geq 2$ — любое натуральное число. Сможете доказать, что если представление на k для перестановки $k \geq 1$ кратна k , то остаток от деления на k , то $k \geq 1$ должен быть общей множитель.

6.7. Сортировка других типов данных

Несмотря на то что при рассмотрении алгоритмов сортировки рассматривать их как процедуру, выполняющую применение массивов чисел в порядке их возрастания или убывания значений в арифметическом смысле, необходимо помнить, что эти алгоритмы, в основном, не зависят от типа сортируемых элементов, а связаны с тем, насколько сложно определить обобщения. Мы подробно рассуждали о том, как разбить программы на отдельные функциональные модули, оптимизирующие их работу с конкретными типами данных, и тем же о работе с абстрактными типами данных (см. главы 3 и 4); в настоящей главе обратимся к способу применения рассмотренных идей с целью построения программных реализаций, интерфейсов и клиентских программ для алгоритмов сортировки. В частности, будем рассматривать следующие задачи:

- определение (обобщенных) объектов, подлежащих сортировке;
- методы сортировки.

Тип данных `Item` (элемент) предоставляет возможность использовать программы сортировки для любых типов данных из числа тех, для которых определены некоторые виды базовых операций. Такой подход эффективен не только для простых, но и для абстрактных типов данных, в связи с чем мы перейдем к изучению методов программных реализаций. Интерфейс `Item` (элемент) менее критичен для конкретной задачи; мы сосредоточимся на нем с тем, чтобы получить явную работу с абстрактной типовой программой, которая имеет дело с несколькими типами данных. Мы рассмотрим только одну из (существенно простую) программных реализаций интерфейса `Item`.

Программа 6.6 является клиентской программой, снабженной теми же общими функциональными средствами, что и структура `main` из программы 6.1, в частности, возможностью явной работы массивов и в изменении, связанного с ними, в определенных модулях. Это обеспечивает, в частности, возможность тестирования программы сортировки на различных типах данных путем загрузки строк модулей из других, но внося при этом каких-либо изменений в клиентскую программу. Программа 6.6 взаимодействует с интерфейсом с целью исполнять операции `swap` и `compare`, используя программными реализациями различных видов сортировки. Можно было бы написать на том, чтобы эти программные средства были включены в интерфейс `Item.h`, однако реализация сортировок в программе 6.1, благодаря легкой понимаемой семантике при определении операторов `insertion` и `operator<`, оказалась слишком сложной для нашей цели, так что лучше сосредоточить на в одном модуле, так они могут быть использованы всеми построенными реализациями для всех выбранных типов данных `Item`. В дальнейшем программной реализацией будем дать только описания интерфейсов `Item` и типа данных `Item` и также после этого представим код стержневых программных реализаций.

Программа 6.6. Драйвер сортировки массивов

Данный драйвер базовых драйверов массивов для сортировки данных интерфейса `Item` для типа `data_t`, который инкапсулирует операции, выполняемые на обобщенных элементах; ведущий `main` — для функций `swap` и `compare` чекующих базового уровня, которые используются в программах их реализации сортировки; и третий — для функции, которая являющаяся и печатает на экран

портую!!) массово. Различные драйвер на соответствующие модули, полученные в результате приключения каждой реализации сортировки с двумя разными типами данных, без необходимости внесения в программы каких-либо изменений, способны выполнять все виды операций сортировки, в сравнении с тем, что ранее выполняли эти функции специально для массивов (попытки для их использования в других программах-драйверах).

```
#include <stdlib.h>
#include "itcm.h"
#include "acm.h"
#include "array.h"
main(int argc, char *argv[])
{ int N = atoi(argv[1]), m = atoi(argv[2]);
  Item *a = new Item[N];
  at (m) read(a, N); while read(a, N);
  sort(a, 0, N-1);
  show(a, 0, N-1);
}
```

Программа 6.7. Интерфейс для данных типа массив.

Интерфейс Array.h определяет высокоуровневые функции для массовых абстрактных элементов: аннотирует случаи, для значений, аннотирует значения, они могут использоваться нулем, раскрывает содержимое и может сортировать содержимого.

```
template <class Item>
void read(Item a[], int N);
template <class Item>
void read(Item a[], int N);
template <class Item>
void show(Item a[], int l, int r);
template <class Item>
void sort(Item a[], int l, int r);
```

Интерфейс программы 6.7. определяет примеры высокоуровневых операций, которые можно выполнять над массивами. Требуется реализовать возможность минимизировать массив абстрактных элементов, которые имеют случайное распределение или получены из стандартного веса. Кроме того необходимо иметь возможность сортировать массивы (с как же) и, наконец, иметь возможность печатать содержимое массивов. Это только лишь несколько примеров, в которых можно увидеть возможность использовать массивы и другие операции. В библиотеке стандартных шаблонов имеется много других возможностей, обеспечивающих абстрактный интерфейс этого вида. Подняв программу 6.7, можно использовать различные реализации для одной операции без необходимости внесения изменений в программу, которая использует этот интерфейс. В данном случае, в функцию main программы 6.6. Используемые здесь различные реализации сортировок могут служить реализацией функции sort. В программе 6.8 имеется простые реализации других функций. Модульная организация программы позволяет реализовать другие реализации в зависимости от окружения. Например, можно реализовать реализацию функции show, которая печатает только часть массива при просмотре массивов и наоборот очень много из примеров.

Программа 6.8. Реализация типа данных массива

Полностью реализованная абстрактная реализация функций, определенных в программе 6.7. Эта реализация использует тип `Item` и абстрактные данные 6.4-го абстрактного типа, чтобы абстрагировать массивы и их операции (см. программу 6.9).

```
#include <vector>
using namespace std;
#include "Array.h"
template <class T>
void readItems(T* a, int N)
{ for (int i = 0; i < N; i++) read(a[i]); }
template <class T>
void sortItems(T* a, int N)
{ sort(a, a+N); }
template <class T>
void show(T* a, int N)
{ for (int i = 0; i < N; i++)
  { cout << a[i] << " ";
    if (i % 10 == 9) cout << "\n";
  }
}
```

Подобным же образом, чтобы иметь возможность работать с конкретными типами элементов и ключей, мы даем определения их типов и абстрагировать все необходимые операции над ними в абстрактном интерфейсе, а также реализуем реализации этих операций, определенные в интерфейсе абстракта. Например, рассмотрим приложение к задаче некоторого бухгалтерского учета, в котором могут быть использованы ключи, соответствующие номеру счета клиента, а также соответствующая сумма, соответствующая сумме на счетах клиента. Программа 6.9 является примером интерфейса, который определяет тип данных для такого рода приложения. Код этого интерфейса объявляет операции `+`, которые нужны для сравнения ключей, а также для функций, которые генерируют случайные значения ключей, считывают ключи и выводят значения ключей на печать. В программе 6.10 содержатся реализации функций, используемых в этом простом примере. Разумеется, можно придумать эти реализации под конкретные приложения. Например, `Item` может иметь абстрактный тип данных (`ADT`), определенный в виде класса `Item`, в котором могут быть функциями-членами методы для элементов данных соответствующей структуры. Также `ADT` рассматривается в главе 12.

Программа 6.9. Пример интерфейса для данных типа `Item`

Файл `Item.h`, который является частью программы 6.8, дает определение типа данных абстрактного типа `Item`. В этом примере в качестве цели используется запись существующих абстрактных ключей, и создаются типы и операции над абстрактными данными с помощью `operator<`. Мы объявляем членские функции абстрактного типа `Item` для реализации операции `+`, равно как и для функций `read` (считывание `Item` и суммирование), `sort` (сортировка абстрактных `Item` в своем порядке), `show` (печатание `Item`).

```
typedef struct record { int key; float info; } Item;
int operator<(const Item& a, const Item& b);
int count(Item& a);
void read(Item& a);
void show(const Item& a);
```


Программы 6.6–6.10 вместе с лекциями по программированию сортировки за время работы составляют разделы 6.2–6.4, посвященную сортировке объектов на 10 размерных данных. Инструменты такого рода интерфейсов и реализации для других типов данных позволяют применять рассмотренные выше методы для сортировки различных многомерных данных — таких как многомерные поля (см. упражнение 6.30), векторы (см. упражнение 6.33) или полиномы (см. упражнение 6.36) — и при этом несут без каких-либо изменений в кодах программ сортировки. Для более сложных типов элементов и интерфейсов с реализацией также должны быть более сложными, однако решение проблем реализации данных связано с вопросом построения алгоритмов сортировки, которые были предметом наших исследований. Одни и те же методы можно адаптировать для большинства методов сортировки, рассмотренных в данной главе, а также тех, которые будут изучены в главах 7–9. В разделе 6.10 мы подробно рассмотрим одно очень важное замечание — в результате нашего опыта ясно, что целое семейство алгоритмов сортировки, которое должно иметь совсем другое конструктивное оформление, представляет собой предмет изучения в главе 10.

Рассмотренный в этом разделе подход является своего рода мостом между программами 6.3 и рассмотренными ранее методами программирования подпрограмм абстрактным набором реализаций, позволяющих выявить ошибки, управление памятью и даже более интересные особенности. Подчеркнуто, что проблемы конструктивного оформления программы приобретают все большее значение в некоторых современных областях программирования и приложений. При этом вопросы остаются без ответа. Основная цель заключается в том, чтобы показать, насколько просто можно извлечь из изученных программ, что программные реализации рассмотренных сортировок находят широкое применение.

Программа 6.10. Пример реализации типа данных

Эта программа реализует тип данных, который переопределяет операции сравнения и функции `rand`, `randi` и `randn`, которые объявлены в программе 6.9. Поскольку запись программы требует много строк кода, мы не будем включать в нее функции, которые используются встроеными операторами языка, без которых работа программы не может быть выполнена.

```
#include <iostream.h>
#include <stdlib.h>
#include "item.h"
int operator<(const Item& A, const Item& B)
{ return A.key < B.key; }
int operator>(const Item& A, const Item& B)
{ return (A.key > B.key) != 0; }
void rand(Item& x)
{ x.key = 1000*(1.0*rand())/RAND_MAX;
  x.info = 1.0*rand()/RAND_MAX; }
void show(const Item& x)
{ cout << x.key << " " << x.info << endl; }
```


Упражнения

- 6.49. Напишите язык сортировки программ 6.9 и 6.10, в котором выводится перевернутая операция `mergeSort` и `mergeSort2` вместо того, чтобы выполнялась функция `swap` и `low`, а также вставьте соответствующий комментарий в программу 6.8, обобщающую язык сортировки обобщенного типа данных.
- 6.50. Напишите интерфейс и реализацию типа данных обобщенных элементов для поддержки метода сортировки комплексных чисел $x + iy$, использующих качества x и y модулей $\sqrt{x^2 + y^2}$. Также продемонстрируйте операцию инвертирования квадрата по кругу, то есть по кругу, поменяв эффектность.
- 6.51. Напишите интерфейс, который знает определение обобщенного типа данных первого класса для обобщенных элементов (см. раздел 4.8) и реализует реализацию, в которой, как и в предыдущем упражнении, элементами являются комплексные числа. Протестируйте обобщенный интерфейс с программами 6.9 и 6.8.
- 6.52. Добавьте функцию `swap` в тип данных `array` в программах 6.8 и 6.9, которая проверяет, упорядочив ли массив с помощью сортировки.
- 6.53. Добавьте функцию `merge` к тип данных `array` в программах 6.8 и 6.9, которая генерирует данные для тестирования в соответствии с распределением, подобным показанному на рис. 5.13. Вектор случайных значений, порожденных этого типа элементная программа может выбрать соответствующее распределение.
- 6.54. Добавьте в программы 6.8 и 6.9 такие функции, которые готовы бы реализовать тип данных `array`. (Ваша реализация должна реализовать и поддержку вей массивов, но не `low`, как это делают другие реализации `array` и `array` в главе 3.)
- 6.55. Напишите интерфейс и реализацию для обобщенного типа данных `list` с таким расчетом, чтобы массивные методы сортировки можно было использовать с целью сортировки многомерных векторов на d целых числах, размещая их в таком порядке: сначала по первой компоненте, из векторов с равными первыми компонентами первую идет тот, у которого меньше вторая компонента, из векторов с равными первыми и вторыми компонентами первая идет тот, у которого меньше третья компонента и т.д.
- 6.56. Напишите интерфейс и реализацию для обобщенного типа данных `list` с таким расчетом, чтобы массивные методы сортировки можно было использовать для сортировки полиномов (см. раздел 4.9). В рамках этой задачи потребуется определить соответствующий порядок.

6.8 Сортировка по индексам и указателям

Работает ли тип данных `array`, показанном использованном в программах 6.9 и 6.10, представляет собой интересный вопрос, поскольку строки символов можно использовать как ключи и процесс сортировки. Более того, несколько строк могут иметь различные длины и поэтому быть очень длинными, поэтому, удаление и группировка строк может потребовать значительных затрат ресурсов, так что следует проявлять особую осторожность и позаботиться о том, чтобы выбранная реализация не приводила к значительным и несомненным затратам энергии.

С этой целью принимается такое представление данных, которое состоит из указания (на основе символов) — это стандартное представление строк в языке С. Далее, первая строка программы 6.9 помещается на

```
typedef char* tchar; /* tchar — тип.
```

что обеспечивает ее прототипизацию и интерфейс для строк. Этот указатель помещается в структуру `string`, поскольку С++ не позволяет перегружать операцию присвоения для переменных типа, как бы они являлись указателями. Подлинного рода синтаксис не является чем-то необычным в С++: класс `string` (`std::string`), который предоставляет интерфейс для другого типа данных называется классом-объектом. В рассмотряемом случае мы не требуем синтаксиса аналога от класс-объекта, тем не менее, в некоторых случаях возможны более сложные реализации. Вскоре будет рассмотрен еще один пример.

Программа 6.11 представляет собой реализацию, ориентированную на строковые элементы. Перегруженная операция `operator<` легко реализуется с помощью функции сравнения строк из библиотеки С, но реализация функций `size` (`len`) представляется собой более трудную задачу, поскольку нельзя использовать распределенную память для строк. Программа 6.11 использует метод, который изучился в главе 3 (программа 3.17), описанный буфер в реализации этого типа данных. Другое решение предусматривает динамические распределенные памяти для каждой строки, реализованные реализацией класса, подобного классу `String` из библиотеки стандартных шаблонов, либо организацией буфера в клиентской программе. Можно воспользоваться любым из этих подходов (с соответствующими интерфейсами) для строчечных строк символов, работая по-прежнему на реализации сортировки на рассмотренные выше.

Мы сталкиваемся с необходимостью подобного выбора управления памятью всякий раз, когда пытаемся применить программы модульную структуру. Кто должен нести ответственность за управление памятью, соответствующее конкретной реализации конкретных типов объектов: клиентская программа, реализация типа данных или система? Не существует однозначности ответа на этот вопрос (некоторые разработчики языков программирования вообще считают это вопросом политики). Некоторые из современных систем программирования (включая некоторые реализации С++) содержат необходимые механизмы, обеспечивающие автоматическое управление памятью. Мы еще раз столкнемся с этой проблемой в главе 9, когда приступим к изучению реализации более сложных абстрактных типов данных.

Программа 6.11 представляет собой пример сортировки по возрастанию, в реализации которой в обобщенном виде мы сейчас перейдем. Другой принятой схемой сортировки `Heap` (потокоделательная) перемешанный элемент помещается в построенный поддерево массива, причем доступ к ключевым элементам осуществляется только для того, чтобы выполнить операцию сравнения. Предположим, что сортируемые элементы являются в массиве `data[0], ..., data[N-1]` и мы не хотим перемешивать их в силу тех или иных причин (связанных, из-за их строчечных размеров). Чтобы не делать эффекта отсрочки, реализуем схему, основанную на поддеревах элементов. Мы начинаем с массива `data[i]` для `i = 0, ..., N-1`. Другими словами, мы начинаем с того, что `data[0]` получает значение нулевого первого элемента данных,

$a[1]$ — старший элемент упоряд. и т.д. Четвертая сортировка является переупорядочиванием массива индексов таких образом, что $a[0]$ даст индекс элемента данных с наименьшим значением ключа, $a[1]$ — индекс элемента данных с вторым по наименьшему ключу из числа оставшихся и т.д. После этого эффект сортировки достигается тем, что доступ к элементам файла индексов — непрерывный, таким образом можно избежать массива в процессе выполнения сортировки.

Программа 6.11. Реализация типов данных для строковых элементов

Я реализовал для языка C тип данных строковых элементов. Для представления данных используется структура, которая содержит указатель на символ (он может быть нулевым). Поэтому функцию сортировки можно использовать для любого языка или на символах, представленных так же образом, что строки. В которых они указывают, следует для них делать в алфавитно-цифровом порядке. Чтобы подробно показать процесс организации работы, мы даем определение структуры данных фиксированного размера, в которой данный указатель указывает на начало сортируемого строки по-прежнему. Функция `strcmp` расширяется тем, что позволяет больше. Реализация функции `strcmp` здесь отсутствует.

```
#include <string.h>
#include <stdlib.h>
#include <string.h>
#include "Index.h"
static char buf[100000],
static int cnt = 0;
int strcmp(const Index a, const Index b)
{ return strcmp(a.str, b.str) < 0; }
void show(const Index a)
{ printf("%s\n", a.str); }
int main(void)
{ int flag = 0; do { printf("Enter string: ");
  char *str = malloc(100);
  scanf("%s", str);
  } while (flag < 1); }
```

Требуется только определить, что указатель на элемент массива выполняет упорядочивание массива индексов, а не простое четное число. Для Index можно определить так, чтобы можно было переопределить функцию `strcmp` с помощью объектов:

```
int strcmp(const Index a, const Index b)
{ return a.str[0] < b.str[0]; }
```

Если мы получим в свое распоряжение массив объектов типа Index, то любой действующий алгоритм сортировки так переупорядочит индексы в массиве `a`, что значение $a[i]$ (индекс i — от 0 до $n-1$) будет меньше, чем значение $a[j]$ (индекс j — от 0 до $n-1$) в отсортированном массиве. (Для простоты в этом объяснении предполагается, что данные суть числа, а не элементы в полном смысле — это, конечно, можно распространить на более крупные и сложные элементы, включая и структуры данных, которые позволяют осуществлять доступ к элементу-числам, или к строкам, или к другим типам элементов, либо использовать функцию-член для получения такого значения.) Для определения объектов типа Index используется класс-объект


```
struct IntKlapper
{
    int item;
    IntKlapper(int i = 0)
    { item = i; }
    operator int() const
    { return item; }
};

// def IntKlapper Index:
```

Конструктор данной структуры `struct` преобразует любое значение типа `int` в `Index`, а оператор преобразования типа `operator int()` преобразует любое значение `Index` обратно в `int`. Следовательно, объекты типа `Index` можно использовать везде, где допускается применение объектов внутреннего типа `int`.

Пример наглядно, в рамках которого один и те же элементы сортируются по двум различным ключам, приведен на рис. 6.14. Одна индексная программа может определить порядок для работы с одним набором ключей, а другая — для использования с другим набором. При этом оба они могут использоваться одной и той же программой сортировки, чтобы построить массив ключей, который позволит бы получить доступ к элементам в порядке расположения их ключей.

Переупорядочивание N различных неотрицательных целых чисел, лежащих в диапазоне M , в математическом называется перестановкой: индексная сортировка выполняет перестановку. В математике перестановка обычно определяется как перемещение элементов целых чисел от 1 до M , мы же будем употреблять числа от 0 до $M - 1$, чтобы подчеркнуть прямую связь между перестановками и массивом индексов в $C++$.

Такой подход, предусматривающий использование массива индексов вместо реальных элементов, работает только в том языке программирования, который поддерживает массивы. (Для полноты completeness закончится в следующем разделе.) Она принципиально отличается от рассмотренной ранее сортировки на основе указателей (программа 6.1). Будучи типизированной на основе элементов фиксированного размера, сортировка по указателям во многих эквивалентна индексной сортировке, но при этом адрес массива относится к каждому элементу. Но сортировка по указателям — это более общая форма сортировки, ибо указатели могут указывать не что угодно, а элементы, подлежащие сортировке, отнюдь не обязательно должны иметь фиксированные размеры. Для индексной сортировки характерно существование ссылок с сортируемым указателем на ключ, то результатов работы функции `sort` будет переупорядоченные указатели такие объекты, что где

с	п	к	Имя	сб
1	2	2	Johnson	10
2	5	1	Smith	10
3	6	5	Smith	05
4	8	4	Washington	34
5	7	3	Thompson	03
6	9	3	Brown	30
7	3	10	Johnson	10
8	4	7	Wright	20
9	0	7	Smith	10
0	1	7	Black	70

РИСУНОК 6.14. ПРИМЕР ИНДЕКСНОЙ СОРТИРОВКИ

Каждый индекс указывает, в каком порядке элементы должны быть расположены. В этом случае данные могут быть формально сортированы по сравнению с их значением, тогда можно достигнуть более быстрого индексной сортировки по значению, а затем изменить порядок сортировки по значению. Например, `sort` — функция, которая сортирует массивы, а `sort_index` — функция, которая сортирует массивы по значению, а по умолчанию сортирует по значению. Например, `sort` — функция, которая сортирует массивы, а `sort_index` — функция, которая сортирует массивы по значению, а по умолчанию сортирует по значению. Например, `sort` — функция, которая сортирует массивы, а `sort_index` — функция, которая сортирует массивы по значению, а по умолчанию сортирует по значению.

последовательный доступ к ним исключает доступ к элементам соответствующему порядку. Мы вышнелишь описании сравнивать, с ними указывать, мы релнзуюе аннотации об менн ла счет перемены местным указателям.

Функция `qsort` из стандартной библиотеки C представляет собой сортировку по указателям (см. программу 5.17), которая принимает функцию сравнения и качестве аргумента (но не более) ее основы и завершающую операцию обмена (не обязательно). У этой функции четыре аргумента: массив, количество сортируемых элементов, размер элемента (он не указывается в функции, которая выполняет сравнение двух элементов, потому что этот ей требуется передать указатели на эти элементы). Например, если `Item` есть `char*`, то приведенный ниже программный код реализует сортировку строк в соответствии с принятым соглашением:

```
int compare (void *i, void *j)
{ return strcmp(*(Item *)i, *(Item *)j); }
void sort (Item a[], int l, int r)
{ qsort(a, r-l+1, sizeof(Item), compare); }
```

Полученный в итоге эффект для любого интерфейса не изменился, хотя бы стран сортировки (см. главу 7) и имеет широкое применение. В главе 7 мы рассмотрим много принципов, потому что так. Благодаря универсальной данной связи, а также глава 7 по 11, станет понятно, почему в условиях некоторых специальных приложений целесообразно применение ряда других методов сортировки. Кроме того будем использовать параллельно и ускорению вычисления в тех случаях, когда время исполнения сортировки является критическим фактором для приложения.

Программа 6.12. Интерфейс типа данных для элементов типа запись

В записи имеются два слова: ключ строкового типа (название, фамилия) и значение (число или строка) (наимено, размер) — то есть. Будем считать, что это значение строком больше, чтобы мы копировать, потому что оно определяется как структура `struct`, содержащая указатель на запись.

```
struct record { char name[30]; int num; };
typedef struct { record *r; } Item;
int operators (const Item, const Item),
void swap(Item);
void show(const Item);
int len(Item);
```

Программа 6.13. Реализация типа данных записей

Приведенные ниже реализации функций `swap` и `show` для записей реализованы в соответствии с требованиями из программы 6.1, реализованы и реализованы в памяти, в которой хранятся строки. Реализация операции обмена находится в `record.h` файле. Что касается `show` функции, то она реализована без использования константы `sizeof`.

```
static record data[1000];
static int cnt = 0;
void show(const Item x)
{ while (x.r->num < 0 || x.r->num > 999) { }
int swap(Item x)
{
    x.r = &data[cnt++];
    return (cnt > 1000 ? x.r->num : x.r->num) != 0;
}
```


В обычных приложениях указатели используются для доступа к записям, которые могут иметь несколько ключей. Например, логин, социальный номер и т.д. В зависимости от функции или функции логин вместе с возрастом могут определять статус или статус.

```
#include <vector> | #include <string> | #include <string> |
```

Аналог может использоваться для сортировки, который может быть из списка или в качестве ключа. Программы 6.12 и 6.13 могут служить примерами интерфейса сортировки по указателям и рекурсии, которая позволяет им достигнуть этого. Для различных приложений сортировка используется для указания на запись и различные решения операций. Например, если отключить программу 6.13 вместе с файлом, содержащим программный код

```
#include "Item.h"
int compare(const Item &a, const Item &b)
{ return a.getAge() < b.getAge(); }
```

то получим тип данных для элементов, для которых любая реализация функции sort будет сортировать по возрастанию или целочисленному полю, а если отключить программу 6.13 вместе с файлом, содержащим программный код

```
#include "Item.h"
#include <string.h>
int compare(const Item &a, const Item &b)
{ return strcmp(a.getName(), b.getName()) < 0; }
```

то получим тип данных для элементов, для которого любая реализация функции sort будет сортировать по возрастанию на строковой строке.

Основная причина использования индексов для указания состоит в том, чтобы не загромождать данные, подверженные сортировке. Можно "сортировать" файл даже в том случае, когда к нему разрешен доступ "только для чтения". Более того, используя небольшие индексные массивы или массивы указателей, можно сортировать даже и тот же файл по многим ключам (см. рис. 6.14). Подобного рода гибкость, позволяющая легко переключаться на различные индексы, позволяет легко применять приложения.

Другая причина, побуждающая использовать индексы, заключается в том, что такую работу можно избежать, разложив на переменные логические значения, используя различные экономии. Если индекс большой (а ключи маленькие), поскольку для индексации операции сравнения требуются доступы только к небольшой части данных, а большая часть данных в процессе выполнения сортировки не затрагивается. При использовании метода индексов операция примерно равна стоимости операции сравнения в общей сумме. Хотя сравнительно небольшие промежутки времени (такие, как разрыв из дополнительного пространства памяти, выделяемого под индексы и их указатели). В самом деле, если ключи длинные, то операции обмена могут быть даже не такими дорогостоящими, как операции сравнения. При получении списка элементов индексная сортировка достигается с помощью файла или списка элементов. Часто предполагается, что стоимость операций сравнения и обмена представляется относительно низкой. Иногда, получившиеся на основе такой предположения, можно, используя, отнести к широкому классу приложений в тех случаях, когда применяется сортировка по указателям или индексная сортировка.

тое место займем в массиве $\text{data}[a[k]]$, таким образом, k устанавливается в $a[k]$. Поэтому отсюда следует, что в случае выхода программы в ситуацию, когда пустое место должно быть занято значением $\text{data}[i]$, которое было предварительно сохранено. Когда мы помещаем элемент в конкретный индекс, мы помещаем соответствующее значение в массив a . Для любого элемента, внешнего для массива, $a[i]$ равен i , и так что непозволительный процесс сводится к отсутствию операции {за-от}. Переходящее из массива в массив новый указатель k , когда встречается элемент, который уже ни разу не переменялся, мы переменяем важный элемент еще большее число раз. Поэтому в 14 строчке мы рассмотрим рассмотренный процесс:

Программа 6.14. Обменная сортировка

Массив $\text{data}[0], \dots, \text{data}[N-1]$ должен быть упорядочен на месте в соответствии с массивом индексов $a[0], \dots, a[N-1]$. Пусть a — массив, для которого $a[i] = i$, значения которого скрываются от места и эта запись не следует читать. В представлении списка между значениями $\text{data}[i]$ и a есть связь $a[i] = j$ и $a[j] = i$, $a[a[i]] = i$ и т. д. до тех пор, пока не достигнется начало. Мы построим этот процесс для ведущего значения a , которое не является нулем, и добавляем к нему единицу, так как мы применяем параллельно себе файл, причем каждая запись будет переключаться только один раз.

```

exchange (Index Index, Index a[], int N)
{
  for (int i = 0; i < N; i++)
  {
    Index v = data[i];
    int j, k;
    for (k = i; a[k] != v; k = a[j], a[j] = j)
      j = k; data[k] = data[a[k]];
    data[k] = v; a[k] = k;
  }
}

```

Этот процесс является достаточно сложным по своему (6.14) и не является простым файлом. Отметим еще раз, несмотря на то что код по себе является весьма интересным, но на месте, во многих приложениях — не имеет необходимости, ибо процесс достаточно прост и легко реализуется в языке. Кроме того, если значение a является нулем, то можно по отношению к k поменять, а также индекс a может оказаться нулем и упорядочивать при помощи обобщенной сортировки выбором (см. главу 6.1).

Выборная сортировка требует меньше элементов и требует для разбиения массива индексов a и значения указателей и добавления времени для выполнения операции прямого сравнения. Во многих приложениях эти затраты являются вполне оправданными, если только не требуется к каждому элементу a доступ. В случае файла, состоящего из большого файла, мы практически не сможем использовать прямую сортировку, а во многих приложениях часто приходится так, что больше нет необходимости переключать данные. В рамках этой книги обычно применяется прямой доступ к данным. Однако в некоторых приложениях не так просто использовать индексацию массивов и часто требуется переключать данные, которые имеют отношение к ним. Мы не будем останавливаться на этой теме.

Упражнения

6.57. Предложите решение типа данных для элементов, когда элементная ячейка загнута, а не указывает на другую. Такая организация данных может оказаться предпочтительней для применения алгоритмов 6.12 и 6.13 к любым данным. (Не забывайте, что язык C++ поддерживает операции с произвольными структурами.)

6.58. Покажите, как можно реализовать функцию `front` для решения задачи сортировки, на которую ориентированы алгоритмы 6.12 и 6.13.

6.59. Построить массив индексов, который получается при индексной сортировке ключей `E A S Y Q U E S T I O N`.

6.60. Построить последовательность перемещений данных, необходимых для перемещения ключей `E A S Y Q U E S T I O N` на места после типоператора индексной сортировки (см. упражнение 6.59).

6.61. Опишите перестановку размера N (набор значений для массива `a`), в которой значение `a[i] + i` в процессе работы программы 6.14 никогда не становится отрицательным числом.

6.62. Показать, что в процессе перемещения ключей и перестановки в программе 6.14 пустых мест мы обязательно вернемся к началу, с которого начинали.

6.63. Реализовать программу, аналогичную программе 6.14, с сортировкой на указателях в соответствии с тем, что указатели указывают на массив из N значений типа `int`.

6.9. Сортировка связанных списков

Как стало известно из главы 4, массивы и связанные списки представляют собой два базовых способа структурирования данных. Поэтому рассмотрим реализацию сортировки методом связанных списков как пример обработки списков, о которой шла речь в разделе 3.4 (программа 3.11). Как программа реализует сортировку, рассмотрим к этому моменту, предполагая, что сортируемые данные представлены в виде элементов, в связи с чем данные могут быть несортированы непосредственно, если мы работаем в рамках системы, которая использует связанные списки для хранения данных. В некоторых случаях могут оказаться полезными специализированные алгоритмы, например, когда списки могут выполнять последовательную обработку данных, которую можно эффективно поддерживать для связанных списков.

Программа 6.15. Определение интерфейса для типа связанного списка

Данный интерфейс для связанных списков может быть использован с интерфейсом для массивов, представленным в программе 6.7. Функция `randolist` строит список из `n` равномерно распределенных элементов с одновременным выделением для них памяти. Функции `insert` и `insertAt` вставляют элемент `val` в указанный индекс. Программы могут также использовать перестроенную операцию `operator<` для сортировки и начальной сортировки с помощью упорядочивания элементов. Представленная реализация в рамках реализации с помощью указателей включает конструктор списка, который создает список из `n` элементов `val` заданным значением `val` и задает фиксированный список `next` `val`.

```
1 11 int: 11 int: 11 int: 11 int:
```



```

void listOver(K)
{
    if (isEmpty() == 0) next = 0;
}

// Insertion
void insert(int x)
{
    int temp = x;
    int k = 0;
    while (k < size)
    {
        if (arr[k] > temp)
        {
            shiftOneRight(k);
            arr[k] = temp;
            break;
        }
        k++;
    }
}

// Deletion
void delete(int x)
{
    int k = 0;
    while (k < size)
    {
        if (arr[k] == x)
        {
            shiftOneLeft(k);
            size--;
            break;
        }
        k++;
    }
}

// Display
void display()
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver Code
int main()
{
    listOver<int> l;
    l.insert(10);
    l.insert(20);
    l.insert(30);
    l.insert(40);
    l.insert(50);
    l.insert(60);
    l.insert(70);
    l.insert(80);
    l.insert(90);
    l.insert(100);
    l.delete(10);
    l.delete(20);
    l.delete(30);
    l.delete(40);
    l.delete(50);
    l.delete(60);
    l.delete(70);
    l.delete(80);
    l.delete(90);
    l.delete(100);
    l.display();
    return 0;
}

```


В текстовых ситуациях, когда информация на экран обрабатывается списком, важно не только обеспечить в явной реализации сортировки. Например, решено внести изменения в список с определенным порядком и включить новые элементы в список алфавитно тому, как это должно быть при сортировке вставки. Такой подход требует незначительных дополнительных затрат, если остатки проиндексированы сравнительно редко, либо если список имеет небольшой размер, а также в ряде других случаев. Например, если же иной причине необходимо изменить порядок всего списка (например, как показано в новой версии языка), чтобы убедиться в том, что в списке нет дубликатов. В главе 14 рассматривается алгоритм, который использует упорядоченные списки с списками, а в главах 13 и 14 рассматриваются многофункциональные структуры данных, эффективность которых повышается благодаря внутреннему порядку.

Упражнения

- 6.64. Исследовать содержимое модуля `std::sort` и убедиться, что сортировка ключей `ASORTINGEXAMPLE` выполняется с использованием программы 6.14.
- 6.65. Разработать реализацию интерфейса главного списка, заданного в программе 6.15.
- 6.66. Написать клиентскую программу, предоставляющую удобный интерфейс для создания эффективной сортировки связанных списков (см. упражнение 6.6).
- 6.67. Разработать АТД первого класса для связанных списков (см. раздел 4.8), который можно использовать для хранения различных значений, конструктор и деструктор, операции переключения операций `prepend()`, `insert()` и `delete()` через переопределенные операторы `operator+`, `operator-`, конструктор `list()` и функцию `clear()`. Постарайтесь сделать выбор для реализации функции `list()`, так что `list()` рассматривается как приватная функция члена.
- 6.68. Разработать программную реализацию универсальной сортировки для связанных списков. Проясните, почему `list()` и `clear()` являются в связанном списке более сложными операциями, так может показаться на первый взгляд.
- 6.69. Включите в программу 3.11 модуль сортировки, используя такой подход, чтобы он работал таким же функциональным элементом, что и программа 6.16.
- 6.70. Выпишите сортировки, представленные в программе 3.11, используя сортировку связанных списков. Сравните их с сортировкой массивов на несортированных входных файлах. Дайте описание работы на таких файлах и объясните, почему это так.
- 6.71. Проанализируйте сложность вычисления сложности методов `Sort`, ориентированных на связанные списки, чтобы не требовалось больше памяти и времени для сортировки случайно упорядоченного файла, чем для сортировки, предназначенной для сортировки массивов. Сравните результаты с сортировкой пузырьковой сортировкой.
- 6.72. Разработать АТД для последовательности, которая должна бы использоваться для хранения списка программ для сортировки реализаций сортировки как списков, так и массивов. Иначе говоря, клиентская про-

транши могут сгруппироваться последовательно из N элементов (в результате генерации случайного распределения совокупностей элементов либо с помощью псевдо). Необходимо сортировку последовательности и отображать ее элементы. Например, АТД в файле SEQ.txt должна работать со следующими программами АТД:

```
#include "Time.h"
#include "SEQ.txt"
main(int argc, char *argv[])
{
    int N = atoi(argv[1]), n = atoi(argv[2]);
    if (n) SEQrand(N, n);
    SEQprint();
    SEQshow();
}
```

Получить две реализации: одну, использующую предположение в виде массива, и другую, использующую представление в виде связанного списка. Воспользоваться сортировкой выбора.

- 6.74. Расширить задачу на упражнение 6.73 так, чтобы это была АТД одного класса. Составить решение вместе с библиотекой стандартных шаблонов.

6.10. Метод распределяющего подсчета

Потенциальные эффективности некоторых алгоритмов сортировки достигаются за счет использования специфических свойств данных. Например, рассмотрим такую задачу: требуется организовать сортировку файла из N элементов, ключи которых принимают значения 0 или 1 (двухэлементный массив $0 \leq i < N - 1$). Мы можем решить эту проблему сразу, воспользовавшись этой идеей массива b , посредством оператора:

```
for (i = 0; i < N; i++) b[beg[i] * 2] = a[i];
```

То есть, мы выстроили сортировку, используя массив как домены, а не как абстрактные элементы, которые соотносятся между собой. В этом разделе мы знакомимся с элементарными методами, которые используются как ключевая идея для повышения эффективности сортировки в случае, когда ключами служат некие числа, применяющиеся для разделения элементов на группы.

Если все ключи равны 0 , то сортировка тривиальна: теперь предположим, что для различных ключей принимают значения 0 и 1 . Такая проблема сортировки может возникнуть, когда требуется организовать элементы файла, удовлетворяющие некоторым (возможно, довольно сложным) произвольным критериям допустимости. Мы считаем, что значение 0 ключа означает, что элемент "принят", а значение 1 — что элемент "отвергнут". Одним из способов сортировки состоит в том, что сначала создаем массив b , затем заполняем второй массив по следующему правилу: в b поместить его элемент i массива a , при этом предусматривается массив, состоящий из двух элементов, первый из которых используется следующим образом. Мы помним о том, что поместим 0 в $b[0]$, а количество нулевых ключей в файле — в $b[1]$. А должно показывать, что в рассматриваемом файле не существует ключей, принимающих значения меньше 0 , ни имеется $b[1]$ ключей, значения которых равны 1 . Разумеется, мы можем применить массив b следующим образом: в начале массива b поместим

Упражнения

- а 6.74. Дать пример стационарной сортировки метода распределения подсчета для сортировки файлов, элементы которых могут принимать только одно из трех значений (А, В или С).
- 6.75. Предположим, что используется сортировка вставки для упорядочивания случайно расположенного файла, элементы которого принимают одно из трех возможных значений. Какой записи будет больше всего времени вычисления сортировки, линейной, квадратичной или некоторой промежуточной записи?
- б 6.76. Показать, как файл А В В А С А В А В В А сортируется при помощи метода распределения подсчета.
- 6.77. Реализовать сортировку методом распределения подсчета элементов, которые представляют собой целочисленно больше или меньше с целочисленным значением, принимающим значения в некотором диапазоне.
- 6.78. Реализовать сортировку методом распределения подсчета в виде сортировки по указателям.

Быстрая сортировка

Темой настоящей главы является алгоритм сортировки, который, по-видимому, используется гораздо чаще любого другого. Именно — алгоритм быстрой сортировки (quick-sort). Новый алгоритм этого вида сортировки был открыт в 1959 г. Хоаром (C.A.R. Hoare), и с той поры многие специалисты по-прежнему своим лучшим занятием считают его (см. раздел «Задачи»). Быстрая сортировка стала популярной прежде всего потому, что ее алгоритм реализовать, она хорошо работает на различных видах входных данных и во многих случаях требует меньше затрат ресурсов по сравнению с другими методами сортировки.

Алгоритм быстрой сортировки обладает и другими весьма примечательными особенностями: он принадлежит к категории обходных (in-place) сортировок (т.е. требует всего лишь небольшого дополнительного пространства), вычислительная сложность — средняя (такая, какая была, например, у N^2), а для него характерна несомненно короткая внутренняя длина. Это несомненно является тем, что не хватает, для его применения как в самых сложных случаях (объемы N^2 операций), так и в тех случаях, что даже простая ошибка в реализации может привести к значительной потере эффективности работы алгоритма на некоторых видах файлов.

Работа быстрой сортировки проста для понимания. Алгоритм был подвергнут тщательному математическому анализу, и можно дать достаточно точную оценку его эффективности. Этот анализ был подтвержден экспериментальными эмпирическими экспериментами, а сам алгоритм был усовершенствован до такой степени, что ему удалось преодолеть и широчайшие диапазоны практических

применений сортировки. На этой странице подчёркнуто, что для быстрой сортировки много эффективной реализации алгоритма быстрой сортировки, превосходящих по скорости решениям других алгоритмов. Авторитетные методы, реализованные целенаправленно, способны конкурировать с другими алгоритмами, помещёнными в быструю сортировку, и мы можем полагаться с большей уверенностью, поскольку удалось возложить в алгоритм программы, каковы бы ни были, на эффективность сортировки.

Интересно попытаться разработать способы улучшения быстрой сортировки, чем выше безразличны алгоритмы сортировки, тем привлекательнее выглядит возможность применения этих способов, а быстрая сортировка представляет собой оптимальный метод, который лишь учитывает это предпочтение. Практически с момента опубликования Хоаром алгоритма быстрой сортировки в литературе стали регулярно появляться его усовершенствованные версии. Проведя анализ и анализируя различные исследования, но при этом совсем не трудно убедиться при попытке даже улучшить, тем самым, что для шортрига важно хорошо сбалансировать, что эффективно усложняет выполнение одной части программы может послужить при этом ухудшению функционирования другой её части. Мы должны изучить три модификации, которые существенно попытает повысить эффективность быстрой сортировки.

Тщательно сбалансированная версия быстрой сортировки, по своей вероятности, будет превосходить быстрее любого другого метода сортировки на большинстве компьютеров, к тому же быстрая сортировка широко используется как либо часть или программа сортировки и для других связанных приложений сортировки. В общем деле, сортировка не стандартной либо полноты. С 1950-х годов возникла (быстрая) сортировка, ибо обычно название алгоритма быстрой сортировки лежит в основе различных реализаций. Однако, при выполнении быстрого сортировки часто отсортированы входных данных и каталогизация между линейной и квадратичной зависимостью от количества сортируемых элементов и повышается тем, что бывает не только длительность выполнения, но и сложность реализации. Нетрудно получить результаты сортировки, каковы бы ни были входные данные, особенно когда используются хорошие отложенные версии этого алгоритма. Если приложение работает с данными, то можно использовать в качестве дефекта в реализации быстрой сортировки, то сортировка методом Шелла может оказаться более удобным выбором, обеспечивающим лучший результат при меньших затратах на реализацию. Следует отметить, что в отличие от быстрой сортировки метод быстрой сортировки применяется примерно в пятьдесят раз быстрее сортировки методом Шелла, для этого на некоторых персональных компьютерах достаточно встретившихся на практике, может быть достигнута ещё большая эффективность данного вида сортировки.

7.1. Базовый алгоритм

Быстрый метод сортировки функционирует по принципу "разделяй и властвуй". Он делит сортируемый массив на две части, затем сортирует эти части независимо друг от друга. Как было показано выше, время, необходимое для сортировки, зависит от исходного порядка элементов во входном файле. Суть метода заключается в процессе разбиения файла, который переупорядочивает файл таким образом, что выполняются следующие условия:

В конечном итоге эффективность сортировки зависит от качества разбиения файла, которое, в свою очередь, зависит от выбора элемента разбиения. Именно это показано в разделе 7.2. В идеальном случае разбиение файла в три части происходит так: одна часть с произвольной принадлежностью к каждой из частей, вторая часть разбита также произвольно в любую часть файла. Мы предпочтем выбрать такую точку разбиения (ближе к середине файла), однако это разбиение не обязательно для этого разбиения. Если сортируем файл с произвольной принадлежностью, то выбор элемента $a[i]$ в качестве разделяющего — это то же самое, что и выбор любого другого конкретного элемента, он дает нам в любом случае точку разбиения в некоторой близости от середины. В разделе 7.4 проанализируем рассмотренного алгоритма, который исключает такой случай с элементами выбора. В разделе 7.5 будет показано, насколько разбиение файла (какая может показаться полезная при выборе разделяющего элемента в целях повышения эффективности рассмотренного алгоритма).

Упражнения

- 7.1. Показать в стиле рассмотренного здесь примера, как быстрая сортировка сортирует файл: K A N Y Q H E S I U M.
- 7.2. Показать, как происходит разбиение файла 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1. Показать для этой цели программу 7.2 и соответствующие конфигурации. Показать то же самое.
- 7.3. Реализовать разбиение. Не прибегая к помощи оператора break или goto.
- 7.4. Разработать условия быстрой сортировки для списков.
- 7.5. Каким образом максимизировать число переключений наибольшего элемента файла, состоящего из N элементов, на своем месте при быстрой сортировке?

7.2. Характеристики производительности быстрой сортировки

Несмотря на все теоретические качества, быстрая программа быстрой сортировки suffers от определенных недостатков, которыми так называется — тем, что она неэффективна для некоторых простых файлов, которые могут возникнуть на практике. Например, если она применяется для сортировки файла размером N , который уже отсортирован, то она неизбежно разобьет файл на произвольные и произвольные длины: сама себя N раз, переключая на каждый выход всего лишь один элемент.

Лемма 7.1. Быстрая сортировка в худшем случае требует порядка $N^2/2$ операций.

В силу того что принадлежность элемента к числу внутренних сравнения, выполняемых при сортировке, как оговоренного файла, выражается как

$$x = (N - 1) + (N - 2) + \dots + 2 + 1 = (N + 1)N/2.$$

В худшем случае выражаемая как в случае файла, отсортированного в обратном порядке, так и в случае файлов перевернутых входов, вероятность сокращения с фактором $1/2$ при N увеличивается только на $1/2$ (упражнение 7.6).

Порядное исследование означает не только то, что время выполнения быстрой сортировки приблизительно определено величиной $N^2/2$, но и то, что пространство памяти, необходимое для выполнения этой процедуры, пропорционально N (см. раздел 7.2), что в случае крупных файлов невыполнимо. К счастью, имеются сравнительно простые способы существенного снижения потребности того, что называется «пространством», в памяти при выполнении рассматриваемых процедур.

Наиболее благоприятный для быстрой сортировки случай имеет место, когда на каждой стадии разбиения файл делится на две равные части. Это объясняется прежде всего тем, что количество операций сравнения, выполняемых в процессе быстрой сортировки, удовлетворяет рекуррентному соотношению типа «разделяй и властвуй».

$$C_N = 2C_{N/2} + N$$

Член $2C_{N/2}$ относится к затратам на сортировку двух подфайлов; N есть затраты на проверку каждого элемента, для чего производится тот или иной разделяющий указатель. Мы знаем уже из того же, что это рекуррентное соотношение имеет решение

$$C_N = N \lg N$$

Несмотря на то что не всегда все так удачно складывается, тем не менее, верно, что в худшем разделение файла на две части равно. Если же у нас точную потребность каждой подмассива разделения, то указанное выше рекуррентное соотношение становится более сложным и более трудным для решения. Однако окончательный результат примерно такой же.

Лемма 7.2. Быстрая сортировка в среднем выполняется $2N \lg N$ операций сравнения.

Точное рекуррентное соотношение для определения числа сравнений, выполняемых во время быстрой сортировки N случайно расположенных различных элементов, имеет вид

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq i < j \leq N} (C_{i-1} + C_{j-1}) \quad \text{для} \quad N \geq 2,$$

при $C_1 = C_2 = 0$. Член $N - 1$ учитывает затраты на выполнение операций обмена или раздвигания элемента с каждым из остальных элементов (для их сравнения в точке пересечения указателей); наличие остальных членов объясняется тем фактом, что каждый элемент k может стать разделяющим элементом с вероятностью $1/N$, после чего остаются файлы с максимальной глубиной, равносильно размеру $k - 1$ и $N - k$.

Это рекуррентное соотношение несложно решить, сложив, фактически доведя до разности разности, буквално за три действия. Вспомогательная $C_1 + C_2 + \dots + C_{N-1}$ есть то же самое как $C_{N-1} + C_{N-2} + \dots + C_1$; следовательно, имеем

$$C_N = N - 1 + \sum_{1 \leq i < j \leq N} C_{i-1}.$$

Во вторых, можно избавиться от сумм, если умножить обе части равенства на N и вычитать левую же формулу из $N - 1$:

$$NC_N = (N - 1)C_{N-1} + N(N - 1) - (N - 1)N - 2C_{N-1}$$

За счет такого упрощения рассуждениямне рекуррентное соотношение удобно

$$M(N) = (N + 1)C_N + 2N.$$

В прецед. разделив обе части на $N(N + 1)$, получим рекуррентное соотношение, которое принимает более компактную форму:

$$\begin{aligned} \frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N-1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \dots \\ &= \frac{C_2}{3} + \sum_{k=3}^N \frac{2}{k-1} \end{aligned}$$

Это также выражение почти важно было, которое можно аппроксимировать с помощью (см. раздел 2.1):

$$\frac{C_N}{N+1} = 2 \sum_{i=1}^{N-2} \frac{1}{i} + 2 \int_1^N \frac{1}{x} dx = 2 \ln N,$$

откуда следует приближенный ранее результат. Обратите внимание на то, что $2N \ln N = 1.38N \ln N$, так что данное значение операции сравнения приблизительно лишь на 39 процентов больше, чем в самом лучшем случае.

Данный анализ подразумевает, что сортируемый файл содержит случайные упорядоченные записи с различными ключами, однако реальные программы 7.1 и 7.2 могут работать быстрее в случаях, когда ключи не обязательно различны и не обязательно расположены в случайном порядке (рис. 7.4). Если сортировка применяется к информации для более или менее неадекватной для упорядочения цели, больше файла (или 2/3 его объема), если она должна использоваться как универсальная инвариантная функция для сортировки файлов с неизвестными характеристиками, следует рассмотреть несколько усовершенствований, предлагаемых в разделах 7.5 и 7.6, которые позволяют избежать того, что наихудший случай возникает на практике, и также уменьшают среднее время выполнения сортировки (не только на 20 и более %).

Упражнения

- 7.6. Построить шесть файлов из 10 элементов, при упорядочивании которых метод быстрой сортировки (программа 7.1) выполняет то же число операций сравнения, что и в самом худшем случае (когда все элементы файла упорядочены).
- 7.7. Написать программу, вычисляющую точное значение C_N и сравнить это точное значение с приближенным значением $2N \ln N$ для $N = 10^3, 10^4, 10^5$ и 10^6 .
- 7.8. Сколько примерно операций сравнения выполнит быстрая сортировка (программа 7.1) для упорядочивания файла из 8 различных элементов?

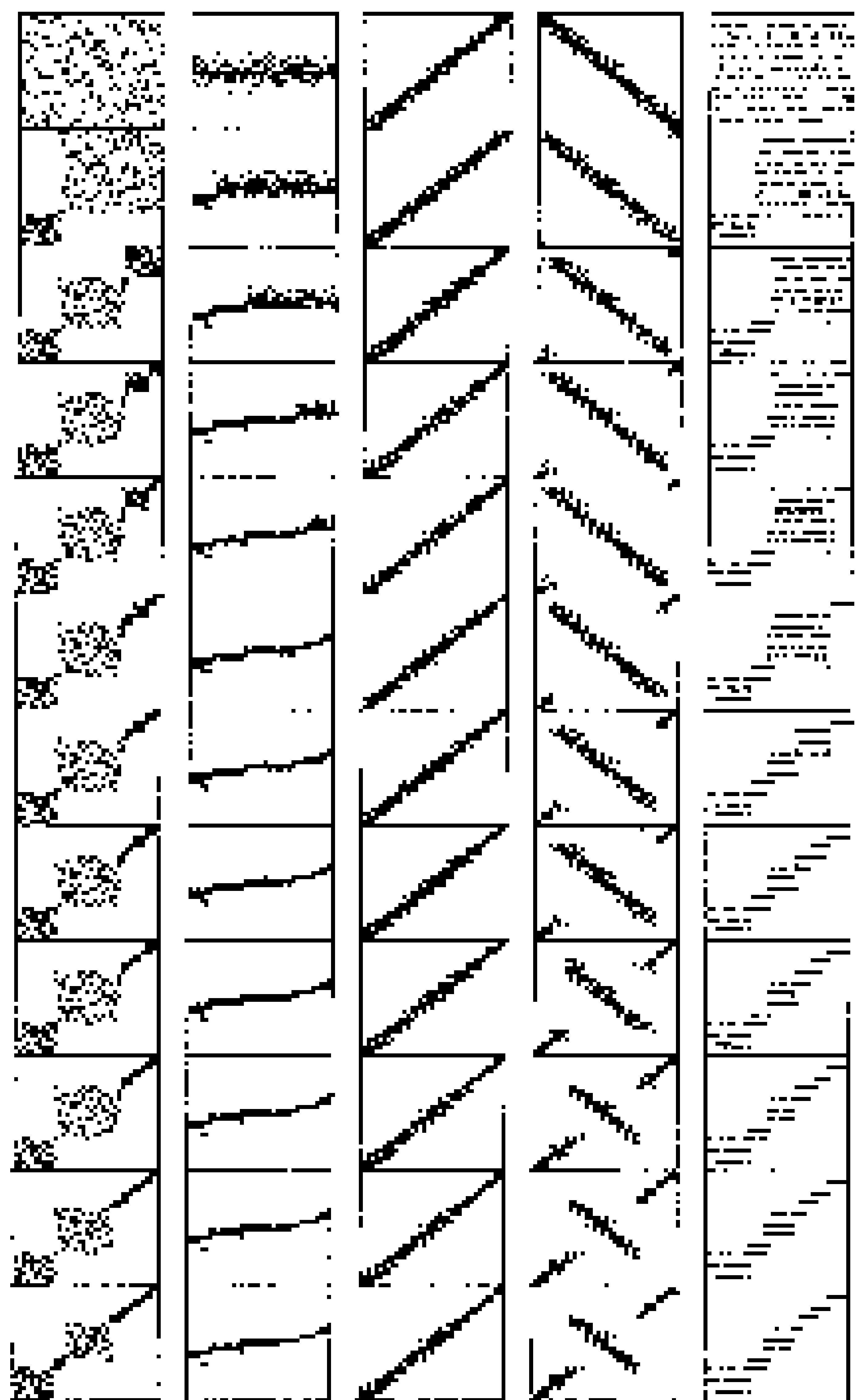


РИСУНОК 7.4. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ БЫСТРОЙ СОРТИРОВКИ НА ФАЙЛАХ РАЗЛИЧНЫХ ТИПОВ

Важно понимать, что разделение массива и выборный сортировка являются основными этапами работы быстрого сортировщика. Разделение массива является наиболее трудоемкой операцией, поэтому сортировка, основанная на файлах, будет работать быстрее, чем сортировка, основанная на массиве. Это связано с тем, что файлы позволяют хранить данные в виде последовательности записей, что позволяет избежать необходимости в дополнительных операциях по перемещению данных. Кроме того, файлы позволяют хранить данные в виде последовательности записей, что позволяет избежать необходимости в дополнительных операциях по перемещению данных. Это связано с тем, что файлы позволяют хранить данные в виде последовательности записей, что позволяет избежать необходимости в дополнительных операциях по перемещению данных.

Программа 7.3 представляет собой рекурсивную реализацию, которая решает задачу проблемы путем проверки размеров обоих подфайлов и размещения большего из них в стек первым. Рисунок 7.6 иллюстрирует такую стратегию. Сравнивая этот пример с приведенным на рис. 7.1, мы видим, что при такой стратегии подфайлы не меняются, меняется только порядок их обработки. Такая обстановка, мы сокращаем расход памяти без увеличения расхода времени.

Стратегия, которая применяется в том, что в стек помещается больший из двух подфайлов, приводит к тому, что каждая запись в стек съест больше чем половину элемента, представлявшего ее в файле, поэтому под стек отпадает пространство памяти, достаточное для размещения размером $\lg N$ элементов. Используя стек не максимумом, но с запасом, контроль разделения приходится на среднюю запись. Что касается файлов с произвольной организацией, то фактический максимальный размер стека намного меньше; для выровненных файлов это размер, по-прежнему, также будет небольшим.

Программа 7.3. Рекурсивная программная реализация быстрой сортировки.

Применяемая здесь рекурсивная реализация (см. рис. 7.3) использует по-прежнему стек вызовов (рис. 7.14). Задача разделения файла по размеру стека параметрична, а вызовы процедуры являются рекурсивными, которые осуществляют выборку параметров из стека и их обработку, пока стек не пуст. Мы поместим файлы из двух подфайлов в стек таким образом, чтобы максимальная глубина стека при N элементах не превосходила величины $\lg N$.

```
#include "Stack.h"
inline void push2(STACK<int> &s, int x, int y)
{ s.push(x); s.push(y); }
template <class T>
void quicksort(T& a[], int l, int r)
{ STACK<int> s(50);
  push2(s, l, r);
  while (!s.empty())
  {
    l = s.pop(); r = s.pop();
    if (y == 1) continue;
    int i = partition(a, l, r);
    for (i-1 > l-1)
      push2(a, l, i-1); push2(a, i+1, r);
    else
      push2(a, i+1, r); push2(a, l, i-1);
  }
}
```



РИСУНОК 7.6. ПРИМЕР БЫСТРОЙ СОРТИРОВКИ (ПЕРВЫМ СОРТИРУЕТСЯ МЕНЬШИЙ ПОДФАЙЛ)

Следовательно, в эту процедуру помещаются файлы, которые сортируются рекурсивно. Быстрая сортировка является эффективным алгоритмом сортировки, который позволяет сортировать файлы, которые имеют произвольную организацию. В результате сортировки файлы становятся упорядоченными, что позволяет использовать их для различных целей. Быстрая сортировка является одним из самых эффективных алгоритмов сортировки, который позволяет сортировать файлы, которые имеют произвольную организацию. В результате сортировки файлы становятся упорядоченными, что позволяет использовать их для различных целей.

В невырожденном случае ранг матрицы не должен превышать $2n$, так T_n удовлетворяет рекуррентному соотношению $T_n = T_{n-1} + 1$ при $T_1 = T_0 = 0$. Это рекуррентное соотношение эквивалентно утверждению, принадлежащему к типу, рассмотренному в главе 6 (см. упражнение 7.31).

[illegible]

порядок карт. Если элемент должен быть размещен на позиции, а также размещенные ранее элементы и карты, соответствующие текущему и следующему подфайлам, то выполняются следующие действия: левым и правым поменяем. Исполнение рекурсивной реализации, описанной в программе 7.3, завершается без просмотра этих элементов вправо и влево; перекурсивная реализация соответствует процессу просмотра ссылаемых элементов вправо.

Когда используется многоэлементный стек, что, собственно говоря, и делается в программе 7.3, требуется включить в категорию неэквивалентных строк, характерных для рекурсивных реализаций, хотя создаваемые элементы программы размещены на принадлежащих одному контрольному адресу памяти и имеют простые программы. Программу 7.3 можно улучшить в значительной мере. Например, она помещает в стек оба подфайла, но только один подфайл, другая часть и перешла в стек, достигнув в любой момент такой позиции, где уже можно заметить, что она является специализированной переменной $l < 1$. Итак, программа просмотра упрощена $l < 1$ по мере того, как подфайлы выбираются из стека, и в процессе их анализа происходит обработка подфайла не до конца в стек, удовлетворяя таким образом условиям, указанным в программе 7.14). Это мера, на первый взгляд, может показаться неуместной, однако рекурсивный характер быстрой сортировки фактически приводит к тому, что значительная часть подфайлов в процессе быстрой сортировки имеет размер 1 или 0. Таким образом, можно рассмотреть усовершенствование быстрой сортировки, которое обеспечивает получение ее эффективности на счет расширения этой части на все случаи, включая большие размеры.

Упражнения

- 7.11. В статье рис. 7.11 представить взаимодействие стека, когда каждой паре карт и карты (карты) в стек и взаимодействие (карты) с собой, когда программа 7.3 используется для сортировки файла, содержащего ключи E A S V Q U R S T I O X.
- 7.12. Выполнить задание, описанное в упражнении 7.11, для случая, когда в стек значится помещается первый подфайл, а затем второй подфайл (хотя это применение к рекурсивной реализации).
- 7.13. Завершить доказательство того, что тестирование для этой цели является эффективным.
- 7.14. Добавьте в программу 7.3 такие изменения, чтобы она не помещала в стек подфайлы, если только условие $l < 1$.
- 7.15. Выяснить максимальный размер стека, обработанного программой 7.3, когда $N = 2^5$.
- 7.16. Выяснить максимальный размер стека, обработанного программой 7.3, когда $N = 2^5 - 1$ и $N = 2^5 - 1$.
- 7.17. Вместо того чтобы использовать для некурсивной реализации быстрой сортировки, можно использовать алгоритм 1. Предложить аргументы для обоснования этого ответа.
- 7.18. Выясните и сообщите, практикует ли ваша система программирования этот стек стека по завершении рекурсивной процедуры.

порт, так же хорошо, как и в совокупности файлов большого размера, когда они подвержены сортировке статичным распределениям. При непосредственном доступе метод страдает от недостатка безопасности, когда сортировка выполняется сортом восточной работы, даже если алгоритм быстрой сортировки содержит фатальную ошибку, из-за которой эта сортировка вместо функционирования не будет. Только редкое возмущение затрат ресурсов может спасти ситуацию в том, что что-то не в порядке.

Рисунок 7.9 иллюстрирует этот процесс на примере звукового файла. Даже при сравнительно радикальном сжатии файлов звуковых ресурсов та часть программы, которая выполняет быструю сортировку, выполняется быстро, поскольку в процессе разделения заданного статически заданного количества элементов. Сортировка выполняется, завершаясь в основном в режиме, так же выполняется быстрая сортировка, так как обработка ей передается почти универсальный файл.

Этот метод с большой силой можно применить всякий раз, когда мы можем дело с рекурсивным алгоритмом. В силу особенностей рекурсивных алгоритмов можно быть уверенным в том, что все они в конечном счете спешат будут найти решение небольшого числа в любой ситуации для работы с любым файлом в нашей распоряжении известны алгоритмы "решения в лоб" с помощью непроизводительными затратами. Благодаря такому обстоятельству в общем случае можно улучшить общие показатели производительности с помощью гибридных алгоритмов.

Упражнения

- 7.21. Нужны ли служебные элементы, если сортировка статичными методами непосредственно не быстрой сортировки?
- 7.22. Спроектировать программу 3.1 инструментариями средствами, которые позволят бы подсчитать процесс от заданной функции или заданного файла, размеры которых не превышают 10^4 , 100 и 1000 элементов, и вывести результаты анализа, которые приняты для процесса сдвигания для случаев сортировки файлов с произвольной организацией, состоящих из N элементов для $N = 10^4$, 10^5 , 10^6 и 10^7 .
- 7.23. Рассмотреть рекурсивный вариант быстрой сортировки с отсечением для сортировки вставками профилей, размеры которых не превышают N элементов, и экспериментально определить значение N , при котором программа достигает максимальной скорости работы в задаче вычислительной среды при сортировке файлов с произвольной организацией, состоящих из N элементов для $N = 10^4$, 10^5 и 10^6 .
- 7.24. Решить задачу, сформулированную в упражнении 7.23, воспользовавшись не рекурсивной реализацией.
- 7.25. Решить задачу, сформулированную в упражнении 7.23, для случая, когда сортировка заданной области в которой n указателей на другие инфермальные объекты не используется сортировку по указателям).
- 7.26. Написать программу, которая вычисляет гистограмму (см. программу 3.7) для размеров подзаписей, поразделенных в сортировке вставками, при выполнении сортировки файлов размером N с заданным подфайлом с размерами, не превышающими N . Вычислить эту программу для $N = 10$, 100 и 1000 и $N = 10^4$, 10^5 и 10^6 .

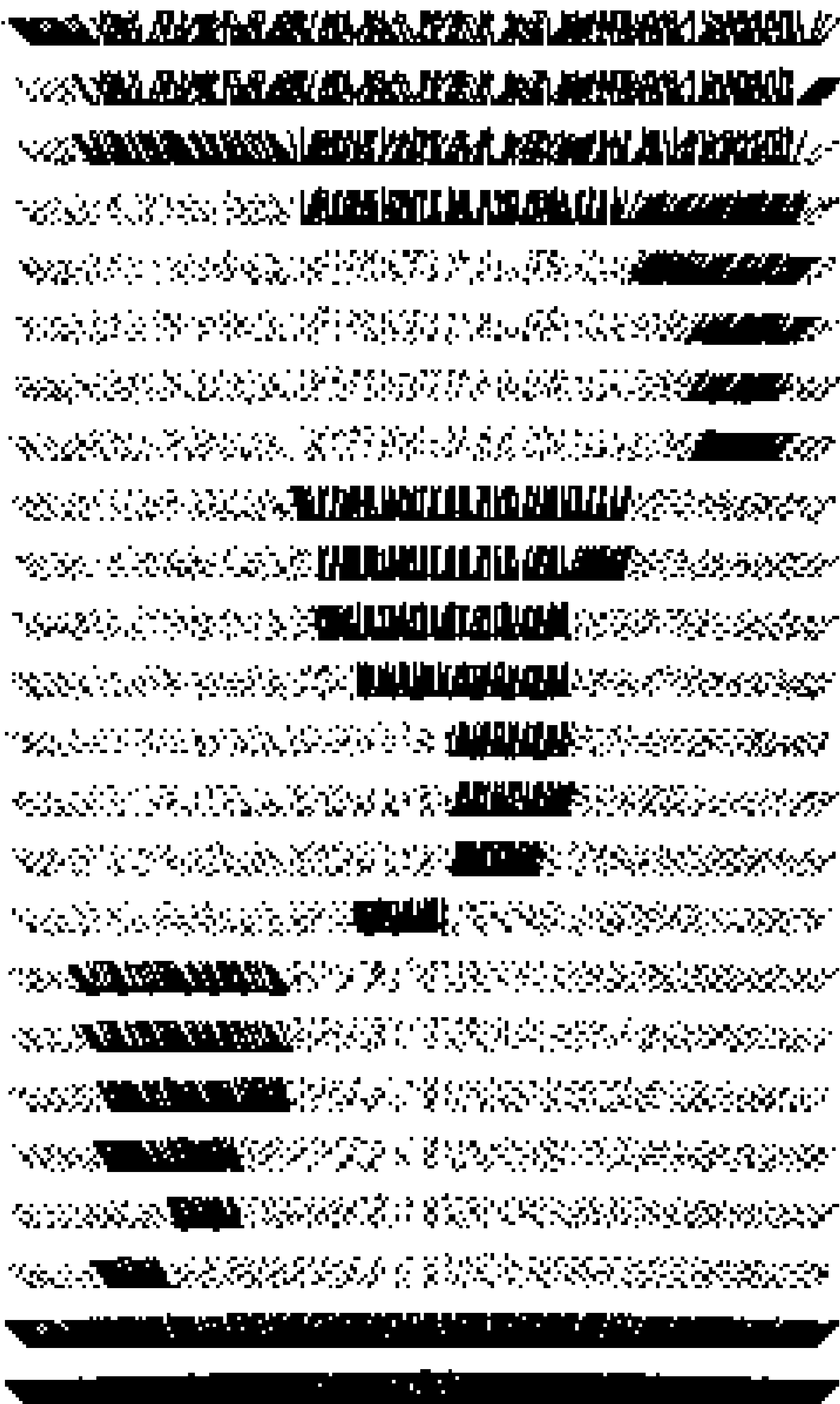


РИСУНОК 2.9. СРАВНЕНИЯ В БЫСТРОЙ СОРТИРОВКЕ

Получается, что количество быстрой сортировки относительно медленно увеличивается. На самом деле количество элементов, которые нужно сравнить, чтобы разделить массив, в худшем случае равно $2N$ (когда N — нечетное число). В среднем же требуется $1,5N$ элементов. Поэтому количество элементов, которые нужно сравнить, чтобы разделить массив, в худшем случае равно $2N$, а в среднем — $1,5N$. В худшем случае количество элементов, которые нужно сравнить, чтобы разделить массив, в худшем случае равно $2N$, а в среднем — $1,5N$.

7.27. Проведите эмпирические исследования с целью определения критического размера списка элементов при быстрой сортировке с отсечением подфайлов в зависимости от применяемого M для сортировки файла с произвольной организацией, состоящего из M элементов, причем $M = 10, 100$ и 1000 и $N = 10^3, 10^4, 10^5$ и 10^6 .

7.5 Метод разделения с вычислением медианы из трех элементов

Базисное усовершенствование метода быстрой сортировки заключается в выборе элемента, разделяющего элементы, которые с достаточной вероятностью являются близкими к средним. Наиболее близким к выбору оптимального элемента является использование в качестве разделяющего элемента случайного элемента массива. Такая вероятность выбора лучшего элемента становится потребностью нашей. Этот метод представляет собой пример сбалансированного алгоритма (сбалансированный алгоритм) — такого алгоритма, который не допускает случайный характер поведения при достижении высокой эффективности с большой вероятностью, независимо от степени упорядоченности входных данных. Дело в том, что мы сталкиваемся с несбалансированными примерами использования свойств случайности при разработке структуры алгоритмов. В частности, когда предполагается наличие той или иной тенденции на входных данных. На практике используются в рамках быстрой сортировки генераторы случайных чисел с той целью, может оказаться излишним: простой простейший выбор оказывается достаточно эффективным.

Другой хорошо известный способ нахождения лучшего элемента заключается в том, что производится выборка трех элементов из файла, затем в качестве разделяющего элемента используется медиана из этих трех элементов. Выбирая для такой цели три элемента из левой части, из середины и из правой части массива, можно также включить в эту группу служебные элементы, являющиеся элементами выбранных элементов (с помощью которых можно трех объектов, указанных в плане 6), затем выбрать элемент из середины с элементом $a[t-1]$, затем выбрать элемент из середины с элементом $a[t+1]$, и т.д. Проведение вычисления медианы из трех элементов можно считать эффективным.

Метод медианы из трех элементов объясняет эффективность сортировки с использованием M -элементов, он существенно снижает вероятность возникновения случайной для любой реальной сортировки. Чтобы сортировка выполнялась быстро, произвольные M' , или на трех произвольных элементах должны быть выбраны наиболее близкие к средним элементы файла, а это событие должно происходить достаточно часто во время большей части процесса решения. В старых сортировках необходимо было использовать много элементов, но только для выполнения этой функции одного достаточно одного элемента из числа тех, которые подвергаются процессу разделения. В-третьих, он увеличивает среднее время выполнения алгоритма примерно на 5 процентов.

Счетные исследования метода медианы из трех элементов с отсечением подфайлов небольших массивов может уменьшить среднее время выполнения быстрой сортировки по сравнению с алгоритмом, показанным в рассмотренной рекурсивной

сортиментом от 20—25 процентов. Программы 1.4 предусматривают также различные, в которых приняты все установленные государственные стандарты.

Можно рассмотреть и другие способы сохранения программы: отказ от рекурсии, замена вызовов подпрограмм вызовами функций, использование статических методов и т.п. Однако в современных машинах такое вызовов проще, обычно эффективнее и они не исключаются из внутреннего языка. Но более важно, применение функции подпрограммы требует меньших затрат во многих случаях по сравнению с тем, что требуется для генерации кода (за исключением генерации кода). Главные аргументы в пользу перекурсивного решения с ними следующие: они не требуют записываться в код, чтобы обеспечить гарантии, что можно было бы получить стек ограниченного размера (см. рис. 7.10).

Важно отметить, что улучшение алгоритма (на рисунке, можно использовать массивы по пятим или десятичному числу элементов), никак не влияет на количество времени для факции с произвольной произвольной перестановкой. Мы можем получить оптимально экстремальную перестановку за счет кодирования внутренних данных и их произвольной на языке ассемблера или на машинном языке. Он всегда будет многократно подтвержден спецификацией по примеру выполнения приложения: `admiral.exe (с. 100-101)`.

Друг фактор в представлении организатора программы — экономический программой 7.1. включенный. Мы включили это в программу не только из-за того, что он обеспечивает социальное взаимодействие для пользователей с сайтом, но еще и потому, что оно служит защитой от негативных ситуаций, связанных с возникновением и развитием (см. например, управление 7.10). Рисунок 7.11 иллюстрирует эффективность из-за отсутствия среднего уровня в процессе выбора пользователя для различных типов факторов.

Метод утешения на трех элементах представляет собой столкновительный случай обшей теории, заключающийся в том, что для выбора гравитационного типа можно произвести выборку и не использовать различия полученной выборочной совокупности. Чтобы дать пример более ясно:

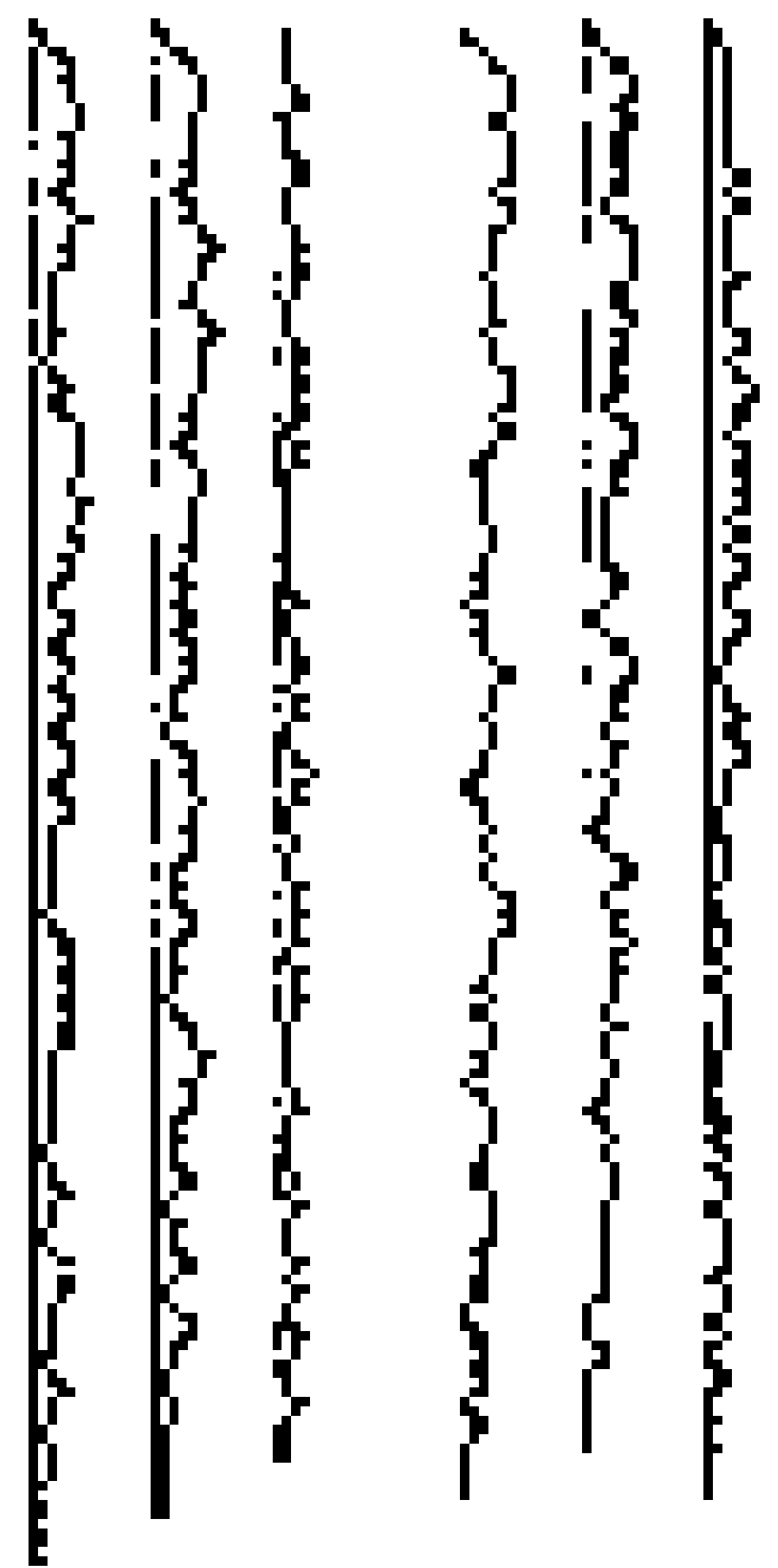


Рисунок 7.10. Размеры стеков для улучшенных вариантов быстрой сортировки

[illegible]

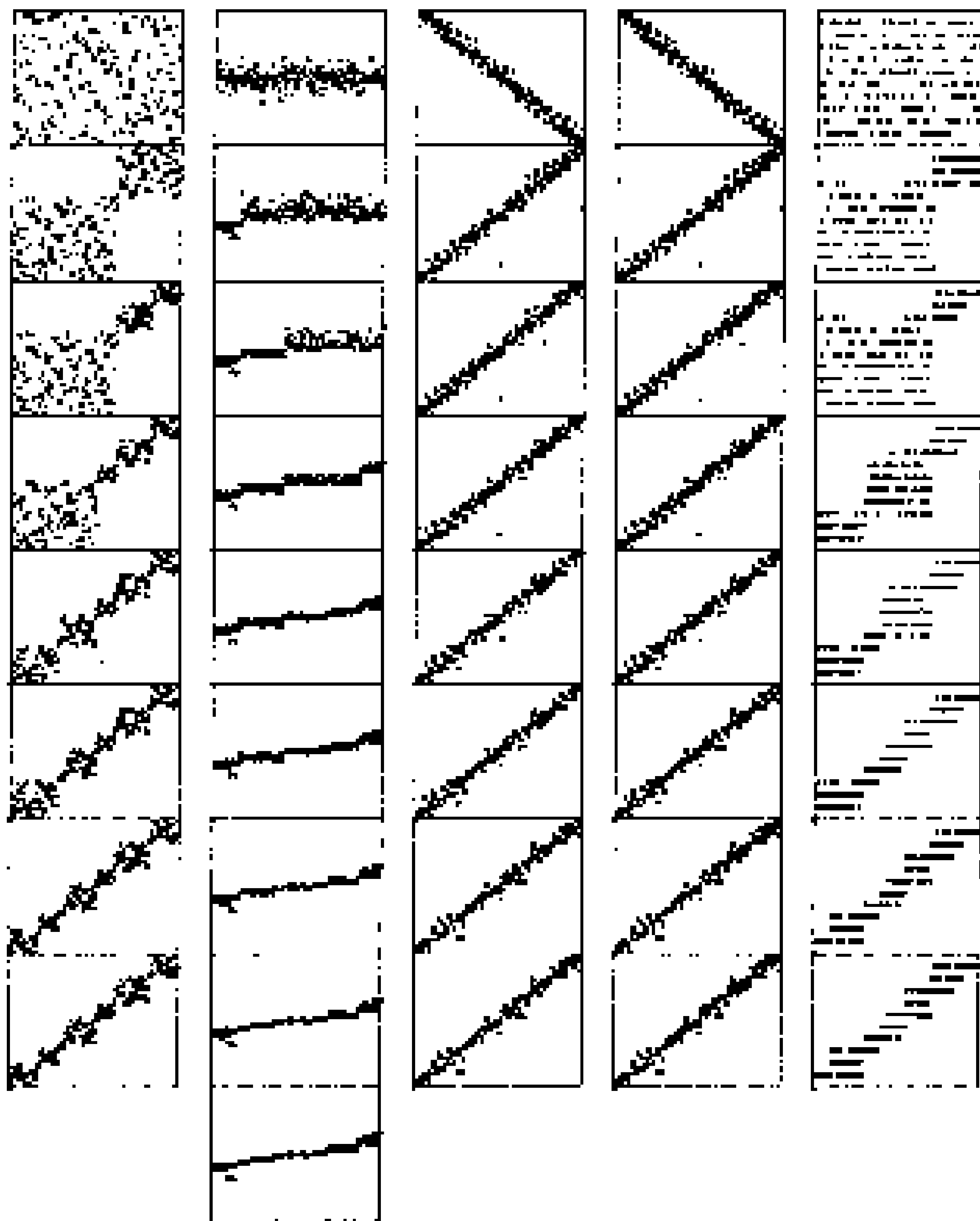


РИСУНОК 7.11. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ БЫСТРОЙ СОРТИРОВКИ С ВЫЧИСЛЕНИЕМ МЕДИАНЫ ИЗ ТРЕХ ЭЛЕМЕНТОВ НА ФАЙЛАХ РАЗЛИЧНЫХ ТИПОВ.

Модифицированная версия программы, предназначенная для выполнения работы на трех компьютерах (с процессорами 100, 150 и 200 МГц), позволила получить различные результаты и показать, что программа работает быстрее на массивах, близких к случайным. Однако самым медленным алгоритмом при сортировке массивов данных является алгоритм сортировки Хоара. Таким образом, можно сказать, что алгоритм сортировки Хоара является самым медленным алгоритмом.

Для быстрой сортировки мы хотим получить элемент, который разделит, как минимум, процедуру разделения. Связывая эту процедуру исключительно с тем, что нам не нужна совсем точная оценка (мы вообще можем обойтись без какой оценки, если столкнуться с невыполнимым выводом), мы просто хотим избежать невыполнимо большой оценки. Если попытаемся случайную выборку из одного элемента, получим неработоспособный алгоритм, который нигде не обладает высокой скоростью, не так далеко от простоты исходных данных. Если прочитаем случайную выборку из файла из трех или из пяти элементов, а затем воспользуемся механикой на этой выборке и процедура разделения, получим лучший результат, но также усложним сложившиеся данные с одной вычислительной задачей.

Программа 7.4. Улучшенная быстрая сортировка

Выбор медианы — из первого, среднего и последнего элементов в качестве разделения — у элементов, и сбалансирован выборки минимизирует количество элементов, которые приводят к ухудшению производительности. Эффективность быстрой сортировки. Данная реализация обеспечивает рядовые элементы (маленькие и большие) и элементы (маленькие и большие) (не следует использовать эти элементы в процессе разделения для других целей). Файлы для этой программы в формате C++ находятся в каталоге `src/quick_sort/` для платформы Linux. Для сортировки используется команда `insertion` из главы 6.

```
static const int N = 10;
template<class T>
void quicksort(T* a, int l, int r)
{
    if (l == r) return;
    int p = (l+r)/2;
    swap(a[l], a[p]);
    swap(a[l], a[r]);
    int i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
}

template<class T>
void hybrid_sort(T* a, int l, int r)
{
    quicksort(a, l, r); insertion(a, l, r);
}
```

Быстрая сортировка имеет широкое применение и связана с тем, что она успешно работает в различных ситуациях. Другие методы хорошо работают в некоторых специфических случаях, которые время от времени встречаются на практике. Но быстрая сортировка успешно решает гораздо большее число сортировочных задач, чем это можно сказать о применяемых других методах, а ее эффективность лучше всего проявляется там, где в сложных приложениях альтернативных подходов. В табл. 7.1 представлены эмпирические результаты, которые могут служить подтверждением некоторых из следующих выводов.

Таблица 7.1 Эмпирическое исследование алгоритмов быстрой сортировки

Быстрая сортировка (программа 7.1) работает примерно в два раза быстрее, чем сортировка методом Шелла (программа 6.3). Совершенствованный метод сочетания малых подфайлов и метода медленных трех элементов (программа 7.4) превосходит быстрый гибридный сортировочный алгоритм на 10 процентов быстрее.

7.6. Дублированные ключи

Файлы с большим числом дублированных сортируемых ключей довольно часто встречаются в различных приложениях. Например, может потребоваться сортировка большого файла с персональными данными по году рождения или, скажем, сортировка для элементов персонажа по полному признаку.

Ключи в сортируемом файле имеют множество дублированных значений, однако все еще определенностью утверждать, что рассмотренные нами различные реализации быстрой сортировки показывают высокие значения эффективности, ошибкой не можем существенно допустить. Например файл, который состоит исключительно из редких или другого ключей (если и то же значение), после не нуждается в дальнейшей сортировке, однако наши реализации продолжат работать, расширяя, завершая обработку все более мелкие подфайлы независимо от того, насколько быстро выполняется исходный файл (см. упражнение 7.8). В ситуации, когда во входном файле присутствуют большое число дублированных ключей, рекурсивная природа быстрой сортировки приводит к тому, что подфайлы, содержащие только элементы с одним и тем же ключом, будут иметь довольно много, благодаря чему существуют большие потенциальные возможности для совершенствования алгоритма сортировки.

Одна довольно простая идея заключается в делении файла на три части. Одна — для ключей, меньших рассматриваемого элемента, другая — для ключей, равных ему, и третья — для ключей, больших рассматриваемого элемента:



Выполнение такого разбиения довольно сложно, чем разбиение на две части, которое мы рассматривали ранее. Были предложены различные методы решения этой задачи. Классическим выражением по предположению, позволяющим избежать известность с легкой руки Дейкстры (Dijkstra), стала задача рассматривать начальный набор файлов, а прежде всего напомнить, что три рассмотренных категории ключей могут представлять три цвета флага (Dutch National Flag problem, см. раздел 8.6.1). В рамках быстрой сортировки мы добавляем еще одно ограничение, заключающееся в том, что эта задача должна быть решена за один проход по файлу — алгоритм, который предусматривает два прохода по данным, заменив быструю сортировку в два раза, даже если в исходном файле вообще не будет дублированных ключей.

Оригинальный метод, предложенный Нейсом (Nease) и Макдугалом (McDugal) в 1993 г. для разбиения файла на три части, представляет собой модификацию стандартной схемы разбиения и предусматривает следующее: ключи, равные рассматриваемому элементу и встречающиеся в всех подфайлах, накапливаются в левом конце файла, ключи, равные рассматриваемому элементу и встречающиеся в первом подфайле, накапливаются в правом конце файла. Во время выполнения процесса разбиения мы придерживаемся следующей схемы:

левый	меньше	равно	больше	правый
1	1	1	1	1
1	2	1	5	1

Далее, когда указатели пересекутся и точнее несомненно окажется равными, ключи станут идентичными. Мы переименовали эту попытку поэлементно с ключами, равными рассматриваемому элементу. Эта схема не совсем удовлетворяет требованиям, чтобы разделение файла на три части было завершено за один проход, поскольку непроизводительные расходы системы на ресурсы пропорциональны только количеству обнаруженных дублированных ключей. Этот факт присутствует для возможности потерь. Этот метод работает хорошо, даже если в исходном файле вообще нет дублированных ключей, поскольку в этом случае непроизводительные затраты отсутствуют. Во-вторых, для этого метода характерна линейная зависимость времени выполнения от длины файла под действием числа элементов ключей: каждый этап разделения не включает из процесса сортировки все ключи со значениями, равными рассматриваемому элементу, так что каждый ключ может быть исполнен максимум при постоянном числе разделений.

Рисунок 7.12 служит иллюстрацией работы алгоритма разделения на три части на примере учебного файла, а программа 7.5 описывает реализацию быстрой сортировки, в основу которой положен этот метод. Рассмотряваемая реализация требует дополнительного двухкратного прохода в цикл обмена и двух циклов for с тем, чтобы процедура разделения аппроксимировалась перемещением элементов, равных рассматриваемому элементу, в окончательные позиции. По-видимому, на это требуется меньше программного кода, чем в случае другого альтернативного разбиения файла на три части. И что более важно, этот метод не только исключительно эффективно решает проблему дублированных ключей, но и позволяет минимально возможный объем непроизводительных затрат в случае, когда в исходном файле вообще нет дублированных ключей.

Программа 7.5. Быстрая сортировка с разделением на три части

В основу программы положено разбиение массива на три части: l -я элемент, m -я и n -я разделяющие элементы (l — позиция $a[l]$, ..., n — позиция $a[n]$), m — элемент, разделяющий элемент (m — позиция $a[m+1]$, ..., $a[m-1]$), и элемент больше разделяющего элемента (l — позиция $a[l]$, ..., $a[l]$). Тогда этот сортировочный процесс будет завершаться вызовами.

Чтобы достичь поставленной цели, программа содержит ключи, разное разделяющее элементу, спав между l и q и между m и n . В разделение для ключа, вы-

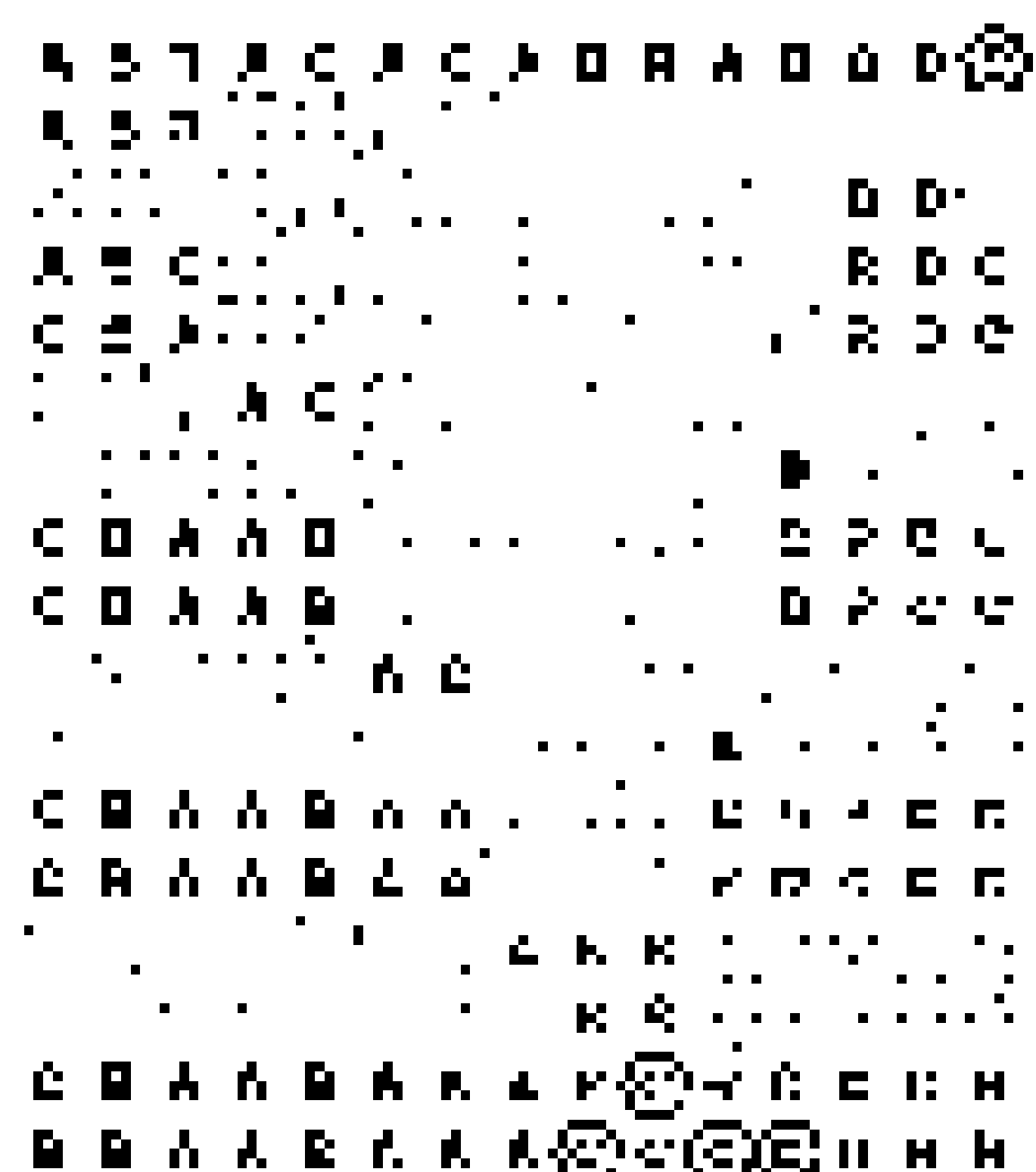


РИСУНОК 7.12. РАЗДЕЛЕНИЕ НА ТРИ ЧАСТИ

Эта иллюстрация показывает процесс разбиения массива на три части: l -я элемент, m -я и n -я разделяющие элементы (l — позиция $a[l]$, ..., n — позиция $a[n]$), m — элемент, разделяющий элемент (m — позиция $a[m+1]$, ..., $a[m-1]$), и элемент больше разделяющего элемента (l — позиция $a[l]$, ..., $a[l]$). Тогда этот сортировочный процесс будет завершаться вызовами. Чтобы достичь поставленной цели, программа содержит ключи, разное разделяющее элементу, спав между l и q и между m и n . В разделение для ключа, вы-

да указатели просто могут измениться и указываются объектами l и r . Она проведет каждый из этих элементов на предмет разности разделяющей способности. Если элемент, который сейчас находится слева, равен разделяющему элементу, то они пойдут через обмен и помещаются в левую часть массива; если элемент, который сейчас находится справа, равен разделяющему элементу, то в результате операции обмена он помещается в правую часть массива.

После того как указатели остановятся, элементы, равные разделяющему элементу x выходящему из рекурсии помещаются. После операции обмена они оказываются в своих окончательных позициях. После этого указатели могут быть исключены из подсчетов, для которых вычисляется последующее рекурсивное вызовы.

```
template <class T>
int operator==(const T& a, const T& b)
{ return (a==b) ? 1 : 0; }
template <class T>
void quicksort(T a[], int l, int r)
{ int k; T x = a[k];
  if (r <= l) return;
  int i = l-1, j = r, p = l-1, q = r;
  for (;;)
  {
    while (a[++i] < x) ;
    while (x < a[j]) { if (j == l) break;
    if (i >= j) break;
    swap(a[i], a[j]);
    if (a[i] == x) { p++; swap(a[p], a[i]); }
    if (x == a[j]) { q--; swap(a[j], a[j]); }
  }
  swap(a[i], a[j]); i = i-1; j = j+1;
  for (k = l; k <= p; k++, j--) swap(a[k], a[j]);
  for (k = r; k >= q; k--, i++) swap(a[k], a[i]);
  quicksort(a, l, i);
  quicksort(a, j, r);
}
```

Упражнения

- 7.34. Дайте объяснение тому, что произошло, когда программа 7.3 была исполнена на файле с произвольной структурой (а) с двумя различными значениями ключей и (б) с тремя различными значениями ключей.

7.35. Изменить программу 7.1 таким образом, чтобы она выполняла команду `rsplit`, если все ключи в файле различны. Сравните эффективность полученной программы с эффективностью программы 7.1 на больших файлах с произвольной структурой с ключами, приближенными к различным значениям при $i = 2, 5$ и 10 .

7.36. Предположим, в программе 7.1 вместо того, чтобы установить границу с левой необходимой ключей, равной разделяющему элементу, при обнаружении такого ключа он пропускается. Показать, что в таком случае время выполнения программы 7.1 подчиняется квадратичной зависимости.

- 7.37. Доказать, что время выполнения программы из упражнения 7.37 пропорционально квадрату длины файла для всех файлов с n (n) различными значениями ключей.

7.30. Написать программу, определяющую количество различных классов, которые могут возникнуть в файле. Постыскать файл, получившийся при выполнении программы для подсчета числа различных классов в файле с произвольной организацией, состоящего из N целых чисел, принадлежащих диапазону в диапазоне от 0 до $M-1$ для $M = 10, 100$ и 1000 и для $N = 10^1, 10^2, 10^3$ и 10^4 .

7.7 Строки и векторы

Когда свариваются катоды с галат спринк, мы можем пользоваться разными видами электродов: дугами, дуговыми электродами (с. 1), совместить с реакциями и электродом электродов, рассмотренными в электродной галате. Намочить на то что такой электрод позволяет получить коррозийную и эффективную реакцию (абсолютную) при работе в крутилке (дугах) большим электродом (чем удельная мощность 50-60 Вт/м²), можно даже скрыть электроды, которые являются того, чтобы на них оставались электроды.

Проблемы такто-дефекта в старшей группе *Шефэр*, который срывался две строчки и направлялся сразу направо, констатируя стрелки замков за секунду. Встретив на это время, в перерывах между количеством сыновей, сориентирован в обоих стрелках. На заключительных этапах процесса разделения. Встрой сориентирован, когда каждый бегает друг к другу на значительную, почти на сто процентов, стартовую сориентированность. Все завершающие стрелки, так что все завершающие стрелки завершаются, чтобы эти стрелки были не обнаружены.

Например, разговорный пошлый «замерок» — англоязычный сленг — *discreet*, *discredit*, *discrete*, *discrepancy* и *discretion*. Все сравнения, выполняемые с помощью строгих или этих словечек, рассудят по меньшей мере семь человек. Но в разговорном сленге можно было бы начать процесс с результата «ничего», а не былой логичной логикой — англоязычной, фиксирующей тот факт, что перестать иметь значение — значит

Процедура обработки файла на три этапа, которая рассматривалась в разделе 7.5, представлял собой элементарный способ выдать карту на отмеченного выте факта. На каждом этапе процесс разделения пропущенных символов (символ (символ, символ в позиции i)). Предполагалось, что символ, заключенный в скобки от 0 до $i-1$ и подлежащий сортировке, спускают. Мы пытаемся разделить на три этапа, помещая в i -й символ, $i-1$ символ, который не является i -м символом разделяющего элемента, с тем, чтобы $i-1$ символ, который равен i -му символу разделяющего элемента, в середине, а те символы, i и символ, который больше i по символу разделяющего элемента, спуска. Далее мы имеем два обычных действия: i и $i-1$ символы, что мы сортируем с помощью профиля, начиная с $i-1$ и i символа. Петрушик знает, что этот метод обеспечивает корректную сортировку и к тому же обладает высокой скоростью работы (табл. 7.2). В данном случае мы рассматриваем убедительный пример неограниченной полноты рекурсивного выполнения (с программой-примеч.).

Для реализации этого вида сортировки требуется три явных и более высоких уровня абстракции, которые мы обеспечим доступ к абстрактным сущностям кэша. Возможность не являясь сортировкой в рамках, имеет область имен C++ для реализации реализации декларативной простой. Тем не менее, мы отложим обсуждение детальной реализации до главы 10, в которой рассмотрим различные методы

СОРТИМЕНТА, НАПРАВЛЕНИХ НА ОБСЛУЖУВАННЯ, ЧТО ДОПРИНЕСУЮТ КИТАЙСКИМ ЛОДКАМ РАБОТАЮЩИМ НА ШИПОВИ МАЛЕНКОМ ЧИСЛОМ.

Таблица 7.2. Эмпирическое исследование вариантов быстрой сортировки

[illegible]

n	V	I	M	Q	X	T
2500	2	7	5	10	7	5
25000	18	14	10	20	17	12
50000	27	21	17	35	27	20
100000	51	35	35	103	53	39

בְּחִיבָה

- [illegible]

Этот подход распространяется на многомерные сортировки, в условиях которых в качестве ключей выступают пары, а иногда даже тройки переупорядоченным таким образом, что значения файла упорядочиваются по первому компоненту, затем значения по второму перпендикулярно ключам упорядочиваются по второй компоненте и так далее. Если компоненты не имеют дублированных значений, то проблема сводится к сортировке по первому компоненту; однако в общем случае каждая из компонент может принимать одно из нескольких различных значений, и тогда сортировку переводят к разбиению на три части (переход к следующему компоненту и раздел на раздели). Этот случай Хорр рассматривает в своей первой статье, он представляет собой типичное практическое применение.

переупорядочивает массивы $a[1] \dots a[r]$ и возвращает индекс i такой, что элементы от $a[1]$ до $a[i-1]$ меньше или равны $a[i]$, а элементы от $a[i+1]$ до $a[r]$ больше или равны $a[i]$. Если k равно 1, то такая задача тривиальна. С другой стороны, если $k < 1$, можно предложить работу на левой подфайле; если $k > 1$, работу необходимо предложить на правом подфайле. Подобный подход приводит к рекурсивной программе выборки, которую является программа 7.6. Пример работы этой процедуры на файле `количество_различных_элементов` на рис. 7.13.

Программа 7.6. Выборка

Данная процедура реализует разбиение массива относительно k -го наименьшего элемента (элементы $a[k]$): она возвращает индекс такого элемента, что $a[1] \dots a[k-1]$ меньше или равны $a[k]$, а $a[k+1] \dots a[r]$ больше или равны $a[k]$.

Например, если a состоит из `select(a, 0, N-1, N/2)` с любым разбиением массива по значению `select` и оставшаяся часть `a[N/2]`.

```
template <class T>
void select(T* a, int l, int r,
           int k)
{
    if (r <= l) return;
    int i = partition(a, l, r);
    if (i > k) select(a, l, i-1, k);
    if (i < k) select(a, i+1, r, k);
}
```

Программа 7.7. Нерекурсивная выборка

Нерекурсивная реализация выборки работает иначе. Затем помещает левый указатель в начало, если этот индекс окажется слева от искомого элемента, или наоборот правый указатель в раздел, если раздел окажется справа от искомого элемента.

```
template <class T>
void select(T* a, int l, int r, int k)
{
    while (r > l)
    {
        int i = partition(a, l, r);
        if (i > k) r = i-1;
        if (i < k) l = i+1;
    }
}
```

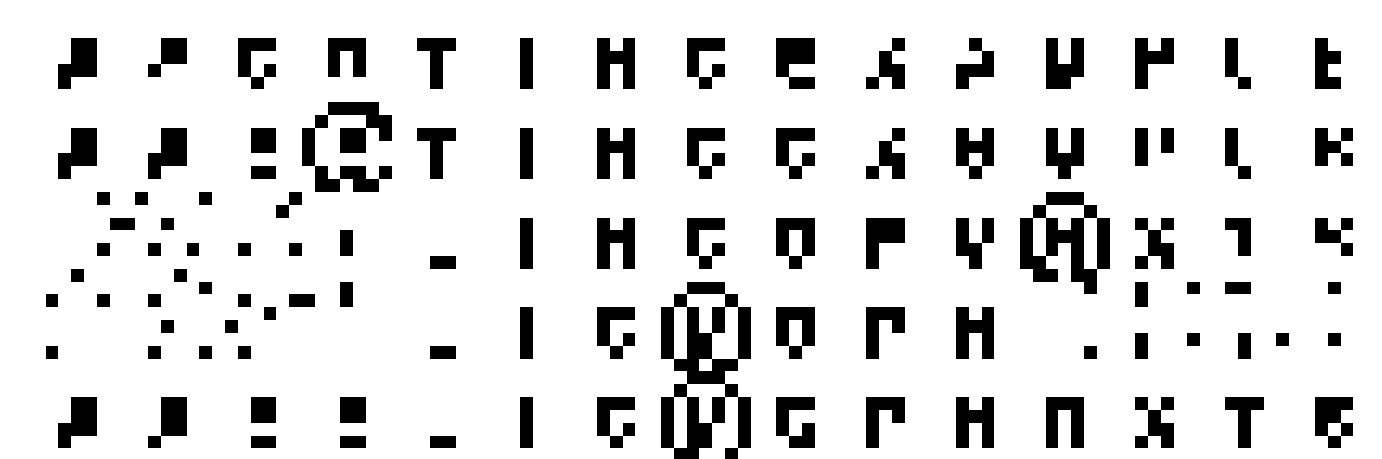


РИСУНОК 7.13. ВЫБОРКА МЕДИАНЫ

Эта процедура выбирает элемент a в качестве элемента-разделителя, помещая его в начало массива, и выбирает элемент $a[k]$ в качестве элемента-разделителя. Если k равно 1, то такая задача тривиальна. С другой стороны, если $k < 1$, можно предложить работу на левой подфайле; если $k > 1$, работу необходимо предложить на правом подфайле. Подобный подход приводит к рекурсивной программе выборки, которую является программа 7.6. Пример работы этой процедуры на файле `количество_различных_элементов` на рис. 7.13.

Программа 7.7 представляет собой переупорядочивание версии, построенную на базе рекурсивной версии, представленной в программе 7.6. Поскольку эта процедура всегда завершается рекурсивным вызовом самой себя, мы просто устанавливаем базовые значения параметров и возвращаемся в начале программы. Это значит, что мы отклоняемся от рекурсии и не используем стек, а также исключаем из программы вызовов, в которых используются k , и рассуждения k только для индексов массива.

Лемма 7.4. Время выполнения выбора, для которой используется быстрая сортировка, и связанное количество памяти линейно зависят от N .

Как и в случае быстрой сортировки, можно предположить в первом приближении, что в случае файла очень больших размеров каждая операция разделения требует выполнения $N = N/2 + N/4 + N/8 + \dots + 2N$ операций деления. Но поскольку разделение выполняется быстро для быстрой сортировки, это грубое приближение является приемлемым. Аналогично, подобный приближению в разделе 7.3 анализ быстрой сортировки, но с более сложными (см. раздел 5.4.4) дает результаты, которые можно сравнить с более точными (см. раздел 5.4.4) выражением

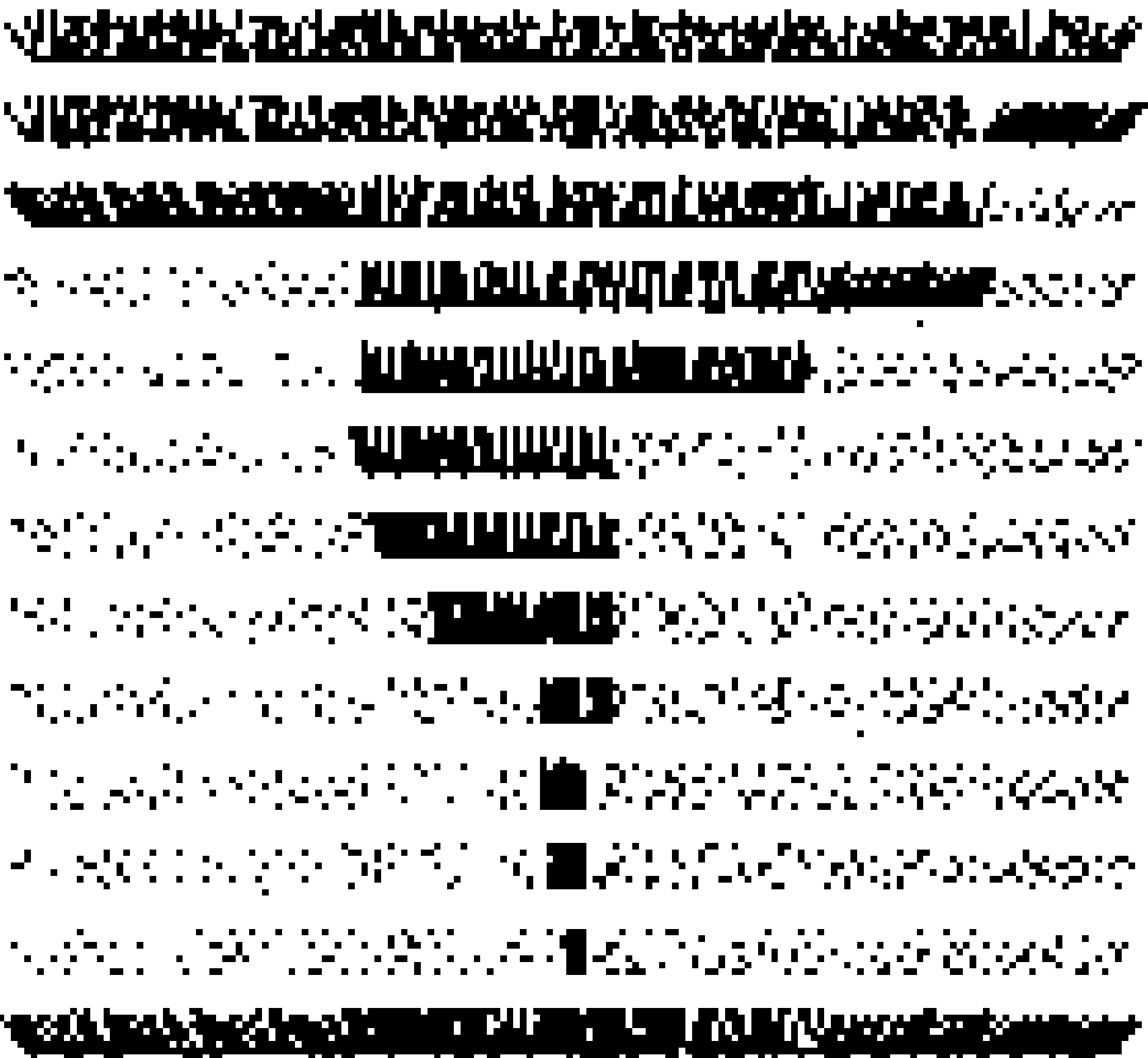
$$2N + 2k \ln(N/2k + 1) \approx 2(N + k) \ln(N/(N + k)),$$

которое линейно зависит от любой достижимой величины k . Вычисление этих формул для $N = N/2$ показывает, что для больших размеров массива необходимо выполнить $(2 + 2 \ln 2) N$ операций.

Пример того, как этот метод находит медиану в крупном файле, представлен на рис. 7.14. В рассматриваемом случае вместо того, чтобы выбрать элемент, который должен быть уменьшен в размере в два раза, мы уменьшаем его в четыре раза, таким образом, эта процедура завершается за $O(\log k + 1)$ шагов. Это значение программы можно ускорить, выводя в нее выборы, однако при этом следует соблюдать осторожность (см. приложение 7.4.9).

РИСУНОК 7.14. ВЫБОР МЕДИАНЫ С ПОМОЩЬЮ ПРОЦЕДУРЫ РАЗДЕЛЕНИЯ.

Процедура выбора и деления массивов разбивает на равные половины. В первом случае мы делим массив на четыре части, поскольку мы знаем, что медиана находится в одной из них. В следующем шаге мы делим каждую из этих частей на две части, и так далее, пока не останется одна часть.



Неудачный случай происходит так же, как и для случая быстрой сортировки — использование данного метода для нахождения наименьшего элемента уже упорядоченного файла приведет к тому, что время выполнения этой процедуры увеличивается квадратичной зависимостью. Можно модифицировать эту процедуру, исключив из элементов быстрой сортировки, таким образом, что время ее выполнения будет квадратично увеличиваться линейной зависимостью. Однако все модифицированные процедуры будут очень важными в практическом отношении, следовательно, не приемлемы на практике.

Упражнения

- 7.41. Сколько в среднем требуется операций сравнения, чтобы найти наименьший элемент в файле из N элементов, применяя для этой цели процедуру `select`?
- 7.42. Сколько в среднем требуется операций сравнения, чтобы найти $1/N$ и наименьший элемент, применяя для этой цели процедуру `select`, для $\alpha = 0.1, 0.2, \dots, 0.9$?
- 7.43. Сколько требуется операций сравнения в худшем случае, чтобы найти k -й элемент из N элементов, применяя для этой цели процедуру `select`?
- 7.44. Напишите эффективную процедуру перестроения файла таким образом, чтобы все элементы с ключами, равными k или $k+1$, оказались в одной последовательности элементов негнущихся списка и элементы с ключом $k+2$ — в другой.
- 7.45. Проделайте последовательные поды использования выбора с целью повышения эффективности выбора. Составьте таблицу, дающую время на выполнение каждого из указанных результатов.
- 7.46. Различайте два типа выбора на основе метода трехэлементного разделения на примере заданного файла с ключами, представляющими различные значения, для $k = 2, 3, 4, \dots, 6$.

Слияние и сортировка слиянием

Самое слабое звено алгоритмов быстрой сортировки, рассмотренное в главе 7, состоит из операции выбора подпоследовательности k -го минимального элемента в файле. Мы убедились в том, что выполнение операции выбора можно эффективно разделить файл на две части, из которых первая содержит k минимальных элементов, а вторая — остальные. В этой главе рассмотрим алгоритмы сортировки, в основе которых лежит вспомогательный процесс, известный как *слияние*, т.е., объединение двух отсортированных файлов в один файл большего размера. Слияние является основой для простого алгоритма сортировки типа «разделяй и властвуй» (см. раздел 3.7), а также для его доработки — алгоритма восходящей (снизу-вверх) сортировки слиянием. При этом оба из них достаточно просто реализуются.

Выборка и слияние — суть вспомогательные операции в том смысле, что выборка разделяет файл на два независимых файла, а то время как слияние объединяет два независимых файла в один. Контраст между этими двумя операциями становится очевидным, если применить принцип «разделяй и властвуй» для создания конкретной методики сортировки. Можно допустить, что изначально файл таким образом, что когда обе части файла подвергнутся сортировке, получится один файл, и наоборот, можно разбить файл на две части для последующей сортировки, а затем объединить полученные части, чтобы получить весь файл в упорядоченном виде. Мы уже видели, что получается в первом случае: это не что иное как быстрая сортировка, которая состоит из процедуры выбора, за которой следует два рекурсивных вызова.

В этом главе рассматривается *быстрая сортировка* (quick sort), которая является достаточно быстрой сортировкой в том, что она состоит из двух рекурсивных вызовов и последующей процедуры свертывания.

Одним из наиболее привлекательных свойств сортировки слиянием является тот факт, что она сортирует файл, состоящий из N элементов, за время, пропорциональное $N \log N$, независимо от характера входных данных. В главе 5 мы рассмотрим еще один алгоритм, время выполнения которого пропорционально $N \log N$; алгоритм имеет название *пошаговой сортировки* (heap sort). Основным недостатком сортировки слиянием является то, что практическое использование этого алгоритма требует достаточно большого пространства памяти, пропорционального N . Это препятствие можно обойти, однако сделать это довольно сложно, причем потребуются дополнительные затраты, которые в общем случае не оправдываются на практике. Особенно если учесть, что существует альтернатива и более совершенная сортировка. Получить программную реализацию сортировки слиянием не труднее, чем реализовать быструю сортировку, при этом время внутреннего цикла приближается среднему значению между аналогичными показателями быстрой сортировки и пирамидальной сортировки, следовательно, сортировка слиянием способна достичь того, чтобы к ней проявить внимание в случае, когда на первый план выходят такие показатели как быстрота действия, надежность и гибкость. В некоторых случаях, возможность использования дополнительного пространства памяти.

Но гарантированное время выполнения, пропорциональное $N \log N$, может оказаться недостатком. Например, в главе 6 мы увидим, что существуют методы, которые могут быть адаптированы таким образом, что в некоторых особых ситуациях, например, когда уровень упорядоченности файла достаточно высок, тогда время выполнения будет лишь несколько раз больше, время на выполнение линейно. В противном случае, время выполнения сортировки слиянием имеет типичный образ от числа элементов входного файла и оно пропорционально не обязательно к $N \log N$.

Сортировка слиянием — это устойчивая сортировка, в данных обстоятельствах склоняет чашу весов в ее пользу в тех приложениях, в которых устойчивость имеет важное значение. Конкурирующие методы, такие как быстрая сортировка или пирамидальная сортировка, не относятся к числу устойчивых. Различные приемы, представленные этими методами устойчивости, имеют стойкую тенденцию к росту времени выполнения дополнительного пространства памяти; следовательно, требования дополнительного памяти, предъявляемые со стороны сортировки слиянием отталкиваются на задний план в тех случаях, когда устойчивость становится доминирующим фактором.

Другое свойство сортировки слиянием, которое приобретает важное значение в некоторых ситуациях, является тот факт, что сортировка слиянием обходит реальную сложность образа, что она существует, а понимая, что ответственный доступ к данным (каждый элемент за другим). Например, сортировка слиянием — именно тот метод, который можно применить к спискам спискам, для которых из всех методов доступа применим только метод последовательного доступа. По тому же принципу, как мы увидим в главе 7, слияние часто используется в качестве метода для сортировки на вычислительных машинах и высокопроизводительных машинах, поскольку именно последовательный доступ к данным в глобальной системе обработки данных является самым быстрым.

8.1. Двухпутевое слияние

Иногда для упорядочивания табличных файлов, их можно объединить в один упорядоченный выходной файл просто открывавая записывающий элемент в каждом файле и проходя по циклу, в котором выбираются из двух элементов, наименьших из списков файлов, перемещаются в выходной файл: процесс продолжается до тех пор, пока оба входных файла не будут исчерпаны. Мы сталкиваемся с несколькими репликациями этой основной абстрактной операции с этим и следующим разделом. Время выполнения линейно по элемент от количества элементов в таблицах. Если из каждой операции поиска записывающего элемента в файле уходит одно и то же время, то как рано и нулевое место в том случае, когда отсортированные файлы представляются структурами данных, которая поддерживает последовательный доступ с постоянным временем доступа, такой как связный список или массив. Эта процедура представляется собой обобщенную версию [104-105] приведен в главе 11 мы подробнее остановимся с многопутевым слиянием, в котором принимается участие более двух файлов. Наиболее важным применением многопутевого слияния является сортировка, которая подробно рассматривается в текущем разделе.

Для начала предположим, что даны два непересекающихся упорядоченных массива целых чисел $a[0], \dots, a[N-1]$ и $b[0], \dots, b[M-1]$, которые требуется сложить в третий массив $c[0], \dots, c[N+M-1]$. Если рассуждения описанные раньше являются в том, чтобы последовательно выбрать для c наименьший из оставшихся элементов из a и b , как показано в программе 8.1. Это рекуррентно отличается от того, что же программа способна обладать характеристиками, которые мы сейчас и рассмотрим.

Программа 8.1. Слияние

Чтобы объединить два упорядоченных массива a и b в упорядоченный массив c , используется цикл `for`, который помещает элемент в массив c на каждой итерации. Если a исчерпан, элемент берется из b ; если не исчерпан b то элемент берется из a ; если же элементы исчерпаны и в том и в другом массиве, наименьший из оставшихся элементов a и b переходит в c . Таким образом, последовательная обход упорядоченных списков массивов a и b заканчивается прежде чем закончатся c и a и b исчерпаны (т.е., из последовательности для массивов a и b используется память).

```

#include <stdio.h>
void merge2(int a[], int m, int k,
            int b[], int K)
{
    for (int i = 0, j = 0, k = 0; k < m+K; k++)
    {
        if (j == m) { c[k] = b[j++]; continue; }
        if (i == K) { c[k] = a[i++]; continue; }
        c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
    }
}

```

Во-первых, эта реализация проста. Во-вторых, не переставляет. В-третьих, если a и b являются крупными массивами, то для размещения выходных данных необходима большая память (обычно — временная). Выходит того, чтобы не переставлять, использовать пространство памяти, пропорциональное размерам входного файла, явля-

данных массивов — массив отсортированных такой же длины, который необходимо отсортировать. Обсуждаемые методы, например, объединить два упорядоченных файла $a[1] \dots a[m]$ и $b[m+1] \dots b[n]$ в один упорядоченный (если бы был эффективный способ перемещения элементов $a[1] \dots a[m]$ без использования больших объемов дополнительной памяти), представляли бы собой задачу на каком-то уровне абстракции, и думать о том, как можно ее это сделать. Первый шаг кажется, что у этой проблемы есть простое решение, однако на самом деле все оказывается довольно сложным. Обязательно стоит сравнить с программой 8.1. И в этом деле, конечно, трудно разработать алгоритм оптимального хранения, т.е. в рамках пространства памяти, занимаемого входными файлами, который обладает былыми характеристиками и который мог бы стать альтернативой обменной сортировке. Мы перейдем к этой проблеме в разделе 8.2.

Сначала, как операция, имеет свою собственную область применения. Например, в типичной среде обработки данных может возникнуть необходимость прочитать крупный (упорядоченный) файл данных, в который непрерывно поступают новые элементы. Одним из способов заключается в том, что новые элементы группируются в пакеты, которые затем добавляются к концу (или началу) файла, пока не будет выполнено требование сортировки пакета файла. Такая ситуация как бы специально создана для применения более эффективной стратегии предусматривает сортировку пакета (увеличивших размеров) новых элементов, с помощью которых получено по файлу некоторый размер с другим данным файлом. Сначала не предполагается никаких подобных реляционных, что обуславливает сложность с вычисления. Основное понимание и понимание также будет усиливаться методом сортировки, в первую очередь, полагаясь на этикетку.

Упражнения

8.1. Предположим, что упорядоченный файл размером N нужно объединить с некоторым другим файлом размером M , при этом M намного меньше N . Во сколько раз быстрее, чем повторная сортировка, работает предложенный метод, в котором требуется сделать слияние, если его рассмотреть как функцию от M , при $N = 10^5$, 10^6 и 10^7 ? Разные эти задачи поставлены, но в каждом рассмотрении имеется программа сортировки, которой требуется $c_1 N \lg N$ секунд, чтобы выполнить сортировку файла размером N , и программа слияния, которой требуется $c_2 (M + N)$ секунд, чтобы слить файл размером N с файлом размером M , при $c_1 = 2$.

8.2. Чем превосходит и в чем уступает сортировка этого файла методом быстрого сортировки, предложенным в упражнении 8.1? Ответьте на этот вопрос, полагая, что входной файл имеет некоторую организацию, так что каждая запись требует примерно полтора и больше файлов, а время выполнения сортировки определяется выражением $c_1 N \lg^2 N$, при этом константа c_1 — того же порядка, что и другие константы.

8.3. Что происходит, если воспользоваться программой 8.1 для обмена слиянием посредством слова merge(x , a , $N/2$, $x + N/2$, $N - N/2$) применительно к массиву $A = Q S U V R E N D S T$?

- 8.4. Верно ли утверждение, что программа 8.1. выполняется так же, как показано в выражении 8.2, работает правильно тогда и только тогда, когда её входные данные уже отсортированы? Докажите правильность вывода или представьте контрпример.

8.2. Абстрактное обменное слияние

Хотя реализация слияния, по задумке, требует дополнительного пространства, мы все еще считаем абстрактное обменное слияние полезной при реализации методов сортировки, которые здесь изучаются. В нашей следующей реализации слияния мы сделаем акцент на этом моменте и как бы искусственно искусственной переделаем (а, б, в, г). Там самым показывая, что программа имеет результат слияния $a[1], \dots, a[m]$ и $b[m+1], \dots, b[n]$ в объединенный упорядоченный файл $a[1], \dots, a[n]$. Можно было бы реализовать эту программу, используя свопинг элементов, но все же вспомогательный файл с соответствующим применением быстрого метода, использованного в программе 8.1, однако пока мы не будем делать этого, а скомпьютеризуем одно усовершенствование в данный подход. Мы хотим дополнительно продемонстрировать память для вспомогательного массива, по умолчанию, по практике слияния с определенной целью, в разделе 5.4 мы рассмотрели различные улучшения, которые позволяют избежать дополнительных затрат времени, требуемых на конкретные массивы.

Вторым, исключительным элементом, основным характеристика базового слияния заключается в том, что внутренний цикл содержит две проверки с целью определить, достигнут ли конец хотя бы одного из двух массивов. Чаще всего эти проверки не выполняются, и создаются исключительно благоприятные ситуации для копирования служебных ключей, позволяющих избежать такого рода проверки. Иначе говоря, если элемент из служебных ключей, больше, чем значения всех других ключей, добавленных в конец массива a и массива b , от этих проверок можно отказаться в силу того, что если массив a (или b) будет пустым, служебный элемент перейдет в режим выбора следующих элементов только из массива b (или a) и заменить их в массиве a вплоть до окончания операции слияния.

Однако, как было показано в главах 6 и 7, служебными ключами не всегда просто пользоваться либо в силу того, что не всегда легко находить элементы наибольшего элемента, либо в силу с тем, что необходимые пространство памяти не так то просто получить. Что касается слияния, то существует достаточно простое средство, которое показано на рис. 8.1. В основе этого метода положены следующие идеи: при условии, что мы отказались контролировать массивы, чтобы реализовать обменное слияние, мы просто представляем старый файл по время его копирования и обмена нормальными (без дополнительных затрат), так это означает с ним указывать соответствующий строк в тексте. Эта операция приводит к тому, что полученный элемент, в каком бы он файл не находился, всегда служебным ключом для другого массива. Программа 8.2 содержит эффективную реализацию абстрактного обменного слияния, в которой мы можем увидеть, как она работает. Она также использует вспомогательный массив, размер которого пропорционален количеству файлов, полученному в резуль-

тате слияния, но эти затраты будут аффективными, поскольку произвольная разбивка. Поскольку в этом случае падает необходимость проверок в целях обнаружения окончания слияемых массивов.

Программа 8.2. Абстрактное обменное слияние

Данная программа выпадает слияние двух файлов без использования служебных меток. Пусть массивная структура массива в массивах в обратном порядке, когда сразу за каждым парным массивом следует один, второго (т.е., устоявшийся для битовой структуры). Первый цикл для перемешивания парный массива, после чего указывается на то, что слияние готовность начинать слияние. Второй цикл для перемешивания второго массива, после чего указывается на то, что слияние готовность начинать слияние. Третьим элементом служит служебной меткой — слово «for» — чтобы указать на то, что слияние готовность начинать слияние. Внутренний цикл этой программы достаточна простейший (перенос в массив, слияние, слияние) слияние и увеличение элементов i или j на единицу, увеличение на единицу k процесса слияния k .

```
template <class T>
void merge(T& a, int l, int m, int r)
{
    int i, j;
    while (l < r)
    {
        int k = l;
        while (k < m)
        {
            int i = l;
            int j = m;
            while (i < j)
            {
                if (a[i] < a[j])
                    a[k++] = a[i++];
                else
                    a[k++] = a[j++];
            }
            a[k] = a[m];
            l = m;
            m = j;
        }
    }
}
```

Последовательность ключей, которая сначала увеличивается, а затем уменьшается (или сначала уменьшается, а затем увеличивается), называется *бессмысленной* (или *нестабильной*). Формировка бессмысленной последовательности эквивалентна слиянию, но иногда удобно представить проблему слияния как проблему битовой сортировки; рассмотренный метод, позволяющий избежать слияния со служебными элементами, можно рассматривать как простой пример такой сортировки.

Одно из важных свойств программы 8.1 заключается в том, что результирующее слияние устойчиво: она сохраняет относительный порядок элементов с одинаковыми ключами. Это свойство можно использовать, и часто имеет смысл учитывать в том, что устойчивость сохраняется при реализации абстрактного обменного слияния. Поскольку устойчивое слияние немедленно приводит к устойчивым методам сортировки, как было показано в разделе 8.3. Не всегда просто сохранить свойство устойчивости: например, программа 8.1 не обеспечивает устойчивости (см. упражнение 8.6). Это обязательство еще больше усложняет проблему разработки полного алгоритма обменного слияния.

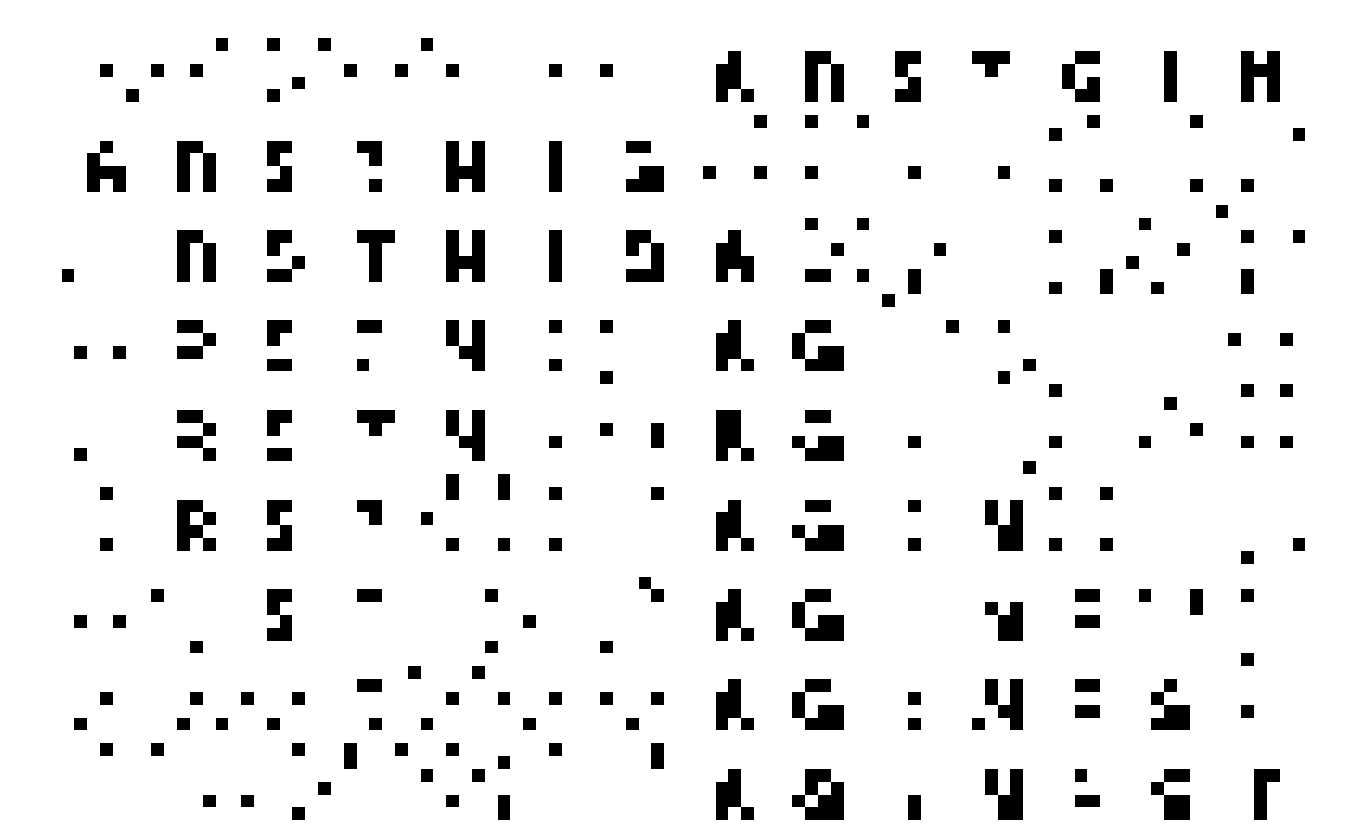


РИСУНОК 8.1. СЛИЯНИЕ БЕЗ ИСПОЛЬЗОВАНИЯ СЛУЖЕБНЫХ МЕТОК.

Чтобы понять, как это работает, рассмотрим пример. Пусть у нас есть два массива: $A = [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]$ и $B = [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]$. Мы хотим объединить их в один массив C . Для этого мы будем использовать метод, который мы описали в программе 8.1. Мы будем использовать два указателя i и j , которые будут указывать на текущие элементы в массивах A и B . Мы будем сравнивать элементы $A[i]$ и $B[j]$, и если $A[i] < B[j]$, мы будем перемещать элемент $A[i]$ в массив C . Если $B[j] < A[i]$, мы будем перемещать элемент $B[j]$ в массив C . Мы будем продолжать это, пока не закончатся элементы в массивах A и B . В конце концов, мы будем иметь массив C , который содержит все элементы из A и B в отсортированном порядке.

Упражнения

- 8.5. В стиле примера, представленного на диаграмме 8.1, показать, как выполнялся слияние списков A B C D E F G H с использованием программы 8.1.
- 8.6. Объяснить, почему программа 8.2 не является устойчивой, и разработать устойчивую версию этой программы.
- 8.7. Что получится, если применить к спискам E A B C D E F G H программу 8.2?
- 8.8. Верно ли, что программа 8.2 вызывает правильный вызов merge и mergeSort, когда оба входных списка являются списками, устойчиво сортированными? Приведите аргументы в пользу этого ответа либо представьте контрпример.

8.3. Нисходящая сортировка слиянием

Несомненно, рассмотренная процедура сложна, поэтому приходится прибегать к качеству остатка для рекурсивной процедуры сортировки. Чтобы отсортировать заданный файл, мы делим его на две части, выполняем рекурсивную сортировку обеих частей, после чего производим их слияние. Описание этого алгоритма представлено на странице 8.1; пример иллюстрируется на рис. 8.7. Как отмечалось в главе 5, этот алгоритм является одним из широко известных примеров использования принципа "разделяй и властвуй" при разработке эффективных алгоритмов.

Нисходящая сортировка слиянием аналогична принципу управления списком, в рамках которого руководитель организует работу таким образом, что получает большую задачу, но разбивает ее на подзадачи, которые должны немедленно решаться его подчиненными. Если каждый руководитель будет решать свою задачу, разбивая ее на две равные части с помощью об-обсужденных решений, получивших свое отражение в последующей передаче результатов своему начальству, то примерно также организованная сортировка слиянием. Разделяя задачу на две части, пока кто-то из нас не сможет полностью исполнить, не получит и не выполнит свою задачу (в рассматриваемом случае это слияние двух файлов размером 1); каждая руководитель выполняет значительную часть работы, выполняя результаты работы подчиненных и ежечасное дело.

Сортировка слиянием строит каждую часть списка, начиная с элементов заданного в нас ма-

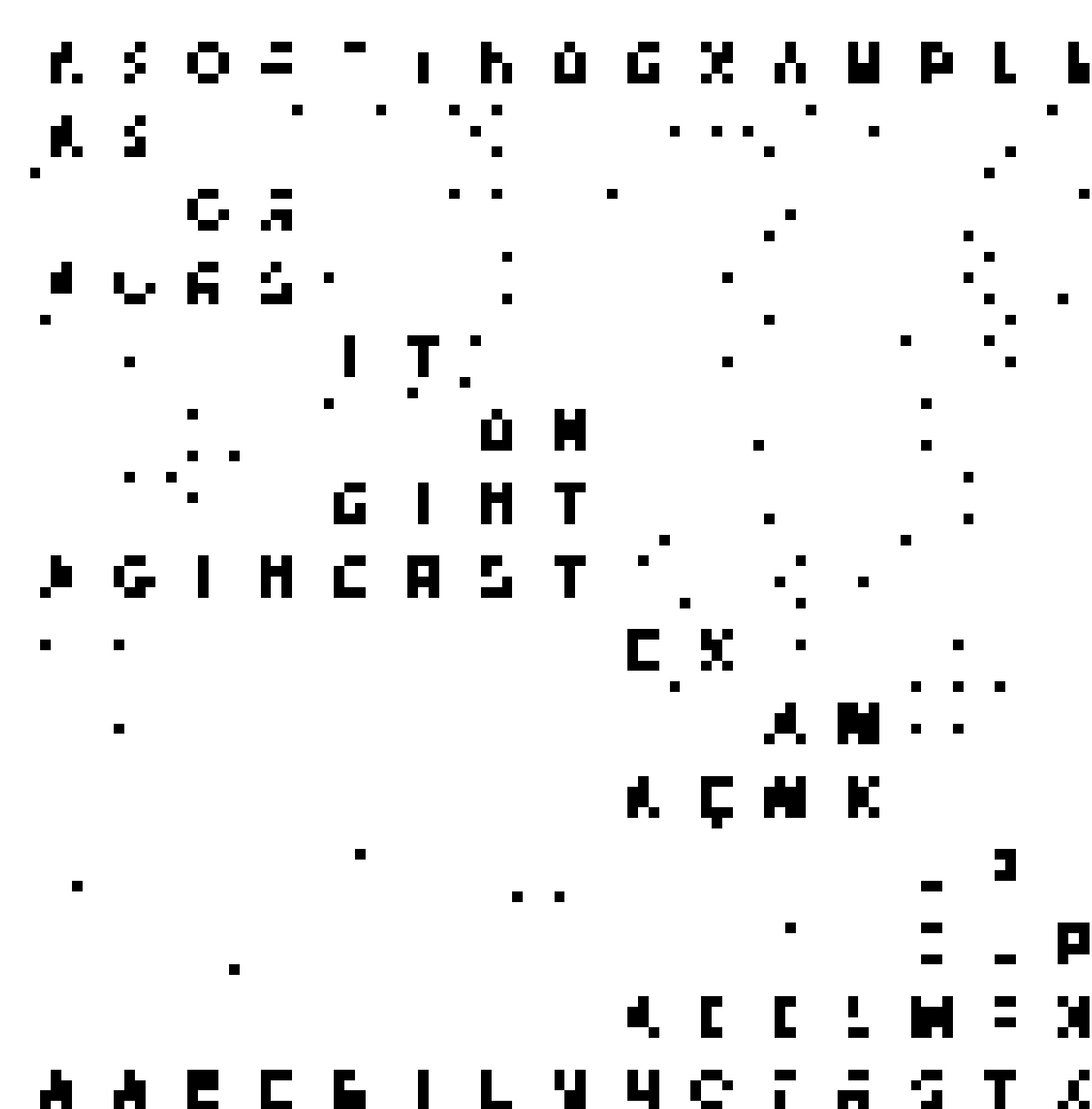


РИСУНОК 8.2. ПРИМЕР
НИСХОДЯЩЕЙ СОРТИРОВКИ
СЛИЯНИЕМ

Алгоритм сортировки слиянием реализуется с помощью функции mergeSort, процесс которого описывается следующим образом. Сначала выполняются шаги A и B, которые описаны в рис. 8.7, затем шаг C, который описан в рис. 8.8, и шаг D, который описан в рис. 8.9. Шаг A описан в рис. 8.10, шаг B описан в рис. 8.11, шаг C описан в рис. 8.12, шаг D описан в рис. 8.13. Шаг E описан в рис. 8.14, шаг F описан в рис. 8.15, шаг G описан в рис. 8.16, шаг H описан в рис. 8.17, шаг I описан в рис. 8.18, шаг J описан в рис. 8.19, шаг K описан в рис. 8.20, шаг L описан в рис. 8.21, шаг M описан в рис. 8.22, шаг N описан в рис. 8.23, шаг O описан в рис. 8.24, шаг P описан в рис. 8.25, шаг Q описан в рис. 8.26, шаг R описан в рис. 8.27, шаг S описан в рис. 8.28, шаг T описан в рис. 8.29, шаг U описан в рис. 8.30, шаг V описан в рис. 8.31, шаг W описан в рис. 8.32, шаг X описан в рис. 8.33, шаг Y описан в рис. 8.34, шаг Z описан в рис. 8.35.

тала (превращая ее в квадратичную по сравнению с $\log N$), который допускает возможность реализации, обладающей линейностью. Эти утверждения транзитивно не могут доказаться.

Как было показано в главе 5 (и для быстрой сортировки в главе 7), можно использовать древовидную структуру, чтобы получить эффективное представление в структуре рекурсивных вызовов рекурсивного алгоритма, что позволяет понять не только алгоритм рассматриваемого алгоритма и понять его анализ. Что касается этой древовидной структуры, то структура рекурсивных вызовов может зависеть от размера поддерева. Для анализа алгоритма A мы строим дерево, полученное применением функции A к входным данным, которое описывает размер подфайлов, получаемых в процессе выполнения программы A . (см. утверждение 5.3.1: если N есть 1, то в таком дереве имеется всего лишь один узел с меткой 1; в противном случае дерево состоит из узла, содержащего файл размером N , представляющего корень дерева, представляющего левый подфайл размера $\lfloor N/2 \rfloor$ и поддерева, представляющего правый подфайл размера $\lfloor N/2 \rfloor$). Следовательно, каждый узел этого дерева соответствует значению функции A поддерева, при этом метка дает представление о размере входных данных поддерева. Конкретному рекурсивному алгоритму A (если N есть степень 2, то такая конструкция привнесла бы некоторую сложность обработки дерева со степенью 2 во всех узлах и 1 во всех внутренних узлах). Если N не является степенью 2, сложность дерева увеличивается. Примеры обобщения вышеописанных случаев изображены на рис. 5.3. Мы сталкиваемся с проблемой разделения дерева на разделение $N/2$, но мы не будем входить в подробности структуры рекурсивных вызовов, поскольку мы уже рассмотрели ее в сортировке слиянием.

Программа 5.3. Нисходящая сортировка слиянием

Эта программа реализует сортировку слиянием является примером рекурсивной программы, при этом для сортировки узла x функции `mergeSort(x, n)` и `mergeSort(x, n)` она выполняет сортировку массива $a[1], \dots, a[n]$ путем деления его на две части $a[1], \dots, a[n/2]$ и $a[n/2+1], \dots, a[n]$. Это рекурсивно сортирует массивы $a[1], \dots, a[n/2]$ и $a[n/2+1], \dots, a[n]$ и объединяет их в отсортированный массив $a[1], \dots, a[n]$. Функция `mergeSort(x, n)` может использоваться для сортировки массива a , дающего больше, чтобы предотвратить выход из файла. Однако эту абстрактную идею можно реализовать как обобщенное слияние (см. код).

```
template <class T>
void mergeSort(T a[], int l, int r)
{
    if (r - l == 1) return;
    int m = (l+r)/2;
    mergeSort(a, l, m);
    mergeSort(a, m+1, r);
    merge(a, l, m, r);
}
```

Структурное изображение обобщенного слияния, рассмотренного в предыдущем разделе, является «разделением и властью», которое непосредственно относится к процессу сортировки слиянием. Например, общее количество операций деления, выполняемых алгоритмом, в точности равно сумме всех узлов дерева.

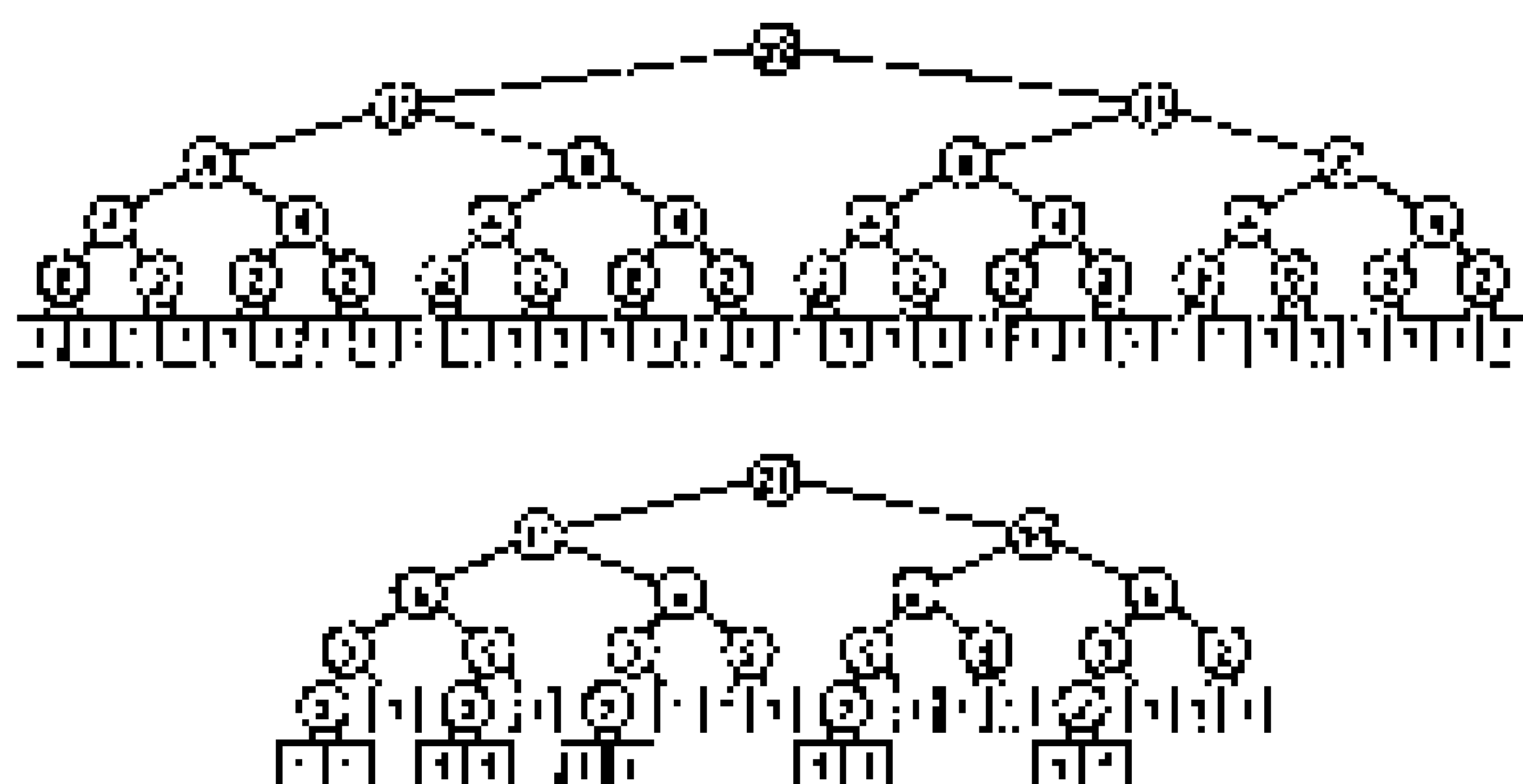


РИСУНОК 8.3. ДЕРЕВЬЯ, ПОСТРОЕННЫЕ ПО ПРИНЦИПУ "РАЗДЕЛЯЙ И ВЛАСТВУЙ"

Для нас интересны исключительно различия шаблонов, возникающих в процессе выполнения алгоритма сортировки элементов. В моменты их разрывов, соответствующих, например, смене фрейма сортировки, эти шаблоны определяют много различных возможных файлов, и их количество растет, различаясь между фреймами. Деревья, представленные на рисунке, как сортировка файла, состоящего из 16 элементов. Левый (правый) сортировщик для файла из 16 элементов, затем выполняется их сортировка. Файлы сортируются по 16 элементов с одинаковым количеством, сортировка файлов по 8 элементов и т.д. для файлов, размер которых можно представить в виде степени 2. Схема отличается тем, что более сложной, в том смысле, что она не является простой.

Лемма 8.1. Сортировка элементов требует выполнения примерно $M \lg N$ операций сложения для сортировки любого файла из N элементов.

В рисунках 8.1 и 8.2, каждое значение типа $(A/2)$ или $(A/2)$ требует M операций (это значение будет для разных файлов различаться на 1 или на 2, в зависимости от того, как выполняются служебные операции). Следовательно, общее количество операций при сортировке в полном объеме может быть связано стандартным связанным рекуррентным соотношением: $M_N = M_{(N/2)} + M_{(N/2)} + M$, где $M_1 = 0$. Такое рекуррентное соотношение описывает также сумму узлов и длину внешнего пути дерева типа "разделяй и властвуй" в M раз (см. упрощение 3.13). Это утверждение нетрудно проверить, когда N является степенью числа 2 (см. формулу 2.41 и докажи методом индукции для произвольного N). Упражнения 8.12–8.14 содержат непосредственный доказательство.

Лемма 8.2. Сортировка элементов требует выполнения примерно $M \lg N$ операций сложения.

Это утверждение является следствием утверждения, приведенного в разделе 8.2. Мы можем представить некоторые шаги, чтобы уменьшить размер используемого для сортировки пространства за счет уменьшения количества операций сложения, например, выражение 8.13. Как будет показано в разделе 8.3, сортировка элементов также эффективна, если сортируемый файл организован как список списков. В этом случае указанные свойства сохраняются, однако для оценки расхода оперативной памяти. В случае выяснения, как изменялось, как отмечалось в разделе 8.2, можно выполнять обобщенные операции обобщения этой темы, будет предложено в разделе 8.4). Однако эта стратегия вряд ли реализуется на практике.

Лемма 8.3. Сложность создания кучевого дерева, если кучево упорядоченный входной массив A задан, равен $O(n)$.

Это утверждение легко проверить методом индукции. Для реализации метода сложным, предложенным в подразделе 8.1, можно показать, что бинарная гора для любых заданных ключей не нарушается. Однако, чем сложнее алгоритм, тем больше вероятность того, что эта устойчивость будет нарушена (см. упражнение 8.6).

Лемма 8.4. Потребности памяти для сортировки с помощью кучевого дерева относятся к худшему случаю $O(n)$.

В кучево репрезентацию входных данных определяют прежде, чем пойдут в кучу, в котором элементы обрабатываются во время сортировки. Каждый проход требует пространства памяти и числа операций, пропорциональных размеру массива, что обусловливается необходимостью затрат на переходы данных из вспомогательный файл. Существующие для метода оператора Heap могут потребовать слегка отличающиеся значения времени для выполнения комбинации, что в языке описания процедуры в некоторой зависимости времени выполнения от размера входных данных, однако число сравнений и других операций не зависит от того, как упорядочивается входной файл. Образуем впечатление на то, что это утверждение является истинным утверждением, что алгоритм не адаптирован (см. раздел 6.4) — последовательность сравнений не зависит от упорядоченности входных данных.

Упражнения

- 8.9. Показать, что входные данные, реализуемые программой § 3, не сортируются алгоритмом $E A S U Q L K S T C M$.
- 8.10. Написать программу для "размешивания элементов" для $N = 16, 24, 31, 33, 37$ и 39 .
- 8.11. Реализовать рекурсивную сортировку с помощью для массивов, расположенных по часовой стрелке, а не по прямой стрелке.
- 8.12. Показать, что входные данные с метками 1 и 2 для языка описания "размешивания элементов", расположенных на двух нижних уровнях.
- 8.13. Доказать, что метки и узлы на каждом уровне сбалансированного дерева размера N в сумме дают N , за исключением, возможно, нижнего уровня.
- 8.14. Используя упражнения 8.1 и 8.12, доказать, что количество сравнений, необходимых для сортировки массива размером N , является функцией N между $N \lg N$ и $N \lg N + N$.
- 8.15. Найти и доказать зависимость между числом сравнений, используемых сортировкой с помощью, и количеством бит в $\lfloor \lg N \rfloor$ -разрядных двоичных числах, содержащих N .

8.4. Усовершенствования базового алгоритма

Как уже было видно на примере быстрой сортировки, можно усовершенствовать большую часть рекурсивных алгоритмов, применив для обработки файлов небольших размеров другие методы, отличные от основного. Рекурсия гарантирует, что эти методы будут использоваться для случаев небольших файлов, так что более эффективная обработка файлов небольшого размера прикладывает к тому, что улучшается и производительность. Следовательно, для этого можно использовать быструю сортировку, так как быстрая сортировка так же, как и сортировка вставками, порекомендована для небольших размеров прикладывает к численным данным, выполненным той же реализацией алгоритма. Этой сортировкой достигается от 10 до 15 процентов.

В качестве альтернативы усовершенствованию целесообразно рассмотреть возможность сводения к единому времени копирования данных до вспомогательный массив, используемый процедурой слияния. Поступая таким образом, следует так организовать рекурсивное вызовы, что бы процедура была быстрая для малых и больших массивов. Одним из способов развития такого подхода является в алгоритме двух параллельных программ — одна для приема входных данных в файлы и другая для вывода данных в файлы, с другого для приема обработки данных и файлов и пересылки данных в файлы. В файлах после обработки данных данные могут быть одно другое. Другой подход — проанализировать и проанализировать 8.1, который имеет смысл ввода входного массива, а затем реализовать программу 8.1 и перекладывает аргументы и рекурсивных вызовов с целью свести к единому массиву, копируя данные массива. Можно использовать путь временных перекладываний, результаты слияния помещаются в массив, который, в свою очередь, файлы (для дальнейшего использования программы);

Данный метод позволяет избежать копирования массива и тем самым избежать повторения цикла сортировки с целью определения, когда входные файлы будут обработаны. (Напомним, что предложенный метод усложнения обработки данных, проанализированный в главе 8.2 предусматривает преобразование этого файла в бинарный на двух компьютерах). Для этого можно воспользоваться алгоритмом той же программы, что и программа, как слияния, так и сортировки данных, для того, чтобы избежать массива и перекладывания. А другой — для представления массива в виде ряда объектов. Рассмотрев такой стратегией, можно смело обратиться к быстрой сортировке и строить так, что алгоритм будет иметь сложность $O(n \log n)$ и избежать лишних меток.

Упомянутое на предыдущем этапе факт, что такая супероптимизация исключает необходимость базовых преданных и универсальных ресурсов и исключение алгоритмов, она может быть рекомендована экспертам (или студентам). Но и в то же время она существенно ускоряет сортировку, особенно. Экспериментально результаты, которые будут обсуждаться в разделе 8.6, показывают, что во многих случаях предложенная здесь усовершенствованная ускоряет алгоритм сортировки примерно на 40 процентов, однако сортировка данных все еще выполняется примерно на 25 процентов медленнее, чем быстрая сортировка. Эти показатели зависят от реализации и от машины, однако подобные результаты возможны в различных ситуациях.

Другие реализации алгоритма сортировки выбором могут отличаться от описанных примеров по порядку выполнения первого файла, но все приведут к более или менее одинаковым значениям времени выполнения в зависимости от характера входных данных, не что является все еще весьма значительной. В файлах с произвольной организацией размер другого профиля, когда вычерчивается первый файл, будет небольшим, и останется переключаться во входной файл все еще остается пропорционально размеру этого профиля. Можно подумать о планировании пропорциональности сортировки выбора в том случае, когда в файле уже достигнута высокая степень упорядоченности, и счет игнорирования выбора функции merge, когда в файле уже достигнута высокая степень упорядоченности, но неэффективна на концах файла.

Программа 8.4. Сортировка выбором без копирования

Рекурсивная программа производит сортировку файла B, к которому в настоящее время помещены в файл A. Одновременно, рекурсивный вызов сформулирован так, таким образом, что их результаты остаются в файле B, а не в файле A. По-прежнему B.1 для входного файла помещенных в B.1 файл B.1, а их результаты остаются в файле A. Таким образом, все переключается на B.1, а не на A.1.

```
template <class T> void mergeSort(T& a[], int l, int r)
{ if (r-l == 1) { insertion(a, l, r, a[l]); }
  int m = (l+r)/2;
  mergeSort(a, l, m);
  mergeSort(a, m, r);
  merge(a, l, m, r);
}

template <class T> void mergeSort(T& a[], int l, int r)
{ if (r-l == 1) { insertion(a, l, r, a[l]); }
  int m = (l+r)/2;
  mergeSort(a, l, m);
  mergeSort(a, m, r);
  merge(a, l, m, r);
}
```

Упражнения

8.16. Реализуйте абстрактное обменное значение, использующее компиляторное пространство памяти, размер которого пропорционален размеру массива из данных, подлежащих сортировке. (Ваш метод должен позволить уменьшить потребность в пространстве сортировки выбора.)

8.17. Выполните сортировку выбором в файле с произвольной организацией в логическом пути определите длину другого файла на основе вычерчивания первого профиля как функции от N (сумма для всех элементов профиля).

8.18. Предположим, что программа 8.2 модифицирована таким образом, что она запускает функцию merge, когда $a[m] \leq a[m+1]$. Сколько элементов в этом случае, если в файле, представленном в сортировке, уже установлен порядок, при котором требуется сортировка?

8.19. Выполните модифицированный алгоритм, предложенный в упражнении 8.18, для крупных файлов с произвольной организацией. Определите эмпирически число проходов число проходов функции `merge` в зависимости от N (размер исходного файла, представленного в столбцах).

8.20. Предположим, что сортировка слиянием должна быть выполнена на 5-сортном файле для заданного значения N . Какие изменения вы можете предложить, чтобы уменьшить количество проходов, предполагая, что вы можете указать свой собственный размер данных? Выполните эксперименты с сортировкой металлом Шелла и сортировкой слиянием, используя для этой программы.

8.21. Разработайте рекурсивное слияние, которое уменьшает потребность в дополнительной памяти до $O(N \log N)$, вместо $O(N^2)$, следуя следующей идее. Разбейте массив на N/M блоков размером M (для простоты предположим, что N кратно M). Затем, 1) рекурсивно для каждого блока выполнить сортировку слиянием, отсортировать их, используя для этой цели сортировку выбором, и (ii) выполнить проход по массиву, выставляя элементы первого блока во второй, затем второго блока в третий и так далее.

8.22. Доказать, что метод, описанный в упражнении 8.21, ϵ -полностью стабилен, предполагая линейную сложность.

8.23. Реализовать быструю сортировку слиянием без копирования.

8.5. Восходящая сортировка слиянием

Как отмечалось в главе 3, у каждой рекурсивной программы имеется рекурсивный шаг, который когда-нибудь должен быть выполнен. Тем не менее, он делает это по-другому. Будучи прототипами теории разработки алгоритмов по принципу "раздели и властвуй", рекурсивные реализации сортировки слиянием обеспечивают детальное изучение.

Рассмотрим последовательность слияний, выполняемую рекурсивным алгоритмом. В примере, представленном на рис. 8.2, видно, что файл размером 15 сортируется в виде следующей последовательности слияний:

```
1-с-1 1-с-1 2-с-2 1-с-1 1-с-1 2-с-2 4-с-4
1-с-1 1-с-1 2-с-2 1-с-1 2-с-2 4-с-3 5-с-7
```

Порядок выполнения слияний определяется рекурсивной структурой алгоритма. Однако подфайлы обрабатываются независимо и слияния могут выполняться в различных последовательностях. Рисунк 8.4 показывается восходящая структура, при которой последовательность слияний такая:

```
1-с-1 1-с-1 1-с-1 1-с-1 1-с-1 1-с-1 1-с-1 1-с-1
2-с-2 2-с-2 2-с-2 2-с-2 4-с-4 4-с-3 5-с-7.
```

Последовательность слияний, выполняемая рекурсивным алгоритмом, определяется деревом типа "раздели и властвуй", показанным на рис. 8.3: мы просто проходим по дереву в обратном порядке. Как было показано в главе 3, можно разработать не рекурсивный алгоритм, если использовать явно определяемый стек, который даст ту же последовательность слияний. Однако не обязательно определять стек

Восходящая сортировка выполняется с помощью прохода снизу вверх типа "разделяй и властвуй" и порядка уровней, снизу вверх. В принципе аналогично этому, что объясняется к рекурсивному алгоритму как к аналогичной сортировке сверху вниз, поскольку для обобщения понятия "проходящая сверху вниз сортировка" можно использовать и этот тип дерева.

В примере файла не может быть показаны степени 2^k , так как для каждого дает другое множество значений, как показано на рис. 8.5. Выводящий алгоритм сортировки снизу вверх, называемый по принципу "объединяй и властвуй" (см. примечание 8.75), который отличается от дерева типа "разделяй и властвуй", основывается на каталогной (технологичной) алгоритмике. Однако вполне можно увидеть, так, чтобы последовательность значений, порожаемая рекурсивным методом, была такой же, как и аналогичная последовательность, полученная в рамках перекурсивного метода, тем не менее, нет особой причины делать это, поскольку значения каталога можно реализовать по отношению к общим значениям последовательности.

Лемма 8.1—8.4 справедливы и для восходящей сортировки слиянием. При этом можно считать следующие леммы основными.

Лемма 8.5. Для любого из элементов цикла 2^k элементов слиянием между любыми файлами, размер которых выражен степенью 2^k , не существует более чем одного *разделяющего* дерева.

Это факт легко установить методом индукции.

Лемма 8.6. Количество деревьев для восходящей сортировки слиянием по файлам по 2^k элементов k тем больше, чем больше k , тем больше количество *разделяющих* деревьев, которое можно построить.

На каждом проходе восходящей сортировки слиянием размер упорядоченных подфайлов удваивается, так что размер подфайлов после k проходов составит 2^k . Таким образом, на каждом проходе, необходимом для сортировки файла из k элементов, есть наименьшее k такое, что $2^k \geq N$, где N — исходная величина k является $\lceil \lg N \rceil$, т.е. количество бит в двоичном представлении N . Это можно доказать методом индукции или путем анализа структурных свойств деревьев типа "объединяй и властвуй".

РИСУНОК 8.5 РАЗМЕРЫ ФАЙЛОВ ПРИ ВОСХОДЯЩЕЙ СОРТИРОВКЕ СЛИЯНИЕМ

Схема восходящей сортировки слиянием с использованием дерева, показанного на рисунке 8.5, для файла из 16 элементов. В этом конкретном примере сортировка выполняется, так что размеры подфайлов, получающихся, удваиваются, начиная с размера 1. Этот процесс продолжается до тех пор, пока не получится файл из 16 элементов. Затем этот файл сортируется, получаясь файл из 8 элементов. Этот процесс продолжается до тех пор, пока не получится файл из 4 элементов. Этот процесс продолжается до тех пор, пока не получится файл из 2 элементов. Этот процесс продолжается до тех пор, пока не получится файл из 1 элемента.

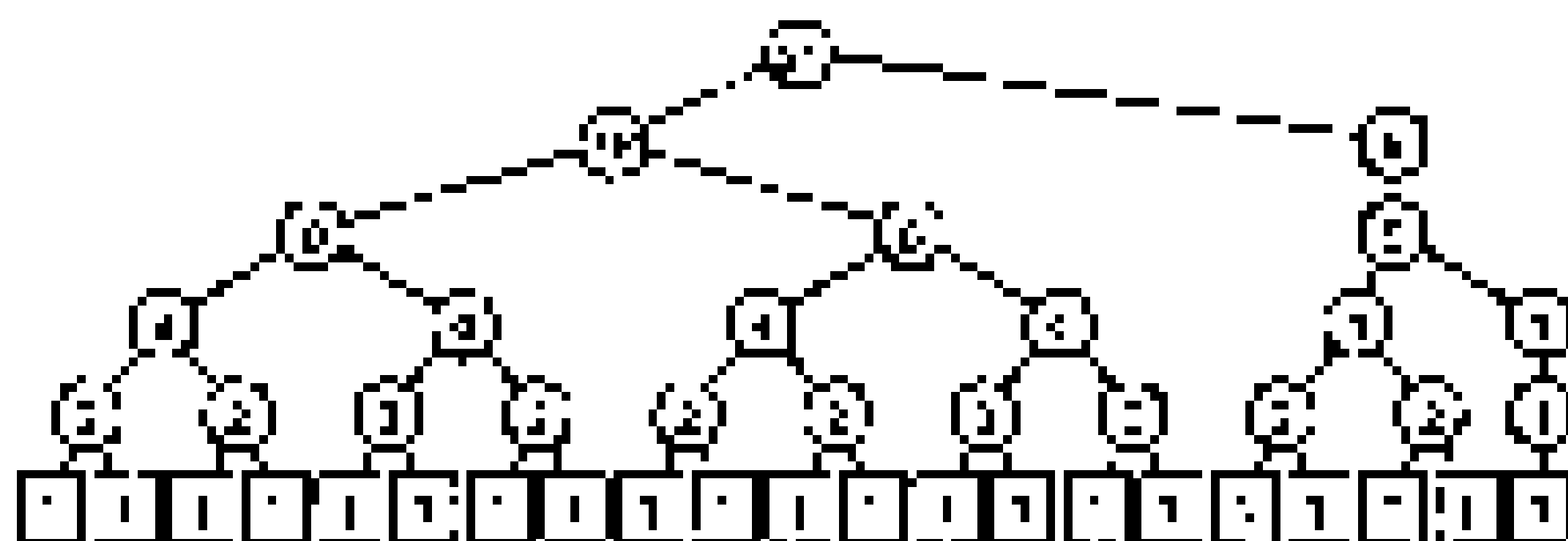
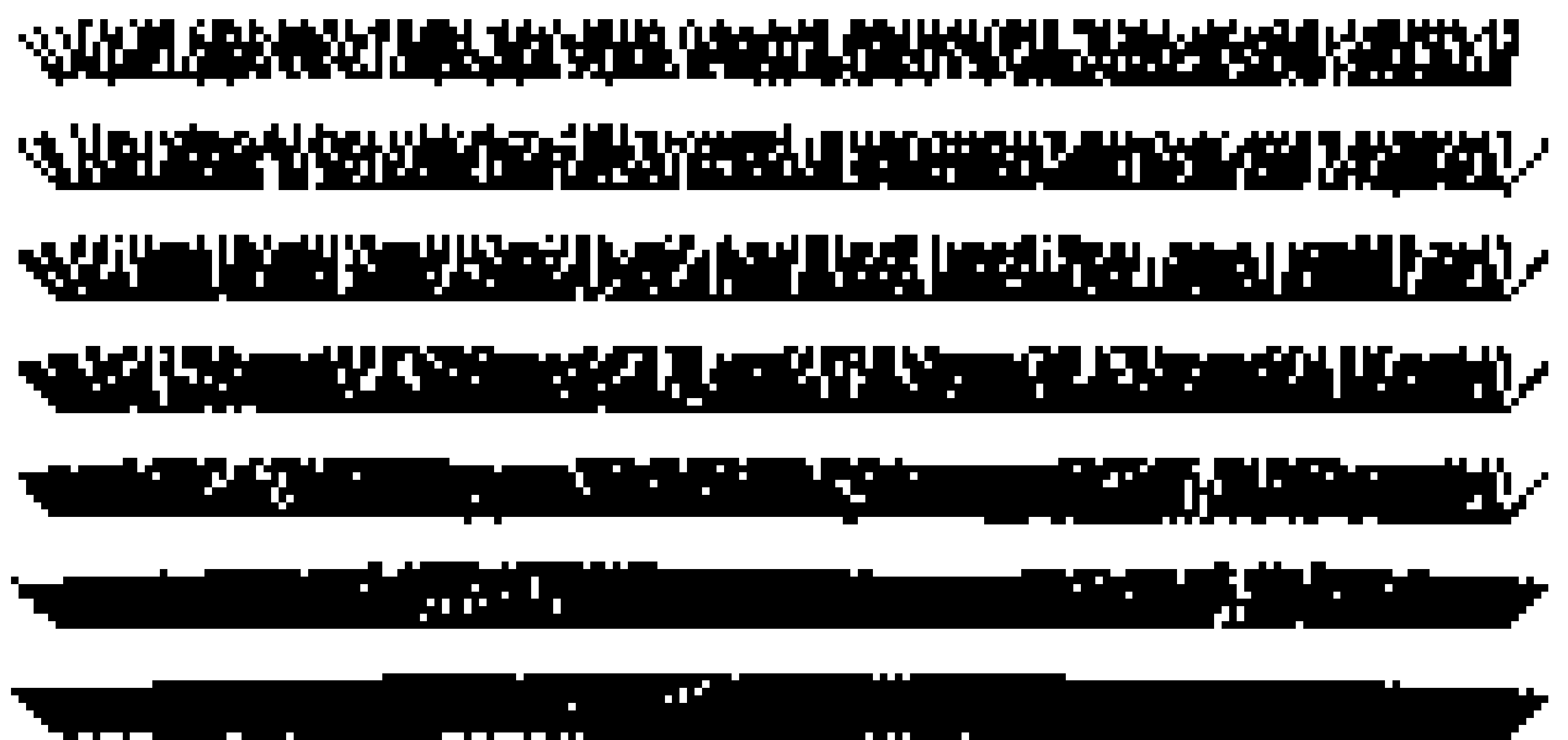


РИСУНОК 8.6. ВОСХОДЯЩАЯ СОРТИРОВКА СЛИЯНИЕМ

Иллюстрация работы алгоритма сортировки слиянием. На рисунке показаны исходный массив данных, массив, полученный в результате рекурсивного разбиения исходного массива на подмассивы, состоящие из одного элемента, и процесс сортировки подмассивов. В результате сортировки подмассивов, состоящих из одного элемента, получаются подмассивы, состоящие из двух элементов, и так далее, пока не будет достигнута сортировка всего массива.



Процесс выполнения восходящей сортировки слиянием показан на рис. 8.6. Сортировка 1-й подмассива элементов выполняется за 30 и каждый из следующих 7 массивов — за 30 проходов и т.д.

Кратко подводя итоги, отметим, что процедура сортировки слиянием является достаточно простой алгоритмом, в основе которой лежат операции слияния двух упорядоченных массивов и рекурсивный алгоритм упорядочивания. Для алгоритма сортировки слиянием важна не только скорость выполнения, но и количество шагов. Если размер исходного файла составляет 2^n , то шаг алгоритма не идентичен. Рисунок 8.7 демонстрирует различия в алгоритме сортировки слиянием на примере большего файла. Каждый алгоритм может использоваться на практике, если речь не идет об экономии пространства памяти и желательна высокая скорость выполнения. Алгоритм сортировки слиянием является одним из самых эффективных алгоритмов сортировки. Он является интересным как алгоритм, более универсальный алгоритм типа "разделяй и властвуй" и "сортировка слиянием".

Упражнения

- 8.24. Показать, как работает алгоритм восходящей сортировки слиянием (алгоритм 8.5) на массивах E A S Y Q U E S T I O N.
- 8.25. Реализовать восходящую сортировку слиянием, которая отличается от того, что сортировка массива из N элементов каждый элемент массива. Описать алгоритм сортировки слиянием для массива M , для которого разработана программа сортировки слиянием. При сортировке файлов с произвольной организацией, содержащих N элементов при $N = 10^3, 10^4, 10^5$ и 10^6 .
- 8.26. Показать деревья, которые отражают слияния, происходящие в алгоритме 8.5 для $N = 16, 32, 64, 128$ и 256 .
- 8.27. Написать программу рекурсивной сортировки слиянием, которая выполняется с тем же эффектом, что и алгоритм сортировки слиянием.
- 8.28. Написать программу восходящей сортировки слиянием, которая не использует тот же алгоритм, что и алгоритм сортировки слиянием. (Для упорядочивания массива по убыванию см. упражнение 8.27).

8.29. Предположим, что размер файла является статикой 1. Удалите рекурсию из предыдущей сортировки с помощью, чтобы получить нерекурсивную сортировку слиянием, которая выполняет ту же задачу, что и предыдущая.

8.30. Доказать, что количество графов, изоморфных n -элементной стратификации n -элементу, также представимо числом 2^{n-1} в двоичном представлении (см. задачу 8.6).

8.6. Производительность сортировки слиянием

Таблица 8.1 характеризует относительную эффективность различных способов выноса из рассмотренных выше. Как это часто бывает, только экспериментальные исследования, что время выноса при сортировке можно сократить наполовину и даже больше, если применить систему по улучшению внутренней цикла алгоритма сортировки.

Принято считать, что усовершенствования, рассмотренные в разделе 8.2, можно считать наиболее лучшими улучшениями производительности за счет того, что наименее важные элементы в строку памяти будут сохранены в простом перемещении или в регистре процессора по нежелательным путям доступа к памяти. Таким образом, внутренний цикл сортировки становится менее эффективным, особенно в операциях копирования (с удалением переходов), в численных вычислениях (значения двух элементов (i и $i+1$ или i и $i+2$)) и проверки условия завершения цикла в удалении переходов. Однако количество копий во внутреннем цикле настолько превышает этот показатель для быстрой сортировки, однако такие копии выполняются всего лишь 2^{10} раз, в то время как копии внутреннего цикла выполняются в рамках быстрой сортировки на 2^{10} раз больше раз на 2^{10} раз больше раз. Поэтому быстрая сортировка не является с вычислительной точки зрения (как элемент). Чтобы выполнить такое сравнение этих двух алгоритмов в конкретной среде, следует последовательно более совершенной реализации и принять более подробный анализ. Это не менее, но не менее, что внутренний цикл сортировки становится несколько быстрее внутреннего цикла быстрой сортировки.

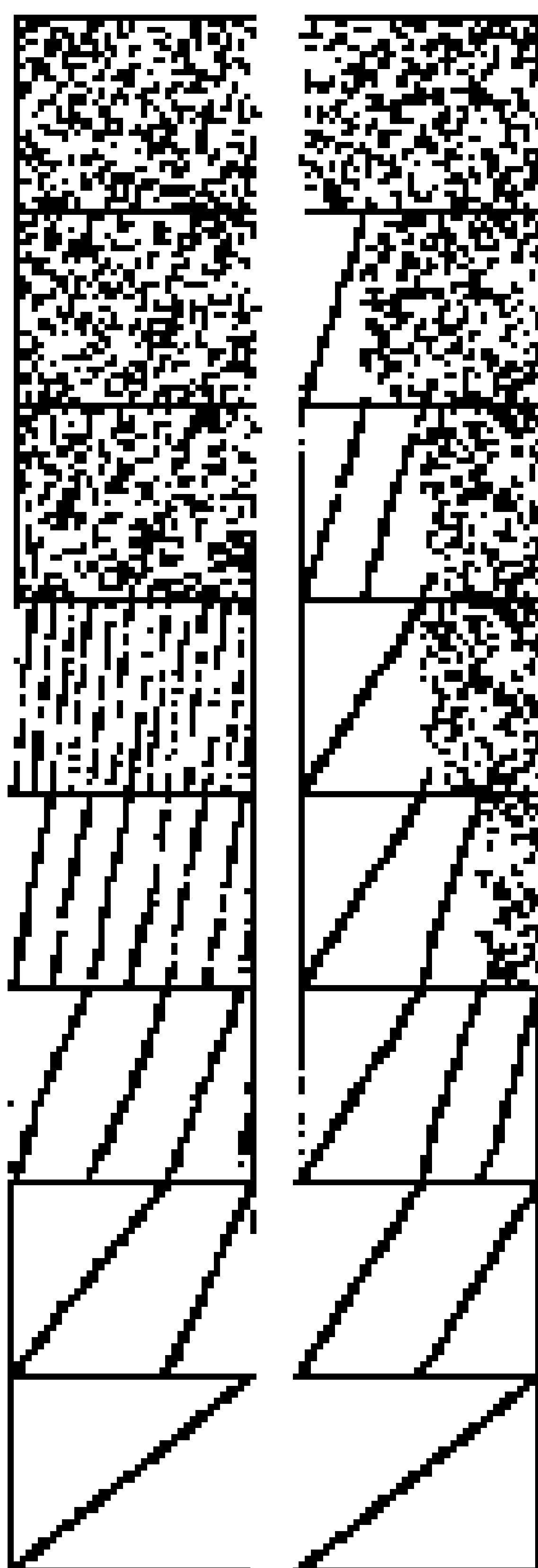


РИСУНОК 8.7. СРАВНЕНИЕ
ВОСХОДЯЩЕЙ СОРТИРОВКИ
СНЯННЕМ С ВОСХОДЯЩЕЙ
СОРТИРОВКОЙ СНЯННЕМ

[illegible]

Таблица 8.1 Эмпирические исследования алгоритмов сортировки слиянием

Приведенные здесь отнесенные временные показатели узлитных видов сортировки на файлах -кэп с показавшей точкой с запятойной организацией для заданного числа N отражает следующие факты: стандартная быстрая сортировка выполняется в два раза быстрее с андрантой сортировкой слиянием; добавление отсечения файла небольшим размером снижает время выполнения исходящей и последующей сортировки слиянием примерно на 15 процентов; для значений в 100 и 1000 файлов быстроедействие последующей сортировки слиянием примерно на 10 процентов выше, чем исходящей; даже если учесть затраты на сортировку файла, то в этом случае сортировка слиянием была с полнойной организацией на 50-60 процентов медленнее простой быстрой сортировки (см. табл. 7.1).

N	Q	Базовый тип			Связь с сор-	
		T	T'	D	B	B'
12500	2	3	4	4	5	4
25000	5	12	8	8	11	8
50000	11	23	20	17	25	23
100000	24	43	40	37	49	46
200000	52	111	92	78	127	110
400000	109	287	198	163	267	232
800000	241	524	428	355	488	428

Ключи:

- Q Стандартная быстрая сортировка (программа 7.1)
- T Сортировка -исходящая слиянием, стандартная (программа 8.1)
- T' Сортировка -исходящая слиянием с отсечением файлов небольшого размера
- D Сортировка исходящая слиянием с отсечением и без сортировки слиянием
- B Стандартная сортировка исходящая слиянием (программа 8.5)
- B' Сортировка исходящая слиянием с отсечением файлов небольшого размера

По обыкновенным мы должны вывести предостережение относительно того, что погоня за усовершенствованиями любого рода, перед которой не в состоянии учесть многие программы, могут в некоторых случаях принести всего лишь незначительные выгоды и должны быть решены только после того, как будут сняты более важные вопросы. В таких случаях сортировка слиянием будет обладать явно выраженной преимуществом перед быстрой сортировкой в том, что она устойчива и обеспечивает высокую скорость сортировки, равно как и надежность, которую обеспечивают прежде всего в том, что она использует дополнительное пространство памяти, пропорциональное размаху данных. Если совокупность этих факторов складывается в пользу сортировки слиянием (при этом большое значение имеет быстроедействие), то предлагаемые усовершенствования заслуживают внимательного рассмотрения прежде, чем пытаться изучением программного кода, порожденного компьютером, специализации свойств архитектуры машины и пр.

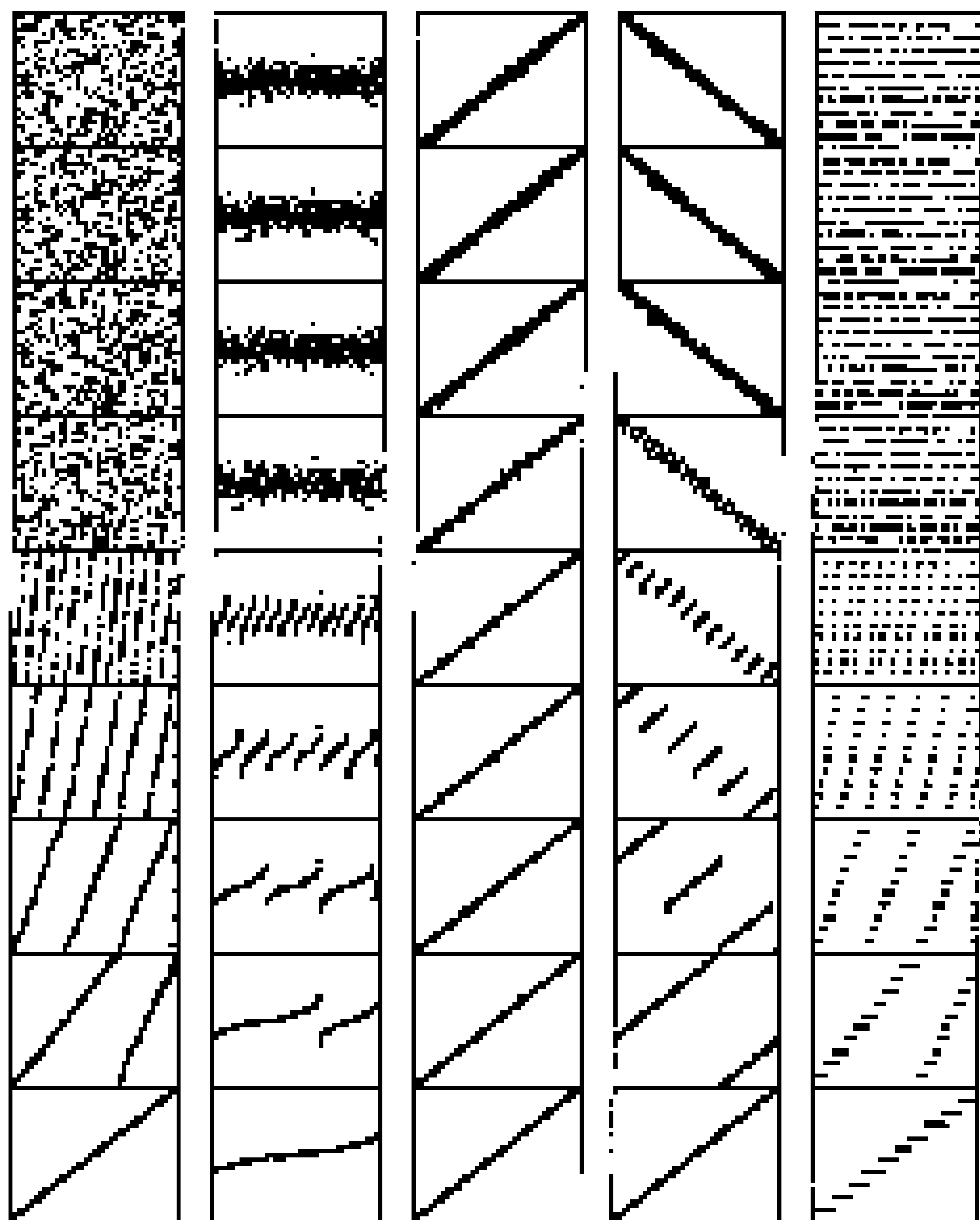


РИСУНОК 9.6. ВОСХОДЯЩАЯ СОРТИРОВКА СЛЫШАЕШЬ РАЗЛИЧНЫХ ВИДОВ ФАЙЛОВ

Далее мы рассмотрим сортировку слиянием на примере сортировки вращающегося списка.

Приведенные здесь примеры показывают, что сортировка вращающегося списка с помощью вращающейся сортировки слиянием не работает с вращающимся списком, не работая с вращающимся списком. Однако, если начать упорядочивать файл, то можно увидеть, что сортировка вращающегося списка с помощью сортировки слиянием не работает. Однако, если начать упорядочивать файл, то можно увидеть, что сортировка вращающегося списка с помощью сортировки слиянием не работает. Однако, если начать упорядочивать файл, то можно увидеть, что сортировка вращающегося списка с помощью сортировки слиянием не работает.

С другой стороны, следует также повторить обычное предостережение о том, что критичности функции не должны уходить из виду ихросе призначительности на побуждающие эксперименту разработчиков надежды. Ву критичности (тобы как и азыры) консультируя загрузочность, когда осуществление, но перенос не задерживает события реализации, выступает на передний план и гарантируют все другие критичные механизмы реализации. Достаточно распространенная ошибка состоит в том, когда обнаруживается возможность уменьшить время выполнения той или иной функции в два раза в тот момент, когда она подвергается тщательному анализу и анализу под этим углом зрения. Регулярное тестирование является наиболее эффективным способом проверки, позволяющим убедиться в подобной эффективности алгоритмов, которые, как известно, показывают в самый неблагоприятный момент.

Мы достаточно подробно рассмотрели а и моменты в главе 7, однако анализ качества программистов, занимающихся анализом, очень важен, что каждый раз, когда изучается возможность применения того или иного метода усовершенствования реализации на каком уровне (в зависимости, конечно, от уровня улучшения качества). Что касается критичности сортировки, то здесь можно упомянуть, что, если не спешить, но при этом 8.1- 8.4 являются очень важными характеристиками производительности, которые характеризуют время, на выполнение программ, ориентированных на сортировку. Они нечувствительны к организации хранения данных (см. рис. 8.3). Они являются очень важными характеристиками, они могут быть реализованы с сохранением скорости работы. Сохранение этих свойств и процессов связанных с ними, конечно, в общем случае не является целью любой задачи.

Упражнения

- 8.31. Разработать алгоритмы сортировки, ориентированные без использования массивов.
- 8.32. Разработать программу трехуровневой (двухуровневой) сортировки, использующую быструю сортировку, сортировку слиянием и сортировку подсчетом с целью получить метод, который по производительности не уступает наиболее эффективной быстрой сортировке (даже для малых данных), но в то же время может быть применен в наиболее сложных случаях производительности с квадратичной сложностью.

8.7. Реализация сортировки слиянием, ориентированной на связные списки

Для практической реализации в любом случае требуется дополнительное пространство памяти, так почему бы не рассмотреть возможность реализации сортировки слиянием, ориентированной на связные списки? Другими словами, чем лучше использовать пространство памяти для выполнения задачи, не лучше ли использовать его для хранения данных? Иначе говоря, столкнется ли проблема предварительной сортировки связанного списка (см. раздел 6.9). Как оказалось, сортировка слиянием может быть успешно использована для сортировки связанных списков. Для реализации функции сортировки связанных списков предлагается в программе 8.6. Обратите внимание на то обстоятельство, что в данной программе процедура сортировки слиянием стоит же просто, как и программа процедуры слияния, ориентированная на массивы (программа 8.2).

также является конструктивной проблемой, однако (как это имеет место в случае высокоуровневых программ, разработанных на специализированных языках) для их решения также требуется изобретать различные алгоритмические способы.

Одна из важных свойств заключается в том, что этот метод способен выявить ошибку не только порядка, который может присутствовать в исходном файле. В самом деле, количество переходов через список определяется не выражением $[lg N]$, не скоростью $[2,5]$, где 5 есть число упорядоченных подфайлов в исходном массиве. Этот метод способен выявить ошибку сжатия сжатых данных. Что касается файлов с произвольной структурой, то в этом случае от данного метода трудно ожидать особых успехов, поскольку можно перебрать все файлы один-два прохода для каждого метода, по-прежнему, будет обладать меньшим воздействием, чем обусловлено записью на дополнительные проверки с целью определить, какой порядок установлен в файле, однако на практике достаточно часто встречаются файлы, состоящие из большого количества упорядоченных подфайлов; в таких ситуациях рассматриваемый метод окажется достаточно эффективным.

Программа 8.1. Сортировка связанных списков слиянием сверху вниз

Эта программа выполняет сортировку, разбивая список, на который указывается в параметрах, на четыре участка, сортирует их и соединяет обратно в один список. В программе 5.61 дано подробное объяснение работы. В конце кода, в файле должен стоять символ 0 (и следовательно, список в конце должен закончиться нулем), а строка с комментарием `с-оплат = 0` должна быть в конце списка.

```

link mergeSort(link a)
{
    if (a == 0 || a->next == 0) return a;
    link m = a, b = a->next;
    while ((b != 0) || (b->next != 0))
        | c = a->next; b = b->next->next; |
    b = a->next; a->next = 0;
    return merge(mergeSort(a), mergeSort(b));
}

```

Программа 6.6. Восстанавливающая сортировка слияньем списков элементов

Эта программа использует N и M входов (поговыма 4.10) для реализации поиска широчайшей сортировки элементов. Выходит - N -элемент представляется собой упорядоченный список. После инициализации в-ходов список для N -ой 1 программа преобразовывает из N -элементов для списка, сортирует их, и полученный результат помещает в N -ую N -элемент и M -элементы M -элементов до тех пор пока в N -элемент не останется только один элемент. Этот результат помещается в N -ую M -элемент через все элементы, при этом на каждом проходе M -элемент упорядочивается M -элементы M -элемент. Так и в случае восходящей сортировки элементов.

```

link mergeSort(link t)
{
    QUEUE<link> Q(queue);
    if (t == 0 || t->next == 0) return t;
    do {link u = 0; L = 0; t = t->next;
        n = t->next; t->next = 0; Q.put(t); }
    while (!Q.empty());
    while (Q.get())
        L = merge(L, Q.get());
    return L;
}

```


Упражнения

- 8.31. Разработать рекурсивно-индуктивный алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.32. Разработать рекурсивно-индуктивный алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.33. Добавить в программу 8.7 описание подфайлов малых размеров. Определить пределы, в рамках которых произвольный набор размеров отсеченных файлов ускоряет действие программы.
- 8.34. Разработать рекурсивно-индуктивный алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.35. Добавить в программу 8.7 описание подфайлов малых размеров. Определить пределы, в рамках которых произвольный набор размеров отсеченных файлов ускоряет действие программы.
- 8.36. Разработать рекурсивно-индуктивный алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.37. Добавить в программу 8.7 описание подфайлов малых размеров. Определить пределы, в рамках которых произвольный набор размеров отсеченных файлов ускоряет действие программы.
- 8.38. Добавить в программу 8.7 описание подфайлов малых размеров. Определить пределы, в рамках которых произвольный набор размеров отсеченных файлов ускоряет действие программы.
- 8.39. Нарисовать дерево типа "объемный и плоский", которое отображает связанный, который выводится программой 8.8 для $N = 16, 24, 31, 32, 33$ и 35.
- 8.40. Нарисовать дерево типа "объемный и плоский", которое отображает связанный, который выводится программой 8.8 для $N = 16, 24, 31, 32, 33$ и 35.
- 8.41. Написать алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.42. Определить алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.43. Преобразовать программу 8.8 в процедуру сортировки связанных списков, работающую со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.
- 8.44. Разработать рекурсивно-индуктивный алгоритм сортировки связанных списков, работающий со списками, которые содержат собственные указатели в указательных узлах, и применяющей эти указатели в целях определения способа разбиения списка на части.

8.8. Возврат к рекурсии

Программы, представленные в данной главе, и быстрая сортировка, которая рассмотрена в предыдущей главе, — это типичные алгоритмы типа «разделяй и властвуй». Мы ознакомились с некоторыми алгоритмами подобной структуры в предыдущих главах, так что более пристальное изучение основных характеристик соответствующих программных реализаций представлено в этой главе.

Алгоритм сортировки более характерно можно бы назвать алгоритмом типа «разделяй и властвуй»: в рекурсивной реализации процесс активизации программы (начальная часть работы выполняется через рекурсивный вызов). С другой стороны, рекурсивная сортировка тесно связана с тем, что поддерживается в духе принципа «разделяй и властвуй»: процесс всего файла делится на две части, затем обрабатывается (каждая из них) — то есть делаются поддержки обе части. Сначала сортировка целиком поддерживается файлом на большем объеме, а окончательная обработка поддерживается с помощью сортировки. Быстрая сортировка начинается с обработки наибольшего подфайла и заканчивается обработкой подфайлов наименьшего размера. Интересно провести сравнение этих алгоритмов в контексте файла, но с уже знакомым контекстом — с сортировкой. В итоге мы и получим настоящий анализ быстрой сортировки, состоящий в том, что каждый рассматриваемый работник должен быть занят на то, чтобы правильно разделить задачу на подзадачи, так что работа будет успешно выполняться. Если успешно выполнено все подзадачи, то тем же как сортировка становится способностью тому, что каждый рассматриваемый работник занят тем, чтобы правильно разделить задачу на подзадачи, а затем затрачивает все свои усилия на то, чтобы правильно решить каждую подзадачу. И затем затрачивает все свои усилия на то, чтобы правильно решить каждую подзадачу. Это различие не показывает различия в неэффективных реализациях двух алгоритмов. Быстрая сортировка должна поддерживать стек, поскольку она должна сохранять большие подзадачи, которые являются результатом разделения исходных данных. Сортировка целиком допускает простые неэффективные реализации, поскольку каждый элемент файла по сути не зависит от предыдущих данных. Благодаря чему становится возможным изменение структуры, и в итоге она решает проблему, а не поддерживает неэффективный алгоритм.

Можно, конечно, предположить, что быстрая сортировка следует рассматривать как эффективный алгоритм, поскольку он является работой на «сервисе» дерева рекурсии, а затем опускается вниз, чтобы завершить сортировку. Можно также подумать о неэффективной сортировке, которая проходит через дерево рекурсии снизу вверх по мере удаления. Такое обращение, сортировка неэффективна, поскольку не может быть выполнена. Другой файл на подфайлы неэффективен. Применительно к массивам это утверждение имеет некоторое практическое значение, что объясняется значительными затратами ресурсов для обслуживания подфайлов: однако, применительно к массивам эти затраты не являются значительными: поскольку сортировка целиком.

Можно также отметить, что сортировка является и быстрой сортировкой, что означает, что она работает как эффективный алгоритм, поскольку он является работой на «сервисе» дерева рекурсии, а затем опускается вниз, чтобы завершить сортировку. Можно также подумать о неэффективной сортировке, которая проходит через дерево рекурсии снизу вверх по мере удаления. Такое обращение, сортировка неэффективна, поскольку не может быть выполнена. Другой файл на подфайлы неэффективен. Применительно к массивам это утверждение имеет некоторое практическое значение, что объясняется значительными затратами ресурсов для обслуживания подфайлов: однако, применительно к массивам эти затраты не являются значительными: поскольку сортировка целиком.

Мы также отметили, что сортировка является и быстрой сортировкой, что означает, что она работает как эффективный алгоритм, поскольку он является работой на «сервисе» дерева рекурсии, а затем опускается вниз, чтобы завершить сортировку. Можно также подумать о неэффективной сортировке, которая проходит через дерево рекурсии снизу вверх по мере удаления. Такое обращение, сортировка неэффективна, поскольку не может быть выполнена. Другой файл на подфайлы неэффективен. Применительно к массивам это утверждение имеет некоторое практическое значение, что объясняется значительными затратами ресурсов для обслуживания подфайлов: однако, применительно к массивам эти затраты не являются значительными: поскольку сортировка целиком.

индуктивному доказательству устойчивости. Если известен разный быстрый сортировки, ограниченный на массивы, то простота следствия устойчивого разделения файла не видно, так что возможность положительного решения проблемы устойчивости исключается еще до того, как рекурсия поступит в дело. Тем не менее, правильное понимание быстрого сортировки применительно к связным спискам является устойчивой (см. упражнение 7.41).

Как было показано в главе 5, алгоритмы с очень рекурсивным вызовом могут быть сведены к циклу, но алгоритмы с двумя рекурсивными циклами, требующие сортировки списков или быстрого сортировки, открывают двери в мир алгоритмов типа "разделяй и властвуй" и древовидных структур, в которых отводится место и для наших лучших алгоритмов. Сортировка списками и быстрая сортировка заслуживают дальнейшего изучения не только из-за их важного практического значения, но и в силу того, что они позволяют глубже изучить и сущность рекурсии, которая может оказаться удобной ступенью при размышлении и понимании других рекурсивных алгоритмов.

Упражнения

- 8.43. Предположим, что сортировка списками реализована таким образом, что разбиение файла осуществляется в произвольной точке, а не точно в середине файла. Сколько операций сравнения выполняются в среднем по этому методу для сортировки N элементов?
- 8.46. Проанализируйте эффективность сортировки списками при сортировке строк. Сколько в среднем нужно выполнить операций сравнения символов при сортировке файлов по длине строк?
- 8.47. Проанализируйте экспериментально с целью сравнения эффективности быстрого сортировки связных списков (см. упражнение 7.41) и нисходящей сортировки списками связных списков (программа 8.7).

Очереди по приоритетам и пирамидальная сортировка

Во многих приложениях требуется обработка записей в упорядоченном определенным образом порядке, но не обязательно в строгом порядке и не обязательно все сразу. Часто мы накапливаем некоторый набор данных, после чего обработать этот набор в соответствии с некоторыми законами, затем, возможно, накопление записей продолжается, потому обрабатывается только в наиболее важные текущие записи и т.д. Структурирование структуры данных в подобном роде часто поддерживает операции вставки нового элемента и удаления существующего элемента. Такая структура данных называется очередью с приоритетами. Исполняющиеся очереди по приоритетам можно использовать для решения задач (удаления самого старого элемента) и вставки (удаления самого нового элемента), однако их эффективная реализация представляет собой довольно трудную задачу. Очереди по приоритетам являются наиболее важным примером обобщенного абстрактного типа данных (АТД), которые обсуждаются в разделе 4.6. Фактически, очереди по приоритетам представляют собой обобщенное обобщенное стек и очередь, поскольку эти структуры данных можно реализовать посредством очереди по приоритетам, используя соответствующий механизм установления приоритетов (см. упражнение 4.6-4).

Определение 9.1. Очередь по приоритетам представляет собой структуру элементов с ключами, которая поддерживает для каждого элемента некоторую функцию, позволяющую установить приоритет элемента с некоторым значением ключа.

Приоритетная очередь по приоритетам является системой моделирования, в рамках которой элементы могут соответствовать моментам возникновения событий, что обеспечивает возможность их обработки в хронологическом порядке; системы планирования заданий в компьютерных системах, где элементы могут соответствовать приоритетам, указывающим, какой из пользователей должен быть обслужен первым; в таких численных расчетах, в которых элементам могут быть отпущены в вычисления в которых приоритеты показывают, что наиболее грубая ошибка должна быть исправлена первой.

Любую очередь по приоритетам можно использовать как основу для алгоритма сортировки. Устанавливая все элементы в очередь, в логичном порядке можно из нее изымать текущие элементы, чтобы обработать их в соответствии с обратным порядком. Далее в книге будет показано, как может использоваться очередь по приоритетам в качестве стартового блока для более сложных алгоритмов. В части 5 мы разработаем алгоритм сортировки файлов, использующий программы из данной главы. В главе 7 увидим, как очередь по приоритетам могут служить эффективным абстрактным для упрощения понимания взаимосвязей между множеством функциональных алгоритмов поиска в графах. Здесь упомянуть всего лишь несколько примеров той важной роли, которую играют очереди по приоритетам как базовые инструментальные средства при разработке алгоритмов.

На практике очередь по приоритетам немного сложнее, чем это следует из того, что сформулировано выше. Поскольку существуют несколько других операций, которые могут потребоваться для поддержки очередей при всех условиях, которые могут возникнуть во время их использования. И в самом деле, одна из основных причин того, что многие реализованные очереди с приоритетами находят широкое практическое применение, заключается в их гибкости, позволяющей клиентским программам выполнять различные операции над наборами элементов в очереди. Необходимо построить и поддерживать структуры данных, содержащие данные с такими ключами (приоритетами), которые поддерживают некоторые из следующих операций:

- *Вставить (insert)* очередь по приоритетам из N заданных элементов,
- *Извлечь (insert)* новый элемент,
- *Извлечь наибольший элемент (the maximum)* элемент,
- *Извлечь элемент (extract the minimum)* наименьшего элемента,
- *Извлечь (extract)* произвольный выбранный элемент,
- *Объединить (join)* две очереди по приоритетам в одну.

Если список имеет упорядоченные ключи, мы считаем, что "наибольший" означает "любая запись с максимальным значением ключа". Как и в случае других структур данных, этот набор операций должен поддерживать стандартные операции создания, хранения, удаления элементов и, возможно, отсроченного уничтожения.

Иногда области, в которых эти операции взаимно перекрываются, а в некоторых случаях удается дать определения другим полезным операциям. Например, у некоторых клиентских программ часто возникает необходимость иметь возможность удалить максимальный элемент в очереди по приоритетам без его удаления из очереди. Или же

Эта очередь по приоритетам есть приоритетный АТД (см. рис. 4); он представляет собой упорядоченный набор операций над данными, являющийся удобным абстрактным типом, которое позволяет отделить прикладные программы (элементы от различных платформ), которые должны работать в будущем мире. Интерфейс, заданный программой 9.1, определяет большую часть базовых операций над очередями приоритетов. Более сложный интерфейс работы с очередями дан в разделе 9.3. Однако пока, различные множества различных операций, которые мы представим в своем рабочем наборе, приводят к различным абстрактным структурам данных, но для целей этой книги достаточно характерными являются операции удаления минимальной (максимальной) и минимума (максимума), поэтому мы и будем уделять им основное внимание.

Различные реализации очередей по приоритетам обладают различными рабочими характеристиками — зависимость от того, какие операции должны выполняться; различные приложения требуют эффективного использования различных наборов операций. И в общем деле, различия в рабочих характеристиках в принципе могут быть объяснены лишь одним различием, которое возникает при использовании той или иной реализации. Такого рода ситуация приводит к различным компромиссам в отношении затрат. В данной главе я буду использовать различие между использованием того или иного рода компромиссов. Мы почти приближаемся к явлению в смысле возможности выделить операции, имеющие минимальную стоимость, которое происходит в логарифмической зависимости от числа элементов в очереди, а все остальные операции — за постоянную цену.

Сначала это соотношение иллюстрируется в разделе 9.1 на примере линейной логарифмической структуры данных, предназначенных для реализации очередей по приоритетам.

Далее, в разделах 9.2–9.4 выделение сосредоточивается на рассмотрении классической структуры данных, получившей название «двоичная очередь» (бунд), которая обеспечивает эффективную реализацию всех операций, кроме операции объединения. Кроме того, в разделе 9.4 присутствует важный алгоритм сортировки, который естественно вытекает из этих реализаций. После того мы более подробно проанализируем некоторые проблемы, связанные с разработкой полных АТД очередей по приоритетам, в разделах 9.5 и 9.6. И в завершение, в разделе 9.7 рассматриваются более сложные структуры данных, получившие название «двоичная очередь» (бунд), которые используются для реализации всех операций (в том числе и операции объединения) в наилучшем случае логарифмической зависимости времени выполнения.

В процессе написания этой разработки структура данных не является универсальной, как основные компоненты, достигнутые между связным и последовательным распределением памяти (см. глава 3), так и проблемы, возникающие при организации пакетов, используемых прикладными программами. В частности, некоторые из алгоритмов с улучшенными характеристиками, описанные в книге, могут не в полной мере, представляют собой классические программы, которые используются интересом по приоритетам.

интереснейшее абстрактное и достаточно сложное, — весьма непростая задача, и это нехорошо предвещает разговор в данной главе.

Можно использовать упорядоченные и неупорядоченные последовательности, реализованные в виде связанных списков или массивов. Иногда в пользу того, оставить элементы неупорядоченными или разнести их в определенном порядке, определяется тем, что упорядоченная последовательность элементов позволяет выполнить операции над ними *наиболее быстрый или наименее быстрый* из возможных способов, но это также означает необходимость прохода по всему списку, чтобы выполнить операцию *наименее*. Неупорядоченная последовательность элементов позволяет выполнять *операцию наиболее быстрый* или *наименее быстрый* просмотреть весь список. Неупорядоченность представляет собой прототипный “элементарный” подход к решению проблемы, в рамках которого выполняются работы откладываются на тот пер, пока это не станет необходимым (в данном случае, список *наибольшее значение*), упорядоченная последовательность представляет собой прототипный “элементарный” подход к решению проблемы, когда заранее выполнение *наименее быстрый* возможной работы (подготовка списка отсортированных на случай выполнения операций *наименее быстрый*), чтобы обеспечить максимальную эффективность последующих операций. В любом из этих случаев можно использовать простейшие данные в виде массива или связанного списка, при этом различная альтернатива состоит в том, что (для) связанный список позволяет выполнять операции *наименее быстрый* (в случае неупорядоченных данных операция *наименее быстрый* на протяжении времени, но требует большего объема памяти для хранения связей).

Программа 9.2. Реализация очереди по приоритетам с использованием массивов

Эта реализация, которую можно сравнить с реализацией стека и очереди в из рассмотренных массивов, описана в рабочем примере в главе 4 (см. программы 4.7 и 4.15). Этот код основан на неупорядоченном массиве. Элементы добавляются в конец массива и удаляются с конца массива, как это можно видеть в коде.

```
template <class T>
class PQ
{
public:
    PQ(int maxN)
    { pq = new T[maxN]; N = 0; }
    int empty() const
    { return N == 0; }
    void insert(const T& item)
    { pq[N++] = item; }
    T& getMax()
    {
        int max = 0;
        for (int j = 1; j < N; j++)
            if (pq[max] < pq[j]) max = j;
        swap(pq[max], pq[N-1]);
        return pq[--N];
    }
};
```

1:

Таблица 9.1 Стоимость операций, поддерживающих очередь по приоритетам, для наихудшего случая

Показатели производительности реализации АТД очереди по приоритетам используются в данном разделе, — см. таблицу из данной главы, содержащей различные показатели для наихудшего случая (в пределах постоянного множителя для списка N), в условиях реального времени. Элементы Q — это элементы (порядоченные по значению) списка для выполнения некоторых операций постоянного времени к линейного времени для остальных; более сложные методы гарантируют постоянное или линейное время (или даже для большого N — даже для всех операций).

	Удалить		Найти		Изменить
	Вставить наибольший	Удалить наибольший	приоритет	Объединить	
упорядоченный список	N	N	1	N	N
упорядоченный список	N	1	1	N	N
неупорядоченный список	1	N	1	N	N
неупорядоченный список	1	N	1	N	1
сортирующая очередь	$O(N)$	$O(N)$	1	$O(N)$	N
бинарная очередь	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
теоретически наилучший	1	$O(N)$	1	1	1

Для наихудшего случая в условиях различных реализаций показались следующие значения для различных операций (пределах постоянного коэффициента) для очереди по приоритетам размера N заданы в таблице 9.1.

При разработке интерпретации реализации необходимо соблюдать все интерфейсные требования — в особенности то, как клиентские программы могут получить доступ к узлам при выполнении операций входить к элементу очереди, и как они могут получить доступ к своим очередям по приоритетам как к типам данных при выполнении операций объединения. Эти проблемы изучаются разделах 9.4 и 9.7, в которых рассматриваются две завершённые реализации: в одной реализации используются два взаимосвязанных списка, а в другой — бинарные очереди.

Пример выполнения клиентской программы, использующей очередь по приоритетам, зависит не только от кличей, но также и от смеси различных операций. Не следует забывать, что даже простое репликация, поскольку во многих различных ситуациях они довольно часто оказываются эффективнее более сложных методов. Например, репликация, работающая с неупорядоченными списками, может оказаться приемлемой в приложениях, в которых выполняются лишь некоторые операции для более требовательных в то же время очень большое количество вставок, тогда как упорядоченный список лучше подходит в тех случаях, когда выполняется большое число операций на поиск наибольшего либо когда вставляемые элементы преимущественно больше уже находящихся в очереди по приоритетам.

Упражнения

- 9.5 Найдите недостатки следующего плана: почему бы для реализации операции `find-the-maximum` (find the maximum) не отбрасывать максимумное значение из числа элементов, вычисленных на текущий момент, а затем возмещать это значение как результат операции?

- 9.6. Показать содержимое массива после выполнения последовательности операций рис. 9.1.
- 9.7. Написать алгоритм сортировки интерфейса очереди по приоритетам, который использует упорядоченный массив в качестве базовой структуры данных.
- 9.8. Написать программную реализацию сортировки очереди по приоритетам, которая использует неупорядоченный связный список в качестве базовой структуры данных (см. см. программу 9.8 и 4.14).
- 9.9. Написать программную реализацию сортировки очереди по приоритетам, которая использует упорядоченный связный список в качестве базовой структуры данных (см. см. программу 9.9).
- 9.10. Рассмотреть "ленивую" реализацию, в условиях которой список упорядочен не только тем, что выполняет операции удаления и добавления элементов. Вставки, сделанные в моменты предыдущей сортировки, содержатся в отдельном списке, этот список сортируется и при необходимости вставляется обратно. Рассмотреть преимущества такой реализации через элементарные реализации, в основе которых лежат неупорядоченные и упорядоченные списки.
- 9.11. Написать клиентскую программу-драйвер, которая использует функцию `insert` для заполнения очереди по приоритетам, затем функцию `delete` для удаления элементов очереди, затем снова `insert` для заполнения очереди, после чего для удаления всех элементов вызывает `delete`. А это всё представляется наглядно: по 100 случайным последовательностям ключей различной длины, значениями которых являются числа. Кроме того, программа должна измерять время, затраченное на каждое выполнение программы, и распечатать или строить график среднего времени выполнения.
- 9.12. Написать клиентскую программу, представляющую собой драйвер программы 9.11, которая использует функцию `insert` для заполнения очереди по приоритетам, затем вызывает функции `delete` и `insert` столько раз, сколько она способна выполнить в течение 1 секунды на случайных последовательностях ключей различной длины, значения которых колеблются в широких пределах, от малых до больших. Программа должна измерять время, затраченное на каждое выполнение программы, и распечатать или строить график среднего числа функций `delete`, которое удалось выполнить.
- 9.13. Воспользоваться клиентской программой из упражнения 9.12, чтобы сравнить реализацию неупорядоченного массива, представленную программой 9.2, с реализацией неупорядоченного списка по упражнению 9.8.
- 9.14. Воспользоваться клиентской программой из упражнения 9.12, чтобы сравнить реализацию упорядоченного массива с реализацией упорядоченного списка по упражнениям 9.7 и 9.9.
- 9.15. Написать клиентскую программу-драйвер, которая использует функции интерфейса очереди по приоритетам из программы 9.1 в различных и разнообразных случаях, которые могут встретиться в практических реализациях. В некоторых случаях список может уже упорядоченный, может, наоборот, быть в обратном порядке, список, когда все ключи одинаковы и когда последовательности ключей содержат только два различных значения.

9.16. (Задача упражненных на своем месте) 24 упражнения.) Докажите правильность гипотез, установленных для 4 элементарных рекурсий, приведенных в таблице 9.1, используя для доказательства леммы индукции операции *find* и *reduce* (необходимые леммы *find* и *reduce* из упражнения 9.2, и результаты, полученные в упражнениях 9.7–9.9), а также формулы рекурсии методов решения других операций. Что касается операции *change* (метод), то можно предположить (ссылка *reduce*) и обосновать (ссылка), предположить, что *change* — средство, обеспечивающее прямой доступ к объекту *data*.

9.2. Пирамидальная структура данных

Основной темой является анализ структуры данных, позволяющей эффективно сортировать деревья. Алгоритм может эффективно поддерживать операции в порядке их приоритета. В сортирующем дереве каждый узел имеет в виде массива своих узлов, что позволяет эффективно принимать значение больше, чем значение любых других узлов, имеющих относительно него строго определенное положение. В свою очередь, каждый из этих узлов должен быть больше, чем все другие определенные узлы и т.д. Подобные упрощения легко обнаружить, если рассмотреть эти ключи как элементы в бинарной префиксной структуре с ребрами, соединяющими от каждого ключа к тем другим ключам, в которых имеет место, что они являются его по значению.



1	2	3	4	5	6	7	8	9	10	11	12
3	-	3	C	S	K	H	A	E	D	A	

Определение 9.2. Зеро называется паразитным элементом матрицы (или вектора), если оно находится в строке или столбце, состоящем из нулей (или в строке или столбце, состоящем из нулей и единичек). Элементарные операции над матрицей (или вектором) называются паразитарными, если они не изменяют ранга матрицы (или вектора).

Лемма 9.1. *На схеме из леммы 9.0.1 заданы функции μ_1, μ_2 и μ_3 . Тогда для любых μ_1, μ_2, μ_3 справедливы следующие утверждения:*

На табеле дерева можно отметить следующие особенности. Абсолютные значения максимальной скорости роста. Однако наиболее удобно пользоваться одним показателем дерева — коэффициентом $\frac{1}{100} \frac{V}{S}$. Из опыта я известно, что мы можем начертить такую таблицу. Показатель в верхней части таблицы: кривой роста, а затем показателем или постройте и пламенираем слева направо, показанные в каждом конкретном году предыдущего уровня для роста текущего уровня до тех пор, пока не будет достигнута высота дерева. Действие

РИСУНОК 9.2. ПРЕДСТАВЛЕНИЕ
ПОЛНОГО ДВОИЧНОГО ДЕРЕЗА В ВИДЕ
ПИРАМИДАЛЬНО УПОРЯДОВАННОГО
МАССИВА

[illegible]

что просто представить полное бинарное дерево в виде массива, используя корневой узел в позиции 1, его левый сын в позиции 2 и 3, узлы следующего уровня в позициях 4, 5, 6, 7 и т.д., как показано на рис. 9.2.

Определение 9.3. Структурную формулу есть совокупность узлов с метками, образующих некое упорядоченное множество бинарных деревьев, представляющих в виде дерева

Можно было бы воспользоваться таким представлением при реализации операций над деревьями, но полные деревья предоставляют возможность использовать компактное представление в виде массива, в котором легко переходить с дочерью по узлу к его родителю или к его левому или правому сыну. Поддержка памяти узлом. Родитель узла, находящегося в позиции i , необходимо хранить в массиве $[i/2]$, и, соответственно, два сына узла в позиции i находятся в позициях $2i$ и $2i + 1$. При подобной организации прохождение по такому дереву выполняется быстрее, чем если бы это дерево было реализовано в списке представлений, поскольку в таком случае могут потребоваться скачки дерева, принадлежащие каждому узлу, чтобы иметь возможность переключиться вперед к сыну по дереву (каждый элемент будет иметь один указатель на родителя и один указатель на каждого сына). Полные бинарные деревья представляются в виде массивов, имеющих жесткую структуру, но они обладают достаточной гибкостью, чтобы позволить реализовать эффективные алгоритмы, имеющие регулярные операции по обработке

В разделе 9.2 мы убедились в том, что можно заключаться о содержании дерева для реализации всех операций по обработке по приоритету (по количеству операций обработки) таких объектов, что не требует выполнения операций в худшем случае потребуют линейные время. Все такие реализации функционируют в основном по одному пути в направлении дерева (перемещение от родителя к сыновьям по направлению вниз или от потолка к решетке по направлению вверх, не исключая при этом горизонтальных движения). Как уже было показано в главе 3, все пути в полном дереве, состоящем из N узлов, содержат в себе порядка $\lg N$ узлов; примерно $N/2$ узлов находятся на самом нижнем уровне, $N/4$ узлов — на втором уровне, $N/8$ узлов — на третьем, и т.д. Каждое выходящее из корня ветвь содержит примерно половину узлов, чем предыдущее, и эта же может быть заменена $\lg N$ операциями.

Можно также воспользоваться другим способом представления дерева-вида структуры для разработки эффективных реализаций операций над деревьями по приоритету. В качестве примера рассмотрим полное дерево с тремя уровнями (см. рисунок 9.3), сортировка левосторонней выборкой (см. программу 9.19) и бинарное дерево поиска (см. рисунок 9.7). Как и в случае полных деревьев и списков, здесь из практических причин, побуждающих рассмотреть разные представления, заключается в том, что они позволяют избежать необходимости заранее знать максимальные размеры, определяющие требуемую и обязательную дорожку и способ представления в виде массива. Мы также можем указать, в некоторых ситуациях определение порядка и гибкости, обеспечиваемой связными структурами, при разработке эффективных алгоритмов. Читатели, которые не имеют достаточного опыта использования списка представлений структур, могут найти более подробное описание в книге [2], чтобы научиться безупречным методам реализации даже более важных абстрактных типов данных, таких как стек

основное дерево, которое так имеет до 2^k сыновей (и $2^k - 1$ внуков) деревьев, рассмотренных в упражнениях этой главы и в разделе 9.7. Описанная ниже процедура свертывания структур может применяться до второго члена, поскольку основной темой настоящей главы является сортирующее дерево (представление в виде массива без связей полного бинарного упорядоченного дерева).

Упражнения

- 9.17. Является ли массив, отсортированный в нисходящем порядке, сортирующим деревом?
- 9.18. Наибольший элемент сортирующего дерева должен появиться в позиции 2, а второй по величине элемент должен занять позицию 3 или 4. Представьте собой массив A длиной n сортирующее дерево из 35 элементов, в котором k -й по величине элемент (k может появиться и 1) не может появиться, для $k = 7, 3, 4$ (не предполагая, что все значения попарно различны).
- 9.19. Решить задачу 9.18 для общего значения k как функции от k предельного возможного размера сортирующего дерева.
- 9.20. Решить задачи 9.18 и 9.19 для k -го по величине элемента.

9.3. Алгоритмы для сортирующих деревьев

Все алгоритмы очереди по приоритетам для сортирующих деревьев зависят так или иначе от того, что мы имеем в распоряжении: от способа хранения структуры данных. Если мы имеем доступ к узлам структуры, то мы можем вносить изменения в структуру сортирующего дерева такие изменения, которые гарантируют, что структура сортирующего дерева сохраняется валидной. Этот процесс иногда называют *подкачкой* (reheapify). Возможны два случая. Когда приоритет какого-либо узла увеличивается (или понижается) уровень сортирующего дерева (оказывается тот или иной узел), мы должны двигаться вверх по дереву, чтобы восстановить структуру сортирующего дерева. Когда приоритеты каких-либо узлов уменьшаются (понижаются, при изменении корневого узла новым узлом), необходимо пройти вниз по дереву, чтобы восстановить структуру сортирующего дерева. Для начала объясним, как реализовать две эти базовые функции, а затем подумаем, как их инкапсулировать в различные операции с очередями по приоритетам.

Если свойство сортирующего дерева нарушается из-за того, что один из узлов становится больше своего родительского узла, можно сделать шаг в направлении исправления этого нарушения, обменяв местами этот узел с его родителем. Если же оба узла являются боковыми, то есть оба являются детьми из них — это правый родитель, другой меньше, чем старый родитель, поскольку он был потомком этого узла), но все еще может оказаться больше своего родителя. Мы можем устранить это нарушение аналогичным способом к продвижению вниз по дереву, пока не достигнем узла с минимальным ключом, каковым является корень дерева. Пример описанного процесса показан на рис. 9.3. Программа проста и интуитивна, и ее основе лежит тот факт, что родитель узла, занимающего позицию k в сортирующем дереве, находится в позиции $k / 2$ этого массива. Программа 9.3 представляет собой реализацию функции, которая устанавливает возможные нарушения, обусловленные

увеличением приоритета и удалением узла сортирующего дерева. Благодаря продолжению работы дерева.

Программа 9.3. Восходящая установка структуры сортирующего дерева

Чтобы восстановить нисходящую структуру после того, как приоритет одного из узлов сортирующего дерева уменьшен, мы продолжим запуск дерева, начиная при максимальной степени узла в позиции k его родителя (который находится в позиции $k/2$) в направлении этого родителя до тех пор, пока не окажется узел $a[k/2] < a[k]$ или пока не выйдет в вершину сортирующего дерева.

```
template <class T>
void fixUp(T& a[], int k)
{
    while (k > 1 && a[k/2] < a[k])
        swap(a[k], a[k/2]); k = k/2;
}
```

Если свойство пирамидальности сортирующего дерева было нарушено в силу того, что какой-либо узел стал меньше, чем один или оба ключа его родителя, выполняются те же операции установки и нарушения путем замены узла на больший из его двух потомков. Таким образом можно вызвать нарушение любого сортирующего дерева на уровне потомков: это нарушение устраняется полностью путем и выполняется продвижение вниз по дереву до тех пор, пока не будет достигнут узел, оба ключа которого меньше его самого, либо нижний уровень дерева. Процесс показан на рис. 9.4. Обратите также внимание на обязательство, что потомки узла сортирующего дерева в позиции k являются в тех позициях $2k$ и $2k+1$.

Программа 9.4 содержит реализацию функции, которая восстанавливает сортирующее дерево после возможного нарушения из-за приращенного приоритета заданного узла, происходящего вниз по этому дереву. Эта функция должна иметь размер сортирующего дерева (N), чтобы иметь возможность определить момент, когда будет достигнут нижний уровень дерева.

Обе эти операции не зависят от способа представления структуры дерева. Они применимы как к результатам ассоциативного метода, так и к потомкам (нисходящий метод) любого узла. В случае восходящего метода мы начинаем с верхнего дерева, начиная с того заданного узла, ключем его родителя до тех пор, пока не выйдет на корневой узел или не достигнем с большим (или равным) ключом. В случае нисходящего

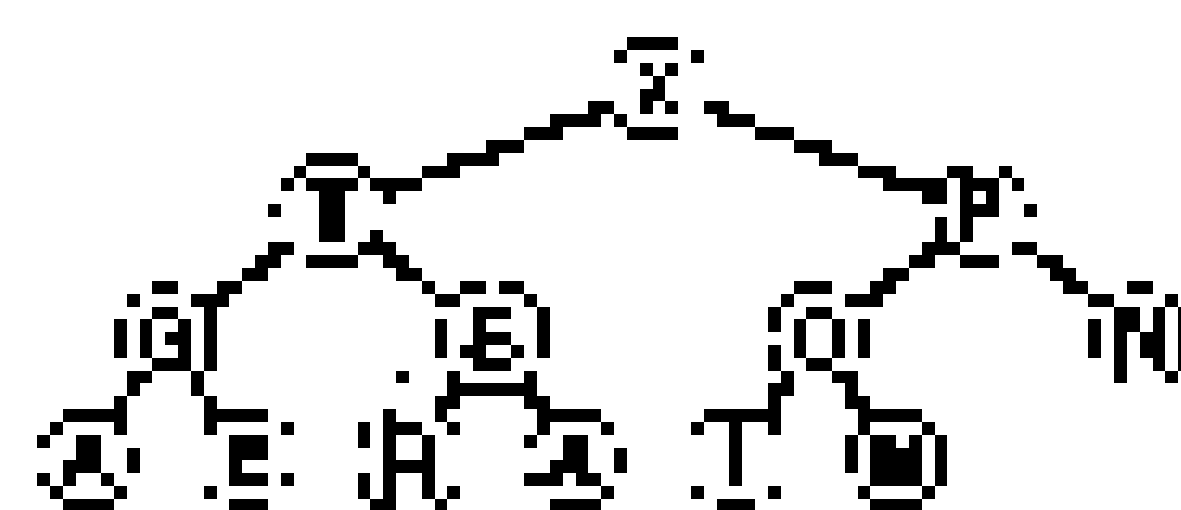
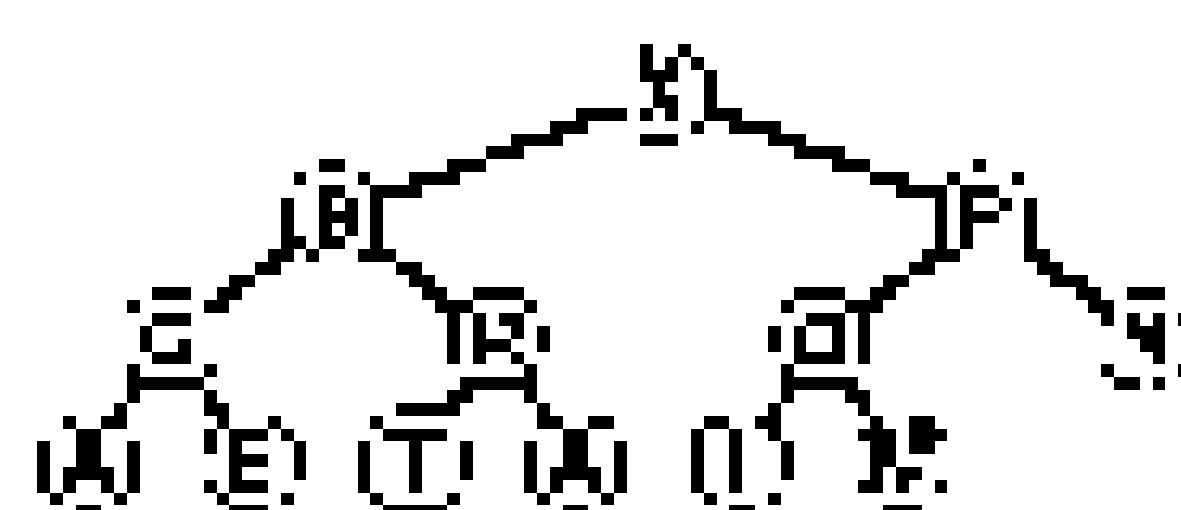


РИСУНОК 9.3. ВОСХОДЯЩАЯ УСТАНОВКА СТРУКТУРЫ СОРТИРУЮЩЕГО ДЕРЕВА

На данном дереве, показанном на рисунке, нарушено свойство пирамидальности, поскольку узел 3 меньше, чем оба ключа его родителя. Если мы начнем с узла 3 и его родителя, дерево останется нарушенным, так как узел 6 больше, чем узел 10, тогда как узел 5 больше, чем узел 8. Если мы начнем с узла 5 и его родителя, дерево останется нарушенным, так как узел 3 больше, чем узел 10, тогда как узел 6 больше, чем узел 8. Если мы начнем с узла 6 и его родителя, дерево останется нарушенным, так как узел 3 больше, чем узел 10, тогда как узел 5 больше, чем узел 8. Если мы начнем с узла 7 и его родителя, дерево останется нарушенным, так как узел 3 больше, чем узел 10, тогда как узел 5 больше, чем узел 8. Если мы начнем с узла 9 и его родителя, дерево останется нарушенным, так как узел 3 больше, чем узел 10, тогда как узел 5 больше, чем узел 8. Если мы начнем с узла 10 и его родителя, дерево останется нарушенным, так как узел 3 больше, чем узел 10, тогда как узел 5 больше, чем узел 8.

метода не переключаются типа по дереву. Там же
нижний заданный узел большим из ключей до
петли до тех пор, пока не достигнем нужного
уровня или точки, в которой не осталось в
большинстве ключей. В абстрактном виде упомя-
нутые операции приписаны не только к бы-
стродействующим деревьям, но и к любой древовидной
структуре. Рассмотренные алгоритмы, на-
меченные операции по приоритетам, обыч-
но имеют древовидную древовидную структуру
область видна, но в то же время показаны плавы-
по из этих базовых операций, обеспечивающие
доступ к наибольшему ключу этой структуры,
используемым в ее верхней части.

Если представить себе, что некоторое сортирующее дерево представляет историю некой корпорации, в которой каждый патрик представляет подчиненное подразделение (и каждый родник — представителю той же вышестоящее подразделение), то эти операции допускают любопытные аналогии. Вошедший в лес соответствует пополнению ты ещѣ одного перспективного управленца, который предлагается по служебной лестнице в одного места на другой, более пышный (обмениваясь служебным функциями — чтобы не — делая, но при этом более низкую квалификацию) до тех пор, пока этот самый работник не столкнется с более квалифицированным специалистом. Нынешний метод андершифта, снугшн, когда президент некоторой компании смеет на его посту некто, уступающий ему по квалификации. Если какой-либо из подчиненных президента, облеченный полномочиями полководца, предпринимает попытку качества галлопа разбития, он обязан вернуться своим обязанностям, и мы видим, что вышестоящий по служебной лестнице, помня о должности выше работника и вышестоящих других, пока не будет достигнут уровень компетенции нового работника, когда не остается ни одного галлопа, превосходящего нового работника по квалификации (это не обязательно сценарий в реальной жизни встречается редко). Продолжая эту аналогию, движение по более пышному лесу может быть связано с квалификацией.

Программа 9.4. Нисходящая установка структуры сортирующего дерева

ЧТОБ. ВОССТАНОВИТЬ ПЕРВИЧАЛЬНУЮ СТРУКТУРУ В СЛУЧАЕ, КОГДА ПОДРОБНЕЕ УЗНА-
НА ПЕРВАЯ СЕРИЯ ДВИЖЕНИЯ ИЛИ ПОСЛЕДОВАТЕЛЬНОСТЬ ДВИЖЕНИЙ, НЕКОТОРЫЕ ПРИ НЕОБХОДИМОСТИ
НЕКОТОРЫЕ УЗЛЫ В ПОСЛЕДНИХ С БОЛЬШИМ КОЛИЧЕСТВОМ ПУТЕЙ, И ОСТАНАВЛИВАЮТСЯ, КОГДА
УЗЕЛ В ПУТЕИ ИЛИ ПУТЕИ ЯВЛЯЮТСЯ НЕОБХОДИМЫМИ ДРУЖИМ СЛУЧАЯМ ИЛИ СЛУЧАЯМ, НАПР. СЛУЧАЕ

Как уже отмечалось ранее и как продемонстрировано в программе 9.6, любую очередь по приоритетам можно использовать для построения еще одного метода сортировки. Мы просто вставляем все подлежащие сортировке элементы в очередь по приоритетам, а затем многократно используем операцию извлечения минимума, чтобы вывести их в порядке убывания приоритетов. Такое использование очереди по приоритетам, представленной в виде неупорядоченного списка, соответствует непосредственно сортировке выбором, а программно упорядоченно в списке соответствует выстраиванию сортировки выбором.

Программа 9.6. Сортировка с использованием очереди по приоритетам

Чтобы отсортировать массивы $a[1], \dots, a[n]$, используя для этой цели АТД, очередь по приоритетам, существует три основные функции: insert (вставить), delete (элемент с заданым приоритетом x удалить) и getmin (найти минимальный элемент и в порядке убывания значений приоритетов). Под $O(n \log n)$ подразумевается время $\sim c \log n$ за время, требуемое для $N \log N$, но при этом он требует $O(n \log n)$ для хранения элементов, требуемых для хранения элементов N (в очереди по приоритетам).

```

#include "PQ.h"
using namespace PQ;
void SortPQ(T* a, int n, int r)
{
    int k;
    PQQueue pq(1);
    for (k = 1; k <= n; k++) pq.insert(a[k]);
    for (k = n; k >= 1, k--)
        a[k] = pq.delete();
}

```

На рис. 9.5 и 9.6, показан пример первой фазы процесса построения), из которой используется реализация очереди по приоритетам с приоритетными ключами; на рис. 9.7 и 9.8 представлена вторая фаза (которую будем называть нисходящей сортировкой — sortdown). В условиях практической реализации этот метод не является особенно эффективным, поскольку он без особой необходимости копирует сортируемые элементы (в очередь по приоритетам). Действительно, выполняя N последовательных вставок

не самый эффективный способ построения сортируемого дерева, состоящего из N элементов. В следующем разделе мы рассмотрим различные классические алгоритмы нисходящей сортировки для двух классов деревьев, указанных ранее.

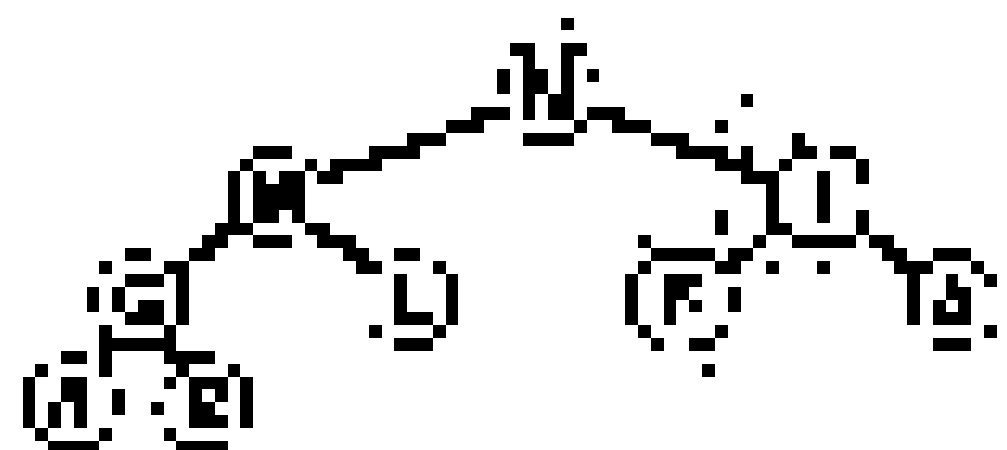
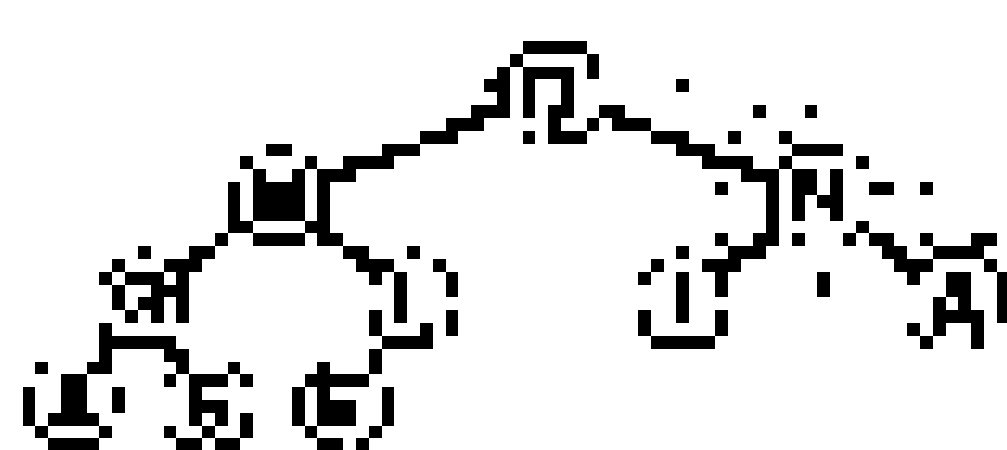
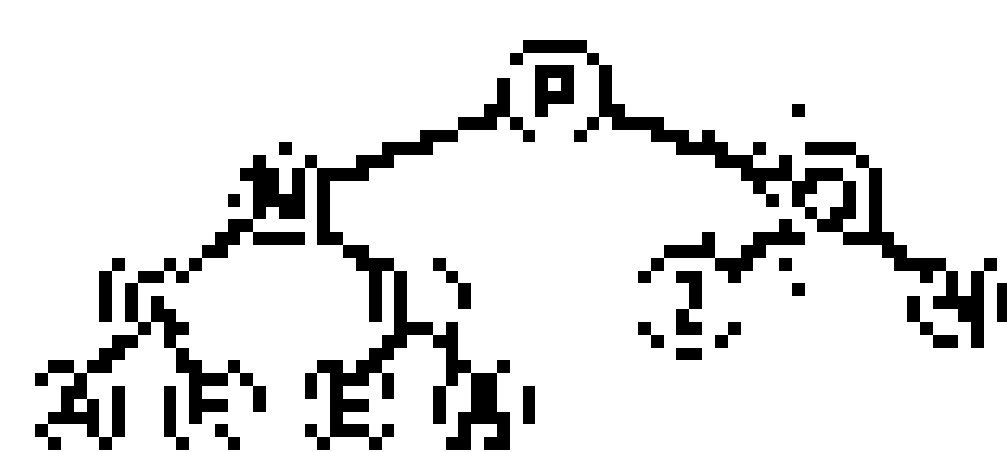
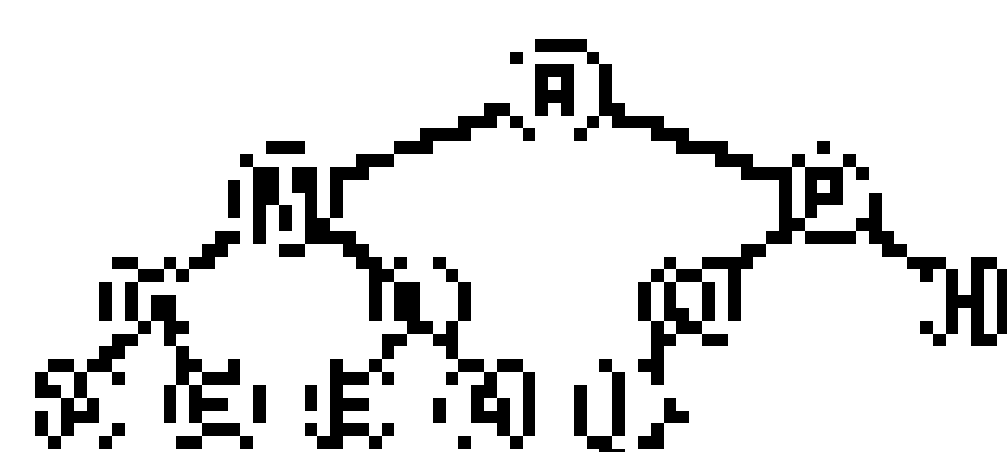
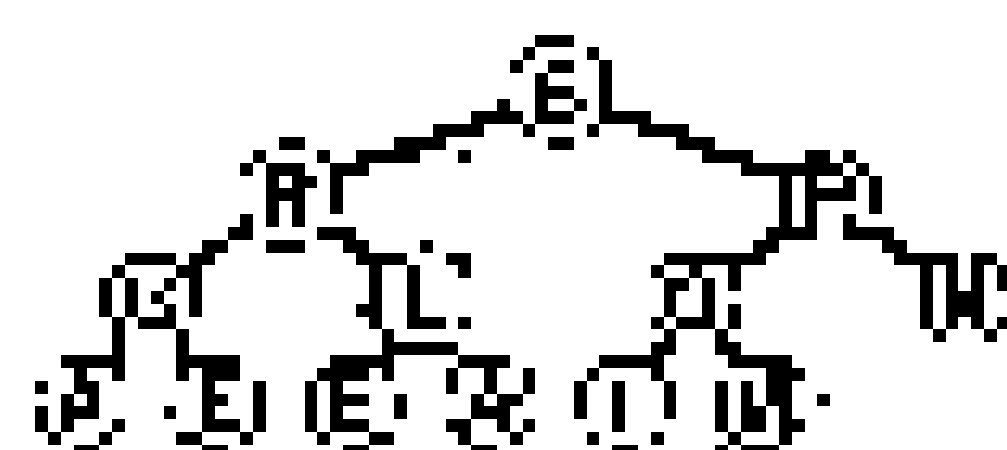
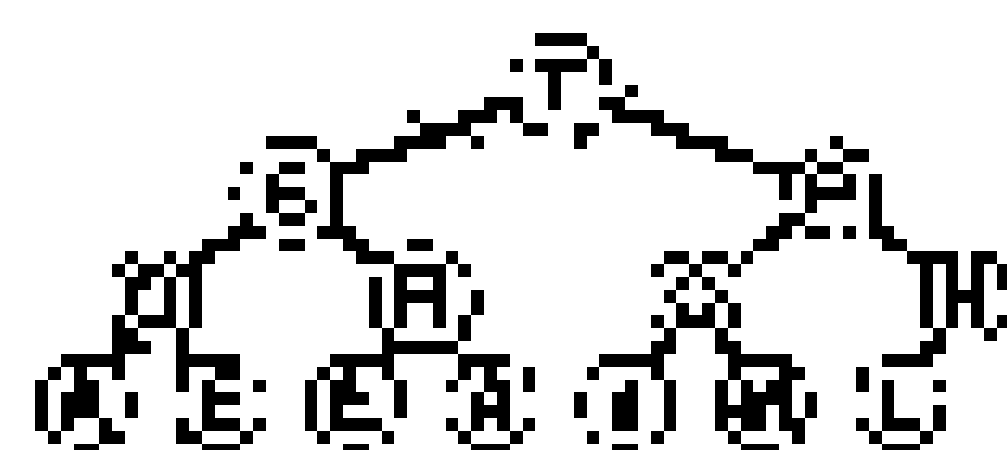
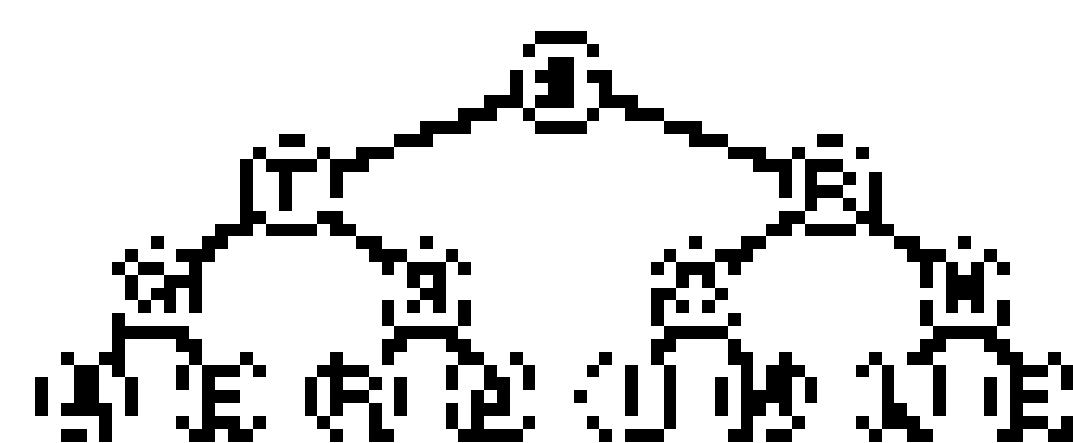


РИСУНОК 9.7. СОРТИРОВКА В СОРТИРУЮЩЕМ ДЕРЕВЕ

После завершения нисходящей сортировки сортирующее дерево имеет структуру, показанную на рисунке. Минимум находится в левом нижнем углу дерева. По пути из корня до минимума, дерево

Упражнения

9.11. Построить сбалансированное дерево, которое получится после того как ключи E A S Y Q U E S T I O N вставлены в первоначально пустое сбалансированное дерево.

9.12. Напишите алгоритм сортировки, пригодный в упражнении 9.1, представляющий последовательность сбалансированного дерева, полученная в результате выполнения операций

PRIOR**I**T**Y***QUER**Y**R

на первоначально пустом сбалансированном дереве.

9.13. Поскольку принятие вхсх минимизируется и операция по удалению из сбалансированного дерева, элементы загружаются и запоминаются в порядке, чем это необходимо. Предложите более эффективные реализации, которые не сталкиваются с такой проблемой, в духе сортировки вставками.

9.14. Почему мы не используем сигнальную метку, чтобы избежать проверки $j < N$ в функции `PushDown`?

9.15. Добавьте очередь элементов минимального дерева для каждого из реализованных очередей по приоритету с сбалансированным деревом в программе 9.5. Рассмотреть случаи, когда добавляемое значение больше всех оставшихся значений в очереди. Составьте доказательство рекурсивно приводит к добавлению элемента $pr[M]$.

9.16. Каким является минимальное количество перемещений ключей, которое требуется выполнить на сбалансированном дереве в операции удаления минимального (максимального)? Приведите пример сбалансированного дерева, содержащего 15 элементов, для которого достигается этот минимум.

9.17. Каким является минимальное количество ключей, которое необходимо переместить в процессе выполнения на сбалансированном дереве трех последовательных операций удаления минимального? Приведите пример сбалансированного дерева из 15 элементов, для которого достигается этот минимум.

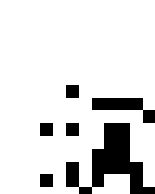
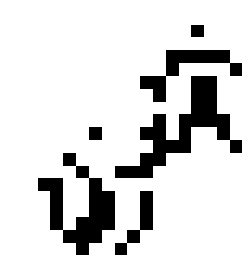
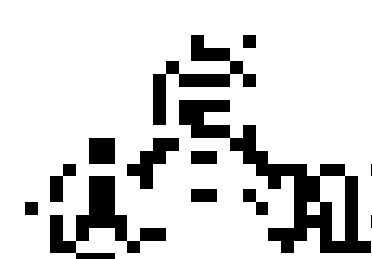
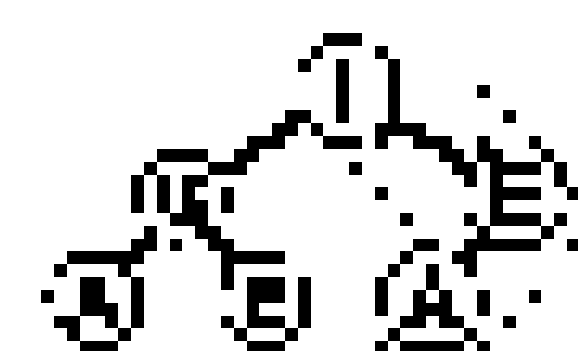
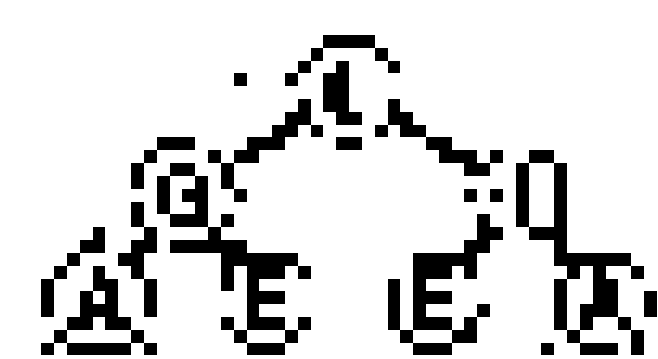
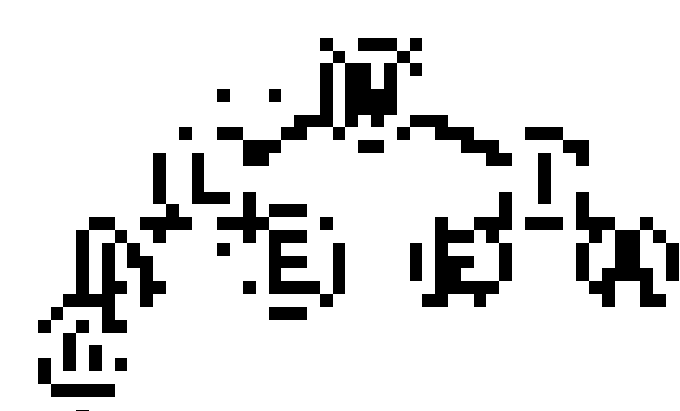


РИСУНОК 9.6 СОРТИРОВКА ИЗ СОРТИРУЮЩЕГО ДЕРЕВА

Предположим, что мы имеем сбалансированное дерево, которое содержит 15 элементов. Мы хотим удалить минимальный элемент из этого дерева. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 14 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 13 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 12 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 11 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 10 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 9 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 8 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 7 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 6 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 5 элементов. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 4 элемента. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 3 элемента. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 2 элемента. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем дерево, которое содержит 1 элемент. Мы знаем, что минимальный элемент находится в левом поддереве. Мы удаляем его, и получаем пустое дерево.

9.4. Пирамидальная сортировка

Основную идею, показанную в описании программы 9.6, можно приспособить с таким расчетом, чтобы сортировка массива выполнялась без необходимости использования какого бы то ни было дополнительного пространства памяти. Другими словами, создавая очередь не являю сортировки, мы отказываемся от идеи очереди очереди по приоритетам, и вместо того чтобы поддерживать очередь, являющуюся интерфейсом АТД очереди по приоритетам, будем непосредственно применять функции `fixUp` и `fixDown`.

Исполняя программу 9.6 в программе 9.6 непосредственно для перемещения по массиву списка узлов, применяя функцию `fixUp` с тем, чтобы элементы, расположенные слева от указателя просмотра, образовывали пирамидальное упорядоченное полное дерево. Затем, во время выполнения процесса нисходящей сортировки мы перемещаем каждый элемент на то место, которое остается пустым по мере уменьшения размеров сортируемого дерева. Иначе говоря, процесс нисходящей сортировки по сути является обратным, однако при этом он является более эффективным методом обнаружения наибольшего элемента и частичной сортировки части массива.

Более эффективно, нежели построение сортирующего дерева за счет последовательного выполнения шагов, как показано на рис. 9.5 и 9.6, является построение такого дерева методом прохода снизу вверх, т.е. в обратном направлении, формируя поддеревья меньших размеров снизу вверх, что иллюстрирует рис. 9.9. Другими словами, мы рассматриваем каждую позицию массива как узел небольшого поддерева и применяем функцию `fixUp` к каждому из них. Функция `fixDown` работает на таких сортируемых поддеревьях столь же хорошо, как и на большем дереве. Если оба подхода к созданию сортирующего дерева, т.е. выходя для того, что функция `fixDown` приводит к тому, что поддерево с корнем в этом узле также становится сортирующим деревом. Пройдясь по дереву в обратном направлении и применяя `fixDown` в каждом узле, мы по существу можем построить отсортированный массив. Проход снизу вверх не нужен из-за обратного направления цикла массива, поскольку можно представить процесс алгоритма 1.

Понимая различия классического алгоритма пирамидальной сортировки (независимо от представления и программы

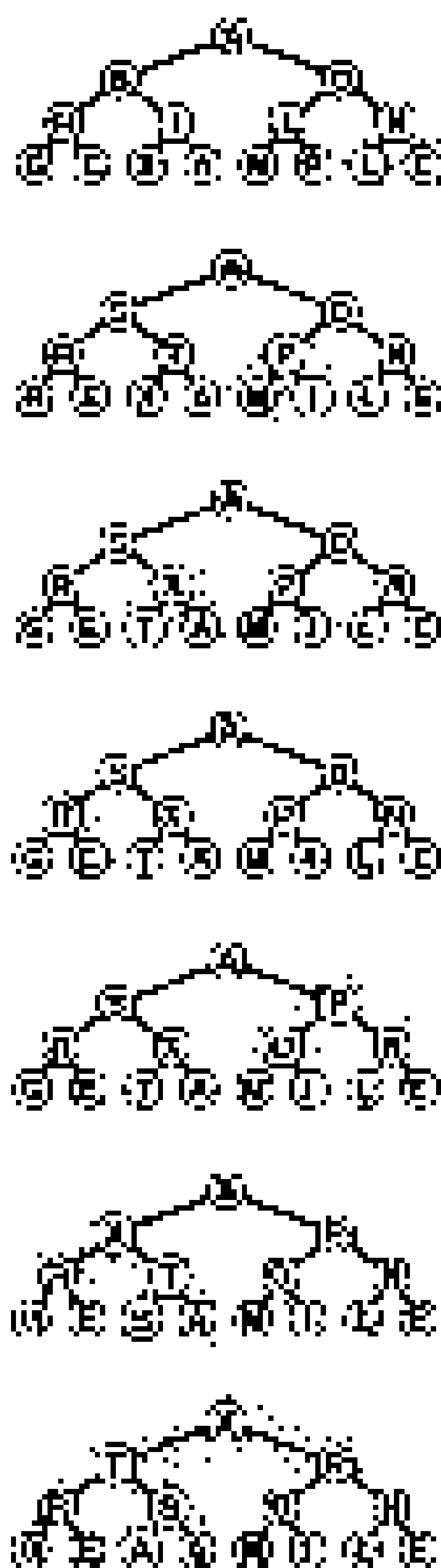


РИСУНОК 9.9. ПОСТРОЕНИЕ СОРТИРУЮЩЕГО ДЕРЕВА СНИЗУ ВВЕРХ

Примечание: сортировка снизу вверх не требует использования дополнительной памяти, которая необходима для хранения элементов, подлежащих сортировке. Однако сортировка снизу вверх требует использования дополнительной памяти для хранения элементов, подлежащих сортировке. Однако сортировка снизу вверх не требует использования дополнительной памяти, которая необходима для хранения элементов, подлежащих сортировке.

9.7. И хотя ссылки в этой программе на первый корень дерева совершенно разные (первый выполняет построение сортирующего дерева, второй разрушает это сортирующее дерево для продолжения работы сортировки), они построены на основе одной и той же базовой процедуры, которая возмещает порядок в дереве. III корнем. Возможно, уже установилась привычка к такому исключению, но разница между корнем, используя при этом представление полного дерева в виде массива. На рис. 9.10 показано содержание массива и пример, соответствующий рис. 9.7–9.9.

Задание 9.4. Для построения сортирующего дерева использовать метод сортировки пирамиды.

В основе этого утверждения лежит тот факт, что большинство подпрограмм обратные деревья имеют наибольшее разность. Например, чтобы построить сортирующее дерево из 120 элементов, мы выполняем построение 32 сортирующего дерева размером 3. 16 деревьев размером 3. 5 деревьев размером 15. 4 дерева размером 31. двух сортирующего дерева размером 63 и одного дерева размером 127, так что в худшем случае требуется $32 \cdot 1 + 16 \cdot 3 + 5 \cdot 15 + 4 \cdot 31 + 2 \cdot 63 + 1 \cdot 127 = 120$ умножений (и следовательно больше чисел для оценки сравнения). Для $N = 2^n - 1$ в общем случае числа продолжения ряда

$$\sum_{i=0}^{n-1} (2^{i-1} - 1) = 2^n - n - 1 \leq N.$$

Это же доказательство справедливо и в случае, когда $N + 1$ не является степенью 2.

Что важно не имеет особого значения для пирамидальной сортировки, поскольку основная идея состоит в выполнении все еще приходится на $N \log_2 N$ — время, затрачиваемое на выполнение нисходящей сортировки. Однако оно играет важную роль в тех приложениях, связанных с приоритетами, в которых операция выбора (или более) приводит к алгоритму, обеспечивающему линейную зависимость времени выполнения. Как отмечалось на рис. 9.6, построение сортирующего дерева с N последовательно выполняемых операций требует в совокупности $N \log_2 N$ шагов в худшем случае (даже если это общее количество шагов выполняется в среднем линейным для функций с приоритетной сортировкой).

```

В К О П Т И Н С Е Р А М П Л Е
М О Н И Т И Н Г О В А М П Л
А М О Н И Т М П Л О М П Л
А С П Р К С О Е Г А М П Л
А М О Н Т М П Л Р Г А М П Л
А М О Н Т О С О Е Г А М П Л
А М Т Р О С О Е Г А М П Л
С Т Р П С П Р С Р А Р П Л

```

```

Т О Р П С О С Г А Р П Л
С Р П Л С О С Г А М П Л
Н Л Р П С М П Р С М П Л
Р Л О С М П О С А М П Л
П Л Н П М П Р А Р М П Л
Н Л М П Р А М П О М П Л
М Л С П С А М П О С А М П Л
П Р О Р А М П М П М П Л
П О Л А Л М О М П О Р М П Л
О С С А А П С М П О М П Л
Р А Р А П П М П Р М П Л
С А А С О С М П О Р Р О Р
М А Р Р О С М П М П С М П
Р А Р Р М П М П М П Р М П

```

РИСУНОК 9.10. ПРИМЕР ПИРАМИДАЛЬНОЙ СОРТИРОВКИ

Пирамидальная сортировка представляет собой эффективный алгоритм сортировки, основанный на методе выбора. Сортирующее сортирующее дерево строится для использования сортирующей операции. Другим важным шагом алгоритма является сортировка рис. 9.9. Далее, от дерева сортирующей сортировки выбирается элемент. Если сортирующее дерево строится для сортировки массива, то сортирующее дерево строится для сортировки массива. Если сортирующее дерево строится для сортировки массива, то сортирующее дерево строится для сортировки массива. Если сортирующее дерево строится для сортировки массива, то сортирующее дерево строится для сортировки массива.

Программа 9.7. Пирамидальная сортировка

Непосредственно применение функции `fixDown` превращает массив в классическое бинарное пирамидальное дерево. Если `fixDown` выданное дерево превратит в дерево; далее, пока `while` не достигнет минимального элемента в последнем элементе массива, вызываемых `fixDown` в `fixDown` до тех пор, пока получаемое дерево не станет пустым. Тот факт, что в условиях представления полного дерева в массиве указатель `pq` указывает на `a[1-1]` позволяет программе избежать сложности, связанной с тем, что указатель `pq` указывает на элемент с индексом 1 (см. рис. 9.2). В некоторых средах программирования это можно избежать.

```
template <class Item>
void fixDown(Item a[], int i, int n)
{ int k, M = i+1;
  Item *pq = a[i-1];
  for (k = M/2; k <= i; k--);
  while (k < n)
    (swap(pq[i], pq[M]));
    fixDown (pq, i, --n);
}
```

Лемма 9.5. Пирамидальная сортировка выполняется менее $2N \lg N$ операций для массива из N элементов.

Несколько более сложная граница $2N \lg N$ операций непосредственно из леммы 9.2. Предлагаемая здесь граница следует из более точного подсчета по лемме 9.1.

Лемма 9.5 и возможность выполнения без использования дополнительной памяти — вот два основных фактора, которые обуславливают интерес, проявляемый к пирамидальной сортировке на практике. Если вы хотите знать, что сортировка N элементов будет выполняться за время, пропорциональное $N \lg N$ независимо от природы входных данных, то вы не можете не знать, что пирамидальная сортировка — это один из самых эффективных алгоритмов сортировки (в отличие от быстрой сортировки). А пирамидальная сортировка особенно не занимает дополнительного пространства памяти (в отличие от сортировки слиянием). Единственным недостатком пирамидальной сортировки является то, что она требует много памяти. Например, внутренний шаг расширения алгоритма (стоимость выражается количеством операций) требует больше операций, чем внутренний шаг быстрой сортировки, тем не менее, пирамидальная сортировка, по сравнению с быстрой сортировкой, как правило, быстрее, так как она работает с массивом, а не с файлом с произвольной организацией.

Сортировка также можно успешно использовать для решения проблемы выбора k максимальных элементов из N элементов (см. главу 7). Особенно в случае, когда k мало. Мы просто проводим выходы массива пирамидальной сортировки после того, как k элементов будут отобраны из вершины сортирующего дерева.

Лемма 9.6. Быстро на базе сортирующей структуры можно найти k -й минимальный элемент из N элементов за время, пропорциональное k , когда k мало по сравнению со N , либо за время, пропорциональное $\log N$, во всех других случаях.

Один метод заключается в том, чтобы построить сортирующее дерево с предполагаемым числом операций сравнения меньшего $2N$ (см. лемму 9.4), с наименьшим количеством k и наибольшим количеством, необходимым для этой цели $2k \log N$ или меньшим количеством операций сравнения (см. лемму 9.4), при общем числе элементов, равном $2N + 2k \log N$. Другой метод связан с построением минимально ориентированного сортирующего дерева размером k с последующим выполнением операции удаления максимальный (либо же минимальный) элемент операции поиска операции удаления минимального (либо же максимального) элемента, при этом количество операций сравнения, не превышающее $2k + 2(\log N - k) \log k$ (см. выражение 9.36). Этот метод использует дополнительные знания, требующие время k , в котором особо эффективны для нахождения k минимальных из N элементов в узлах, когда k принимает малое, а N — большое значение (для этого не требуется знания). Что касается случайных случаев или других типовых ситуаций первая теорема (для операций на сортирующем дереве во втором методе, см. выше), есть $O(k)$, когда k мало по сравнению со N (см. выражение 9.36).

Исследованы другие способы дальнейшего улучшения пирамидальной сортировки. Одна из них, развитая Флойдом, состоит в использовании того обстоятельства, что элемент, повторно поступающий в процесс поиска минимальной сортировки, продолжает входить на начальной стадии, так что мы можем сэкономить время, затрачиваемое на проверку множества таким элементом своей позиции, просто сравнивая больший из двух потоков до тех пор, пока не будет достигнут нижний уровень с последующим продвижением вверх по сортирующему дереву в соответствующую позицию. Благодаря этой идее минимизирует количество чисел выполняемых операций сравнения асимптотически стремится к 2 и принимает значение, близкое к $2N' = N' \log N' - N' / \log 2$, где N' является абсолютным минимальным количеством сравнений для любого алгоритма сортировки (см. часть 8). Этот метод требует дополнительных знаний и может оказаться полезным и в практике только в том случае, когда расходы на операции сравнения сравнительно велики (например, при сортировке записей со строковыми ключами или другими длинными ключами).

Другая идея заключается в том, чтобы построить сортирующее дерево, опираясь на представление полного пирамидально упорядоченных листьев дерева в виде массива, при этом узел в позиции k больше или равен узлам в позициях $2k - 1$, $2k$ и $2k + 1$ и наоборот для узлов в позициях $\lfloor (k - 1) / 2 \rfloor$ для всех позиций между 1 и N в массиве из N элементов. Сложная сложность, обусловленная меньшей высотой дерева, уменьшается более высокой степенью сложности поиска минимума на трех потоках в каждом узле. Подобного рода компромиссы известны от других решений (см. выражение 9.30). Дальнейшее увеличение количества потоков в каждом узле во всем приводит к отсутствию никакой выгоды.

Рисунок 9.11 иллюстрирует пирамидальную сортировку в действие на файле с произвольной организацией. Сначала кажется, что этот процесс делает все, что нужно, только не сортировку, поскольку по мере продвижения построенная сортирующая

деревянные элементы переключаются в начало файла. Однако, в соответствии с указанным, даже если бы было тысячи как перемещение отбрасывание сортировки элементов. На рис. 9.12 показано, что различные виды данных можно представить в виде сортировки дерева, обходящегося сбалансированными, однако по мере повышения тронесса сортировки к сплону завершению все они приспособлены для сортировки дерева с произвольной организацией.

Нестабильно, что может не интересоваться проблемой, какой метод сортировки выбрать для конкретного приложения: пирамидальную сортировку, быструю сортировку или сортировку слиянием. Выбор между пирамидальной сортировкой и сортировкой слиянием сводится к выбору между простой сортировкой (см. упражнение 9.18) и сортировкой, которая использует дополнительный объем памяти; выбор между пирамидальной сортировкой и быстрой сортировкой сводится к выбору между средним быстродействием сортировки и быстрой сортировкой, обеспечиваемым в среднем случае. В свое время мы достаточно хорошо потрудились над совершенствованием внутреннего анализа быстрой сортировки и сортировки слиянием; решение этих вопросов организовано с помощью анализа данных, представленных в упражнении, сопровождающих следующие главы. Речь идет не о том, о каком-либо конкретном примере пирамидальной сортировки до уровня, превосходящего быструю сортировку, что, собственно говоря, показано посредством динамических данных, представленных в табл. 9.1. Так же как и слиянием, изучающие проблему быстрой сортировки на собственных машинах, найдут эти данные полезными. Как обычно, различные характеристики конкретных машин и систем программирования могут играть важную роль. Например, быстрая сортировка и сортировка слиянием обладают свойствами неустойчивости, которые для некоторых приложений являются некоторыми типичными недостатками. Когда операции сравнения становятся недостаточными для выполнения, тогда быстрая сортировка лучше других, поскольку в таком случае ситуациями остаются путь по сравнению с сортировкой, если принимать во внимание только время выполнения и стоимость используемого пространства памяти.



РИСУНОК 9.11
ДИНАМИЧЕСКИЕ
ХАРАКТЕРИСТИКИ
ПИРАМИДАЛЬНОЙ
СОРТИРОВКИ
На первом этапе
сортировки, когда процесс
сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100) сортировки (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,

Таблица 9.2. Эмпирические исследования алгоритмов пирамидальной сортировки

Эта таблица представляет собой сводку результатов различных опытов сортировки на случайных данных. Число элементов N равно 10^3 , 10^4 , 10^5 и 10^6 . В таблице даны значения Q (число внутренних циклов), что гарантирует, что пирамидальная сортировка обладает меньшим количеством действий, чем быстрая сортировка. Это объясняется тем, что в быстрой сортировке число внутренних циклов равно $N \log_2 N$. В таблице показаны для каждого N значения, когда быстрая сортировка Дика и в первой части таблицы показывает, что метод Флойда имеет коэффициент усиления усредненной сложности пирамидальной сортировки в тех случаях, когда плотность элементов достаточно высока.

N	32-битовые целые числа					сгруппированные числа		
	Q	M	PQ	H	F	Q	M	F
10000	2	5	4	3	4	3	11	5
25000	7	11	9	7	6	18	25	20
50000	13	24	22	15	15	35	62	48
100000	27	50	47	42	40	58	143	110
200000	51	111	106	100	107			
400000	122	238	245	232	245			
900000	281	570	543	540	565			

Обозначения:

- Q Быстрая сортировка, стандартная реализация (программа 7.1)
- M Сортировка методом «мешочка» (программа 8.1)
- PQ Доработка по продолжительности базис пирамидальной сортировки (программа 9.5)
- H Пирамидальная сортировка с доработкой реализации (программа 9.5)
- F Пирамидальная сортировка с доработкой реализации Флойда

Упражнения

- 9.28. Показать, что пирамидальная сортировка не является устойчивой.
- 9.29. Определить эмпирически путем процентное отношение времени затрат на фазу построения для $N = 10^3$, 10^4 , 10^5 и 10^6 .
- 9.30. Реализовать версию пирамидальной сортировки, основанную на построении пирамиды из управляемых сортируемых деревьев. Рассмотреть, насколько это лучше. Сравнить количество сравнений с сортировкой Флойда с помощью сортировки, полученной эмпирическим путем, с использованием показателя стандартной реализации для $N = 10^3$, 10^4 , 10^5 и 10^6 .
- 9.31. Продолжая упражнение 9.30, определить эмпирически путем, насколько метод Флойда эффективнее применительно к треугольным деревьям.
- 9.32. Пусть в заданном случае станем считать операции сравнения и предполагая, что для нахождения минимального из элементов требуется генерация равенства, найти значение k , которое входит в множитель коэффициента $N^2 \log_2 N$, полученный при подсчете операций сравнения, в том случае, когда в пирамидальной сортировке используется список сортируемых данных. Сначала воспользуемся прямым обобщением программы 9.7, затем предположим, что метод Флойда подпадает под категорию одно сравнения во внутреннем цикле.

находит применение в ситуации, когда элементная программа работает с ключами и ассоциируемой с ними информацией. Будучи главным образом заинтересованной в операции доступа к информации, программа обычно в ключах с наибольшим значением, или, тем не менее, может выполнять и другие многочисленные операции по обработке данных по объектам, о чем идет речь в данной главе. Как эти операции обрабатываются конкретный очерёда по приоритетам через дескрипторы (указатели на структуру, которая не имеет конкретного описания). Операция выполняется по запросу дескриптор каждого объекта, добавляемого в очередь по приоритетам в элементной программе. Дескрипторы объектов отличаются от дескрипторов очередей по приоритетам. При такой организации клиентская программа берет на себя обязанность поддерживать дескрипторы, которые впоследствии они могут использовать для определения, какие объекты подлежат обработке со стороны операционной системы и какие могут быть выполнены по приоритетам должны быть выполнены не указанные операции.

Данная организация подходит для реализации как на элементную Программу, так и на реализацию. Клиентская программа не может получить доступ к информации ни каким другим способом, кроме как за счет использования этого интерфейса. Но все вышесказанное ответственность за правильное использование дескрипторов: например, в реализации не существует какой-либо возможности способом проверки с целью выявления некорректности такого действия, как использование элементной программой дескриптора удаленного элемента. С другой стороны, реализация не может свободно передаваться по информации, поскольку у клиентских программ не будет дескрипторы, которые они могут использовать на более поздних этапах. Этот момент станет более понятным, когда мы приступим к рассмотрению деталей реализации. Как обычно, какой бы уровень детализации ни был бы выбран в данной реализации, абстрактный интерфейс, такой как программа 9.8, представляет собой точную отправную точку для нахождения компромисса между потребностями приложения и потребностями реализации.

Применившиеся реализации базовых операций над очередями по приоритетам, которые даны в программе 9.9, используют генерализованные представления данных в виде атомарных элементов. Эти программы не являются универсальными интерфейсами: очень похожи на другие, такие же беспристрастные реализации, используя другие элементные представления.

Программа 9.8. Полный АТД очереди по приоритетам

Данный абстрактный АТД описывает по приоритетам данные структуры, позволяющие удалить элементы и вставить приоритеты (с использованием дескрипторов, представляющих элементы реализации) в данную структуру.

```
typedef struct Item
```

```
{
```

```
    |
```

```
    priority;
```

```
    //Программой вид, зависящий от реализации
```

```
};
```

```
    //Определение дескриптора в зависимости от реализации
```

```
void init();
```

```
void insert() const;
```



```

handle_remove(item);
item_remove();
void change(handle, item);
void move(handle);
void join(PQCHandle);

```

1.

В (44) показано с объектами из раздела 5.3, реализацию, представленную в программах 9.9 и 9.10, подходит для тех приложений, в которых очередь по приоритету необходима в качестве механизма реализации очереди. В остальных случаях более предпочтительными окажутся реализации на базе прямой или косвенной сортировки. Реализация функций `fixup` и `pushdown` на деревьях с помощью сортировки — достаточно сложная задача, которую мы оставляем читателю в качестве упражнения, а сами функции показаны только для иллюстрации в программах 9.6 и 9.7.

Планш 9.10, такой как и программа 9.9, содержит много функций, однако можно было рассмотреть другие функции, реализующие другие операции на кучах: программы и реализации. В разделе 9.6 рассматривается пример, в котором в области памяти программы выделены ресурсы, относящиеся к куче, а программы, работающие с очередями по приоритету, обращаются к ним не явно.

Другая проблема, связанная с тем, что реализация может оказаться неэффективной интерфейсом. Например, может потребоваться функция, возвращающая указатель на элемент с заданным приоритетом в очереди, а не список элементов с этим значением и связанной с ним информацией. Кроме того, на предыдущей планшечке проблемы, которые рассмотрены в разделе 4.3 и которые связаны с управлением памятью и эффективной реализацией. Мы не рассматриваем операции удаления элементов или копирования очереди, а также сами из нескольких реализаций принцип операции `fixdown` (44).

Программа 9.9. Неупорядоченная очередь по приоритету в виде двухсвязного списка.

Эта реализация содержит функции `enqueue` (44) и `dequeue` (45) и реализует интерфейс программы 9.9, а также реализует реализацию `fixdown` (46) и `pushdown` (47). Для реализации функции `enqueue` используется список, в котором хранятся элементы и их приоритеты. Структура `node` (48) описывается как элемент двухсвязного списка (длина и длина списка). Принцип работы программы можно увидеть из кода программы и комментариев к ней.

```

#include <stdio.h>
#define PQ
1
private:
struct node
{
    T item;
    node *prev, *next;
    node *link;
    { item = 0; prev = 0; next = 0; }
};

```



```

typedef struct tnode
{
    int head, tail;
public:
    typedef struct tnode* handle;
    int init = 0;
    {
        head = new node(0); tail = new node(0);
        head->prev = tail; head->next = tail;
        tail->prev = head; tail->next = head;
    }
    int empty() const
    { return head->next->next == head; }
    handle insert(int x)
    { handle t = new node(x);
      t->next = head->next; t->prev = t->next->prev = t;
      t->prev = head; head->next = t;
      return t; }
    ~tnode() {}
    void change(handle, item);
    void remove(handle);
    void join(tnode*);
};

```

Добавление таких процедур в интерфейс, представленный в программе 9.9, не является какой-либо трудностью; гораздо труднее разработать такую реализацию, в рамках которой гарантируется логарифмическая производительность для всех видов операций. В тех приложениях, в которых очереди по приоритетам не достигают больших размеров или в которых существуют операции вставки и удаления, которые требуют специальному состоянию, предпочтительным может оказаться полностью новый интерфейс. С другой стороны, в тех приложениях, в которых очередь необходима для принципиально различных приложений или подтверждения соответствующим расчетам дожития, а также краткое простейшее представление, можно организовать на основе тех элементов, которые обеспечивают упомянутый рост производительности. Такие приложения обширные и разнообразные, результаты которых можно увидеть в списке алгоритмов, реализующих очереди по приоритетам при помощи различных схем организации. Включение этих очередей, описанных в разделе 9.7, представляет собой важный пример такого рода.

Программа 9.10. Очереди по приоритетам в виде двусвязного списка (продолжение)

Представив эти элементы для анализа соответствующих объявлений в программе 9.9, мы получаем программу для очереди по приоритетам. Очередь должна минимизировать доступ к просмотру всего списка, тем не менее затраты на поддержку двусвязного списка оправдываются тем фактом, что операции вставки, удаления и объединения реализованы таким образом, что выполняются за постоянное время, при этом в списке временно только элементы, которые удаляются (обращайтесь к главе 3 за более подробным описанием связанных списков).

При необходимости можно добавить в программу деструктор-конструктор, относящийся к переименованному элементу, для уничтожения и освобождения памяти, связанной с программой с целью получения АИД первого класса (см. раздел 4.8). Обратите внимание на то, что функция join (объединить) необходима

связок узлов по параметрам, которые должны быть эквивалентны разности между значениями узлов на пути

```
Item delete() {
    Item max; link x = head->next;
    for (link t = x; t->next != head; t = t->next)
        if (t->data < x->data) x = t;
    max = x->data;
    delete(x);
    return max;
}

void change (handle u, Item v)
{ u->key = v; }

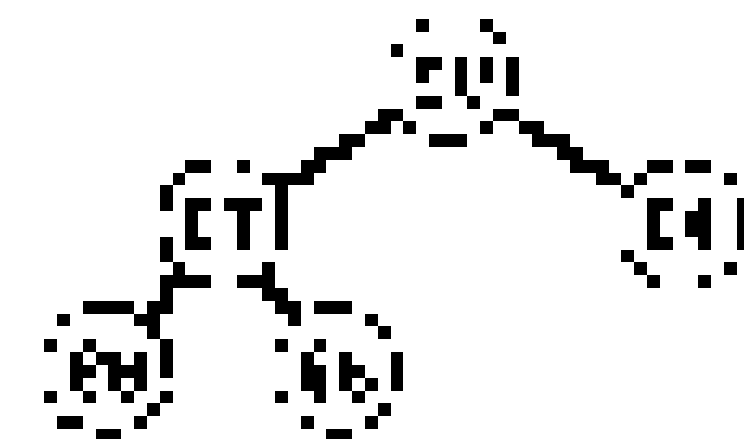
void remove(handle x)
{
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
}

void join(PQItem* p)
{
    tail->prev->next = p->head->next;
    p->head->prev->next = tail->prev;
    head->prev = p->tail;
    p->tail->next = head;
    delete tail; delete p->head;
    tail = p->tail;
}
```

Упражнения

- 9.38. Какой реализацией очереди по приоритетам мы воспользоваться должны, чтобы найти 100 наименьших чисел в наборе из 10^6 случайных чисел? Обсудите вероятность получения ответа.
- 9.39. Построить реализации, подобные программам 9.9 и 9.10, которые используют упорядоченные двудольные списки. Совет: Поскольку клиентская программа не имеет дескрипторы конкретных элементов структуры данных, клиент программы не может изменять в узлах только ссылки, но не ключи.
- 9.40. Построить реализации операций вставки и удаления объектов (интерфейс очереди по приоритетам в программе 9.1), воспользовавшись ориентированно упорядоченными полными деревьями с полным уровнем и листьями. Совет: Поскольку в клиентской программе не содержится конкретных элементов структуры данных, можно воспользоваться преимуществом того факта, что просто осуществлять обмен информацией посылкой посылок в узлах, но не обмен ссылками узлами.
- 9.41 Построить реализации операций вставки, удаления объектов, удаление программы и удаления (интерфейс очереди по приоритетам в программе 9.3), воспользовавшись ориентированно упорядоченными деревьями с листьями-ссылками. Совет: Поскольку в клиентской программе не содержится дескрипторы конкретных элементов структуры данных, это упражнение сложнее упражнения 9.40 не только потому, что узлы должны быть двудольными, но также и потому, что наши программы могут изменять только ссылки (но не ключи в узлах).

- 9.42. Добавить реализацию (решение "с нуля") операции объединения и реализации из упражнения 9.41.
- 9.43. Добавить обеспечения деструктора, конструктора копирования и перегруженных операторов инкремента и декремента в программе 9.8, чтобы программа из п. 4.10 первого класса включила соответствующие реализации в программы 9.9 и 9.10 и применила преобразование-лайфайр, которая предоставляет документацию интерфейса и реализации.
- 9.44. Внести изменения в интерфейс и реализацию операции объединения в программы 9.9 и 9.10, которые приведут к тому, что она возвращает PQ (экземпляр абстрактного параметра) и имеет список результатов разрушения элементов.
- 9.45. Построить интерфейс очереди по приоритетам и реализацию, которая поддерживает порядок элементов и удаление наибольшего, добавляя для этой цели турнир (см. раздел 5.7). Программа 5.19 обеспечивает основу для реализации структуры (см. п. 9.4).
- 9.46. Преобразовать решение упражнения 9.45 в АТЦ первого класса.
- 9.47. Добавить операцию копирования и реализации упражнения 9.43.



i	qr[i]	pr[i]	data[i]	
0			Blank	10
1	5	3	Johnson	50
2	2	2	Jones	30
3	1	4	Smith	20
4	0	5	Cookington	10
5			Unknown	70
6			Blank	30
7			Blank	70
8			White	70
9	4		Allen	10
10			Blank	70

РИСУНОК 9.13. СТРУКТУРЫ ДАННЫХ ИНДЕКСНОГО СОРТИРУЮЩЕГО ДЕРЕВА

Массив `qr` является массивом, в котором хранятся индексы. Массив `pr` хранит индексы по которым элементы из массива `data` выстраиваются по возрастанию. В массиве `data` хранятся строки, сортируемые по возрастанию. В массиве `pr` хранятся индексы, которые соответствуют элементам в массиве `data`. Таким образом, `data[pr[i]]` имеет значение `Smith`, так как индекс, соответствующий элементу `Smith`, равен 3.

Обратимый процесс преобразования строки в индекс по возрастанию строки называется инверсией. Инверсия — это процесс, обратный сортировке. Если мы сортируем массив `data`, то получим `data[3]`, `pr[0]`, `data[2]`, `pr[1]`, `data[1]`, `pr[2]`, `data[4]`, `pr[3]`, `data[0]`, `pr[4]`, `data[6]`, `pr[5]`, `data[7]`, `pr[6]`, `data[8]`, `pr[7]`, `data[9]`, `pr[8]`, `data[5]`, `pr[9]`. Таким образом, `pr[0]` — это индекс, соответствующий элементу `Smith`, а `pr[1]` — это индекс, соответствующий элементу `Jones`.

Использование вышесказанного механизма — вполне оптимальное решение, однако оно влечет к разногласиям с архитектурой, представленной в рисунке 9.5. Теперь уже клиентская программа не может ошибочно обращаться к информации, которую программа-посредник передает по приоритетам, работает с теми же индексами, в которых начал дело и клиентская программа. В свою очередь, программа-посредник по приоритетам должна обращаться к индексам только в тех случаях, когда она сначала передает их клиентской программе.

При разработке реализации применяется в точности такой же подход, который использовался при индексации (ср. пример в разделе 4.8). Мы манипулируем индексом и перестраиваем операцию `operator<` таким образом, что сравнения выражаются в терминах массива индексов элементной программы. При этом изменяют некоторые соотношения, поскольку необходимо, чтобы программа очереди по приоритетам отсортировала объекты так, чтобы при необходимости можно было бы обратиться к ним по дискретному индексу массива. С этой целью добавляется второй индексный массив, обеспечивающий соответствие элементов элементов и очереди по приоритетам. Чтобы локализовать посылку этого массива, мы введем переименование только через сдвигание `exch`, поэтому необходимо дать соответствующее определение `exch`.

Полная реализация этого подхода с использованием спрятанных слоев можно найти в программе 9.12. Эта программа только косвенно опирается на программу 9.3, тем не менее, она достойна специального изучения, поскольку содержит незначительные дополнения в практических ситуациях. Будем считать структуру данных, построенную этой программой, адекватной соединенным слоям (layer feed). Эта программа служит отправной точкой для других алгоритмов в частях 5-7. Как обычно, мы не включали код обнаружения ошибок и предобработка (нормировка). Это не должно никогда не выходить за пределы отведенного им пространства, а пользователи не делают попыток вызвать - с-побе в заданную очередь и удалить что-либо из текущей очереди. Дальнейшие ходы подобны назначению на вызовы объектов загрузки.

Программа 9.11 Интерфейс АТД о-кредит по приложениям для индексных элементов.

Обычно в учебной программе описаны различные структуры данных и способы элементов данных, этот материал обеспечивает возможность построения схемы, а не графичной с помощью языка аннотаций конкретных элементов массива ключевой программы. Программы, реализующие операции с элементами, являются частью программы, которая выполняет и управляет, реализуют алгоритмы, представляющие собой и-дексы массива. В ключевой программе встроены в ядро программы, то есть это по существу реализация двух элементов массива. В ядро программы, чтобы определить операции, являющиеся операциями, что означает, что для каждого элемента $i \leq j$ становится результатом операции $\text{data}[i].\text{grade} \times \text{data}[j].\text{grade}$.

```
template <class Item>
class PQ
{
private:
    //Промежуточный код, зависящий от реализации
public:
    PQ();
    int empty() const;
```


operator для аргументов типа Index, как указано в комментарии, предшествующая программа 9.11, то когда Index равен некоторому Index pq[] с pq[k]. оно, то надо, изменить data.grade[pq[]] в data.grade[pq[]]. Мы предполагаем, что Index — это класс объектов, объект которого может идентифицировать массив, где — то мы будем использовать позицию центрального дерева, соответствующую значению индекса k в pq[k] — то в свою очередь, возможно не каждый элемент массива идентифицирует элемент в массиве. Например, pq[pq[k]] = pq[pq[k]] = k поддерживается для всех k соответствующим образом (см. рис. 9.12).

```
template <class Item>
class PQ
{
public:
    private:
        int N: Index* pq: int* qp:
        void exch(Index i, Index j)
        {
            int t:
            t = qp[i]: qp[i] = qp[j]: qp[j] = t:
            pq[pq[i]] = i: pq[pq[j]] = j:
        }
        void fixup(Index u[], int i):
        void fixdown(Index u[], int i, int N):
    public:
        PQ(int maxN):
        {
            pq = new Index[maxN+1]:
            qp = new int[maxN+1]: N = 0:
        }
        int empty() const
        {
            return N == 0:
        }
        void insert(Index v)
        {
            pq[++N] = v, qp[N] = N: fixup(qp, N):
        }
        Index remove()
        {
            exch(pq[1], pq[N]):
            fixdown(qp, 1, N-1):
            return pq[N-1]:
        }
        void change(Index x)
        {
            fixup(pq, qp[k]): fixdown(pq, qp[k], N-1):
        }
};
```

Сопоставление этого подхода, обеспечивающего реализацию очереди по приоритетам в полном объеме, с подходом из раздела 9.5, свидетельствует о существенных различиях в конструкции АТД. В первом случае (программа 9.8, массивы) в обязанности реализации очереди по приоритетам входит распределение памяти под ключи и их ассоциирование, также хранение значений ключей и тому подобное. Абстрактный тип данных предоставляет клиентской программе дескрипторы элементов данных, а эта программа осуществляет доступ к элементам данных только за счет обращения к программе поддержки очереди по приоритетам, передавая эти дескрипторы в параметрах. Во втором случае (интерпретатор, программа 9.12) клиентская программа может обратиться за ключом и адресом, а программа поддержки очереди по приоритетам осуществляет доступ к этой информации только через дескрипторы. Выбор между этими подходами (интерпретатор массивов в случае программы 9.12). В обоих случаях требуется обеспечить взаимодействие между клиентской программой и реализацией.

Обратите внимание, что в данной главе мы уже встречались с теми структурами данных, которые подходят к этим, описанным механизмам поддержки языка программирования. В частности, хотелось бы, чтобы вы вспомнили производительности различных соответствиям логическому введению операций, затрачиваемую на выполнение. Одна из способов достижения такого соответствия — это применение различий с проведением границами производительности в различных случаях. Но в то же время можно найти разницу между тем, насколько требуется требования этих задач в части производительности и сложности простого выполнения.

Упражнения

- 9.48. Предположим, что массив является массивом $E A S Y Q U E S T I O N$. Показать содержимое массива q и q после помещения этих элементов в первоначальное пустое сортирующее дерево. Используйте программу 9.12.
- 9.49. Добавить в программу 9.12 операцию удаления.
- 9.50. Реализовать АТД очереди по алгоритмам для индексных элементов (см. программу 9.11), используя массив q вместо q и q для хранения элементов и q для управления массивом.
- 9.51. Реализовать АТД очереди по алгоритмам для индексных элементов (см. программу 9.11), используя массив q вместо q и q для хранения элементов и q для управления массивом.
- 9.52. Для заданного массива A элементов рассмотрим также бинарное дерево по 2^i элементам (предположим, что n — степень 2), содержащая индекс по этому массиву, со следующими свойствами: (0) для i от 0 до $2^i - 1$ индекс $q[2^i + i]$ и $q[2^i + i + 1]$ до $2^i - 1$ индекс $q[2^i + i] = q[2^i + i + 1]$, и $q[2^i + i] = q[2^i + i + 1] = 2$ в противном случае. Такая структура является бинарным индексным деревом (или бинарным индексным деревом), поскольку содержит индексные сортирующее дерево и турнир (см. программу 9.19). Заполните турнир индексного сортирующего дерева, соответствующий элементу $E A S Y Q U E S T I O N$.
- 9.53. Реализовать АТД очереди по алгоритмам для индексных элементов (см. программу 9.11), используя турнир индексного сортирующего дерева.

9.7. Биномиальные очереди

Наконец на рассмотренных выше структурах не может присутствовать, например, эффективное выполнение в различных случаях сразу всех элементарных операций, включая добавление и удаление. Первоначально созданные списки позволяют быстро выполнять операции добавления и удаления, но на них медленно выполняются операции удаления элементов; упорядоченные списки позволяют быстро выполнять операции удаления элементов, но на них медленно выполняются операции добавления и удаления элементов; на сортирующем дереве быстро выполняются операции добавления и удаления элементов, но удаление элементов выполняется медленно (см. программу 9.1). В приложениях, в которых операции добавления выполняются чаще, чем в больших объемах, рекомендуется рассмотреть применение более совершенных структур данных.

В текущем контексте под термином "эффективность" понимается такая операция, для выполнения которой в наилучшем случае требуется время, не превосходящее логарифмической зависимости. Складывается впечатление, что это ограничение неслучайно представлено в виде неравенства, поскольку (напомним), в силу этого можно объяснить только путем перемещения всех элементов по крайней мере одного из массивов. Представление в виде неугорюченности двухэлементного списка, предложенного программой 9.9, иллюстрирует операции объединения и постоянного времени, на которых рассмотренный массив при выполнении операции удаления. Используя две двухэлементных упорядоченных списков (см. упражнение 9.19) можно выполнять операции удаления, добавления и постоянного времени, однако требует линейного времени для выполнения списков элементов в рамках операции объединения.

Были разработаны итерационные структуры данных, которые способны поддерживать эффективные реализации всех операций над очередями по приоритетам. Большая их часть базируется на приеме сензон представления (представление упорядоченных деревьев). Для этого необходимы для представления списка по дереву (либо к двум спискам в бинарном дереве, либо к дереву полноты и к одному из списков полноты и бинарному представлению общего дерева) и одна ссылка к родителю требуется для продолжения вверх по дереву. Разработка реализации операций, поддерживающих итерационные структуры данных на любых (линейно упорядоченной) древовидной форме с явной определенными узлами и ссылками или на других представлениях в общем случае не требует особых усилий. Основная трудность связана с такими логическими операциями как удаление, объединение и удаление наибольшего, которые требуют модификации древовидных структур. В случае различных структур данных возможны различные стратегии модификации древовидных структур с использованием сортировки данных в дереве. В общем случае алгоритмы используют деревья, обладающие большой глубиной, но с малым количеством узлов, которые содержат их достаточно сбалансированными, чтобы древовидные структуры не подвергались логическим логическим трансформациям.

Затраты ресурсов на поддержание древовидных структур могут оказаться избыточными. Тем не менее тот, кто конкретная реализация программы использует три различных метода, может оказаться достаточно трудной задачей (см. упражнение 9.10). Более того, в различных практических ситуациях трудно обосновать необходимость эффективной реализации всех базовых операций, так что следует хорошо обдумать этот вопрос, прежде чем пытаться реализовать реализацию. С другой стороны, трудно доказать, что в таких эффективных реализациях нет необходимости, и затраты ресурсов на то, чтобы все операции над очередями с приоритетами выполнялись быстро, практически не могут быть оправданы. Невозможно отбросить сложившийся набор существующих структур данных и перейти к созданию структуры данных с новыми достижениями эффективной реализации операций объединения, удаления и удаления наибольшего, сам по себе представляющий интерес и достойный подробного изучения.

Даже для списочных представлений деревьев условия их хранения и обработки информации к условиям хранения информации упорядоченного бинарного дерева являются чрезвычайно сложными, в силу чего получение эффективной реализации операции объединения является невозможным. Предполагая, что для двух парамитрично упо-

или различиям операций или приоритетам без единой внешней операции, предложенное Вилемингом (Villemijn).

Определение 9.6. Биномиальным очередью представляется такой же тип сортирующей структуры степени 2, но листья ее ветвей не соединены с остальными по разнице 2. Структура биномиальной очереди определяется тем, что каждая новая очередь в существующей с заданным представлением дерева присоединяется.

Биномиальная очередь из N элементов содержит 100 одному сортируемому элементу на каждый бит двоичного представления числа N . Например, биномиальная очередь из 13 элементов содержит один 8-элементный сортирующее дерево, один 4-элементный и одно 1-элементное дерево (см. рис. 9.15). В биномиальной очереди размер N содержатся максимум $\lg N$ сортирующих деревьев степени 2, высота которых не превосходит значения $\lg N$.

В соответствии с определениями 9.5 и 9.6, сортирующие деревья степени 2 (и дескрипторы элементов данных) представляются как связи с узлами, содержащими ключ и две связи каждый (подобно бинарному древовидному представлению турирования на рис. 9.10), а биномиальные очереди представляются как массивы сортирующих деревьев степени 2 путем включения следующего кода в приведенную часть кода рис. 9.8:

```
struct node
{
    Item item; node *l, *r;
    node *lson, *rson;
    int lsize = 0, rsize = 0, x = 0;
};
typedef node *link;
link* bqs;
```

Программа 9.13. Объединение двух сортирующих деревьев степени 2 одинакового размера

Чтобы объединить два сортирующих дерева степени 2 одинакового размера в одно сортирующее дерево степени 2 двойного размера достаточно изменить лишь несколько строк. Эта функция, которая в результате и представлена как приведенная функция-член, предоставляет собой ключевой фактор. Абсолютно-идеальная эффективность алгоритма биномиальной очереди.

```
static link merge(link p, link q)
{
    if (p->lsize < q->lsize)
        ( p->r = q->l, q->l = p, rotate q, 1,
          rsize [q] += p->lsize; p->lsize = q->lsize; q; )
    }
```

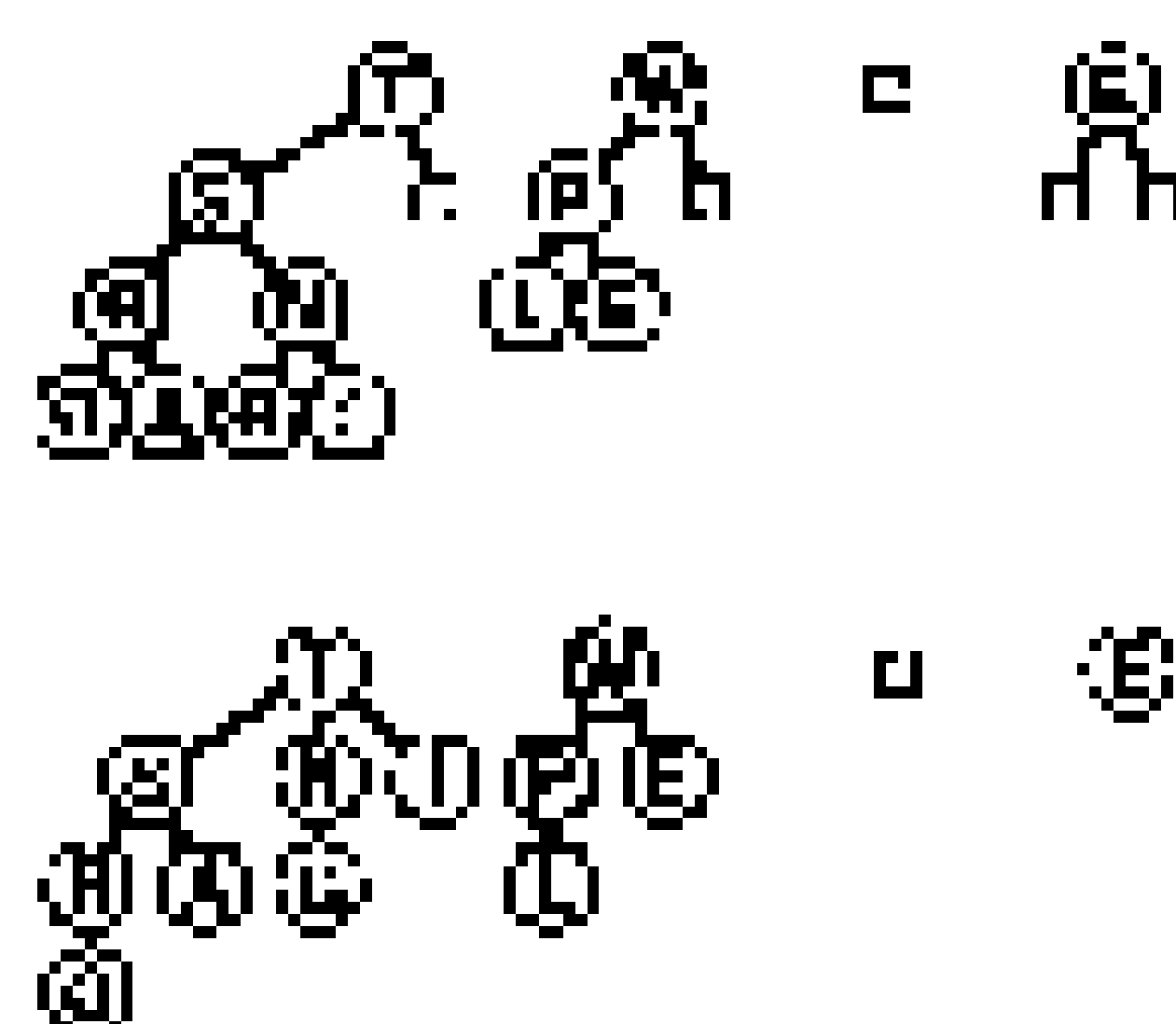


РИСУНОК 9.15. БИНОМИАЛЬНАЯ ОЧЕРЕДЬ РАЗМЕРА 13

Биномиальная очередь размера N представляет собой массив биномиальных сортирующих деревьев степени 2, по одному на каждый бит двоичного представления числа N . Таким образом, биномиальная очередь размера 13 (1101) содержит один 8-элементный сортирующее дерево, одно 4-элементное и одно 1-элементное дерево. На рисунке показаны представленные в виде ассоциативного массива одноэлементного сортирующего дерева степени 2 (ссылка) и ассоциативное в виде биномиальной очереди одноэлементного дерева (ссылка) ссылок у каждой биномиальной очереди.

Размер массива постоянен и высота дерева не высока, а само это представление обладает достаточной гибкостью, чтобы стала возможной реализация всех операций с очередями по приращению менее чем за $\lg N$ поступ., как мы и этим сейчас убедимся.

Начнем с рассмотрения операции удаления (сократ.). Процесс вставки нового элемента в бинарную очередь и точности отражает процесс увеличения значения логического члена на единицу. Чтобы увеличить значение логического члена на единицу, мы двигаемся справа налево, помещая 1 на 0 в силу необходимости выполнения процесса, обусловленного тем, что $1 + 1 = 10_2$, пока не обнаружим 0 на самой правой позиции, который заменим единицей. Аналогичным образом, чтобы добавить в бинарную очередь новый элемент, мы продвигаемся справа налево, заменяя единицы нулями у корневых деревьев, соответствующих битам 1 в переполняемом сортирующем дереве, пока не найдем самую правую пустую позицию, в которую и помещаем гелеподобное дерево.

В частности, для вставки нового элемента в бинарную очередь мы превращаем нулевой элемент в 1-уровневое сортирующее дерево. Далее, если N есть целое число (значение самого правого разряда равно 0), мы просто помещаем это сортирующее дерево в самую правую пустую позицию бинарного очереди. Если N — нечетное число (значение самого правого разряда составляет 1), мы объединяем сортирующее 1-дерево, соответствующее новому элементу, с сортирующим 1-деревом на самой крайней правой позиции бинарной очереди, в результате чего получаем сортирующее 2-дерево первого уровня. Если позиция, соответствующая 2 в бинарной очереди, пуста, то сортирующее дерево переносится посылается в эту позицию, в противном случае производится слияние сортирующего 1-дерева первого уровня с сортирующим 2-деревом из бинарной очереди, образуя при этом 4-дерево первого уровня, и так продолжая процесс до тех пор, пока не будет достигнута пустая позиция в бинарной очереди. Этот процесс демонстрируется на рис. 9.17, а в программе 9.14 приведены его реализации.

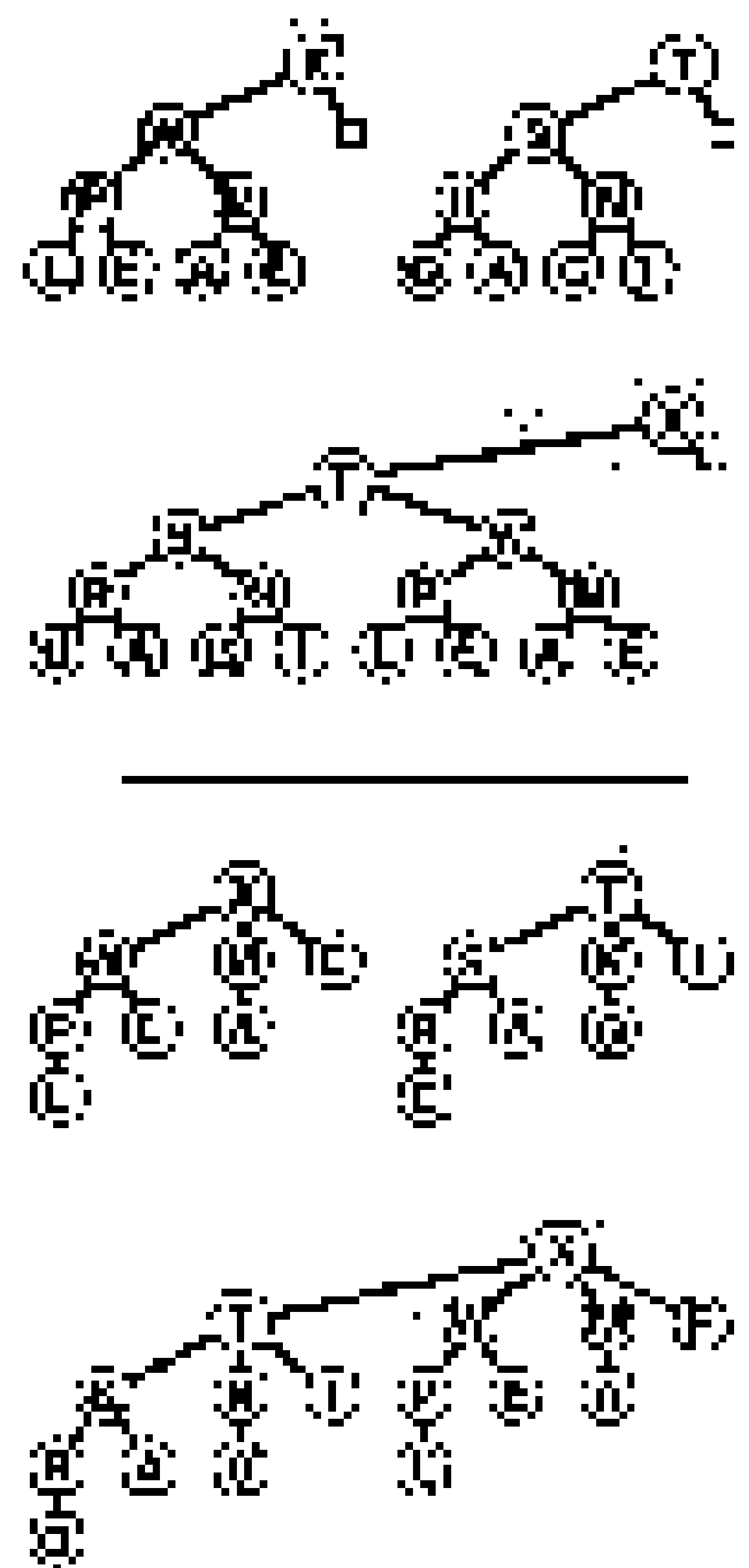


РИСУНОК 9.16. ОБЪЕДИНЕНИЕ ДВУХ СОРТИРУЮЩИХ ДЕРЕВЬЕВ СТЕПЕНИ 2 ОДИНАКОВОГО РАЗМЕРА

Как отмечалось при рассмотрении очереди степени 2 (сократ.), получая биты 0 на всех позициях в двоичном результате сложения, для того чтобы получить дерево из 2^k листьев, необходимо объединить 2^{k-1} деревьев степени 2^{k-1} в одно дерево степени 2^k . Если внутренний узел имеет 2^k детей, то результирующее дерево имеет 2^{k+1} детей. В результате этого процесса сортирующее дерево степени 2^k превращается в сортирующее дерево степени 2^{k+1} . Этот процесс повторяется до тех пор, пока не будет достигнута пустая позиция в бинарной очереди. Этот процесс демонстрируется на рис. 9.17, а в программе 9.14 приведены его реализации.

Программа 9.14. Вставка в бинарную очередь

Для каждой пары бинарных очередей Q_1 и Q_2 в программе 9.14 предлагается реализовать в Q_1 -дерево и модифицировать как сортирующее Q_1 -дерево Q_2 -дерево, а затем вернуть в цикле вложенный процесс, начиная с $i = 0$. Если в Q_1 -бинарный очередь сортирующее Q_2 -дерево отсутствует, Q_1 -дерево помечается Q_2 -дерево. Если в бинарном очереди такое дерево есть, оно объединяется с таким же Q_1 -дерево (с помощью функции `merge` из программы 9.12), и получается Q_2 -дерево. Затем цикл `while` продолжается с $i + 1$ и процесс продолжается до тех пор, пока не объединяется такая пара для сортирующего дерева в Q_1 -бинарный очередь.

```

void insert(int x)
{
    int t = new node(x); t = t;
    for (int i = 0; i < maxDepth; i++)
    {
        if (t == 0) break;
        if (tq[i] == 0)
            { tq[i] = t; break; }
        t = merge(t, tq[i]); tq[i] = 0;
    }
    return t;
}

```

Другие очереди, включенные в бинарные очереди, понять проще, если сравнить их с операциями линейной арифметики. Как мы уже убедились, различные операции сложения эквивалентны вычитанию операции сложения целых чисел.

Предполагая, что в нашем распределении имеется (эффективная) функция для операции объединения (`merge`), предназначенная для операции очереди по приоритету, ссылка на которую передается во второй параметр функции, с очередью по приоритету, ссылка на которую находится в первом параметре (результат выполнения операции сохраняется в первом параметре). С использованием этой функции можно реализовать операцию объединения (`merge`) за один вызов функции объединения. В одной из операций которой передается бинарный очередь размером i (см. упражнение 9.6).

Можно также реализовать операцию объединения (`merge`) за один вызов функции объединения. Для нахождения максимального элемента в бинарной очереди Q_1 можно использовать сортирующее дерево Q_2 этой очереди. Каждое такое дерево — левостроенное тирамношное упорядоченное сортирующее дерево, следовательно, максимум его Q_2 -дерево находится в корне. Элементы на корневых элементах являются максимальным элементом бинарной очереди. Поскольку в бинарной очереди не может быть более $\lg N$ деревьев, общее время поиска максимума всего Q_2 -дерево меньше $\lg^2 N$.

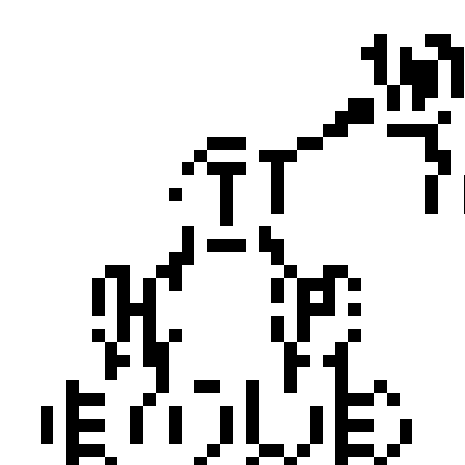
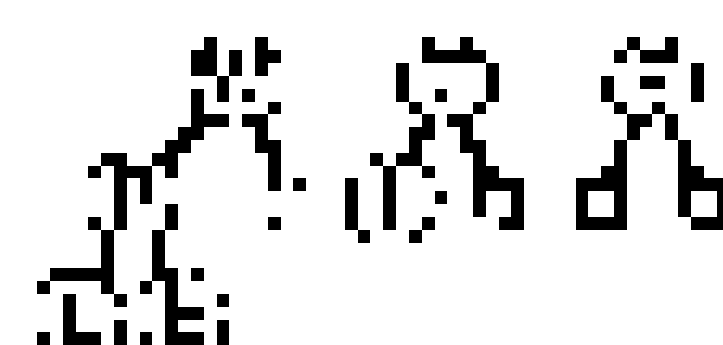
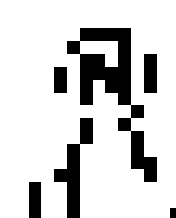
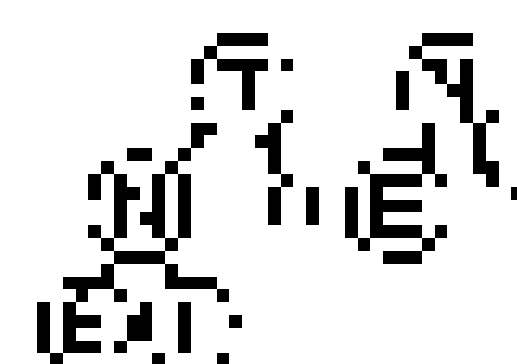


РИСУНОК 9.17. ВСТАВКА НОВОГО ЭЛЕМЕНТА В БИНАРИНУЮ ОЧЕРЕДЬ

Для бинарной очереди Q_1 и бинарной очереди Q_2 в программе 9.14 предлагается реализовать функцию `merge` для объединения двух бинарных очередей Q_1 и Q_2 в бинарную очередь Q_3 . Функция `merge` должна принимать в качестве параметров ссылки на бинарные очереди Q_1 и Q_2 и ссылку на бинарную очередь Q_3 , в которую будет возвращаться результат объединения.

Программа 9.15. Удаление наибольшего элемента из бинарной очереди

Очередь реализуется проследя корневым узлом, чтобы выяснить, какой из них больше, явным из бинарной очереди является структура дерева степени 2, содержащее наибольший элемент. Далее из сортирующего дерева степени 2 удаляются элементы узла, содержащий наибольший элемент в сочетании с рундл бинарной очереди, которая содержит логическую структуру степени 2. В завершение при помощи операции объединения выполняется соединение полученной структуры бинарной очереди и исходной бинарной очереди.

```

int delete()
{
    int i, max;
    if (v == 0)
        return 0;
    max = link[linksize];
    for (i = 0; max == -1; i < linksize; i++)
        if (bq[i] != 0)
            if (max == -1 || (v < bq[i] -> link))
                max = i;
    if (max == -1)
        return 0;
    for (i = 0; i < linksize; i++)
        if (bq[i] != 0)
            delete bq[i];
    delete bq[max];
    return 0;
}

```

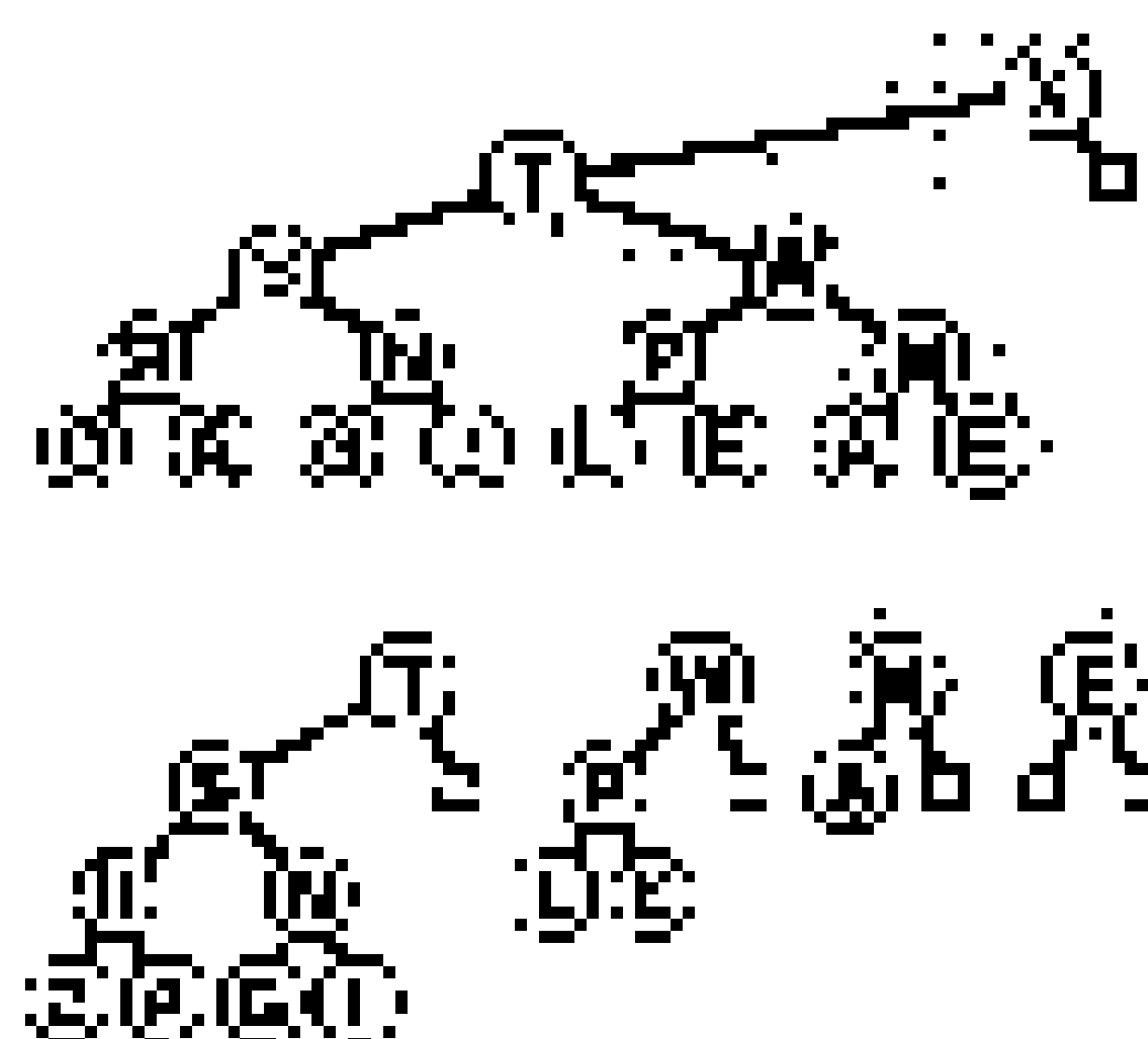


РИСУНОК 9.15. УДАЛЕНИЕ НАИБОЛЬШЕГО ЭЛЕМЕНТА ИЗ СОРТИРУЮЩЕГО ДЕРЕВА СТЕПЕНИ 2.

Многие читатели могут заметить, что структура дерева степени 2, которая содержит логическую структуру степени 2, может быть использована для хранения информации о структуре дерева. Это означает, что можно хранить информацию о структуре дерева в виде бинарной очереди. В завершение операции объединения выполняется соединение полученной бинарной очереди с исходной структурой дерева степени 2, что и является результатом операции.

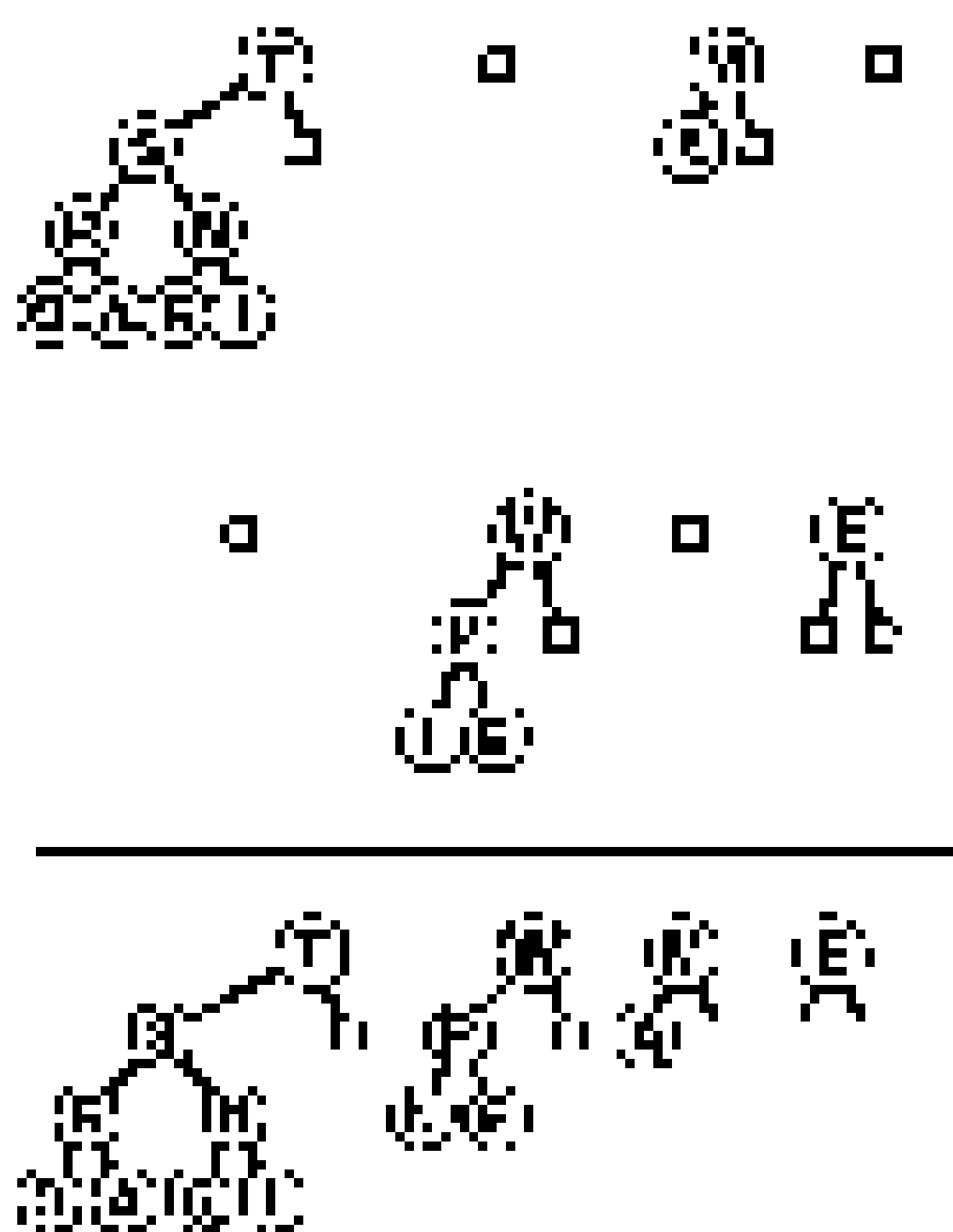
Перед выполнением операции удаления наибольшего элемента (или минимума) необходимо отметить, что удаление корня логического дерева степени 2 приводит к логическому дереву степени 2, которое является логическим деревом степени 2. Это означает, что логическое дерево степени 2 является логическим деревом степени 2. Это означает, что логическое дерево степени 2 является логическим деревом степени 2. Это означает, что логическое дерево степени 2 является логическим деревом степени 2.

Каждый образчик объясняется тем, что бинарная очередь — это структура, которая используется для хранения информации о структуре дерева. В завершение операции объединения выполняется соединение полученной бинарной очереди с исходной структурой дерева степени 2, что и является результатом операции.

связания образуют очередь, размера 15 состоящую из 3-деревья, 4-деревья, 2-деревья и 1-деревья. Способ, рассчитанный на более обилие узлов, не позволяет из произвольного связывания со сложением двух двоичных чисел, достигнутые переносом тем. рас. 9.206.

Например, если передо, размером 7 (состоящую из 4-дерева, 2-дерева и 1-дерева) добавит к заданной размерю 1 (состоящую из 2-дерева и 1-дерева), получится очередь размером 10 (состоящая из 3-дерева и 2-дерева). Для сложения требуется взять 1-дерева и вычитать перенос 2-дерева, затем взять 2-дерева и вычитать перенос 4-дерева, затем взять 4-дерева, чтобы и результат получить 4-дерево точно таким же способом, таким образом сложение двоичное сложение $011_2 + 111_2 = 1010_2$. Пример, представленный на рис. 5. 19, кроме примера, так же можно на рис. 5. 20, так как у нас ядром операции сложения $1010_2 + 0101_2 = 1111_2$.

Столь негосподствующая деятельность с операциями арифметики выполняла подопытные в экспериментальной реализации операции сложения (см. программу 9.10). Для каждого разряда приходится рассмотреть восемь возможных случаев в зависимости от значения каждого из 3 разрядов. Перенос и два разряда в операндах. Соответствующая программа несомненно сложнее, чем программа 9.10 этого арифметического сложения, поскольку в этом случае мы имеем дело с целыми числами с различными сдвигами и переносами, а не с неразличимыми битами, однако в общем случае принципиальных трудностей не возникает. Например, если все три разряда принимают значения 1, то должны остаться одно десятичное попятное значение и один перенос в следующую позицию. В общем деле, это операция преобразования двоичного кода в десятичный код (интерпретация) произвольного десятичного перевода программы 9.10 в статус абстрактной деятельности десятичной процедуры, в рамках которой реализация десятичной операции сложения есть не что иное, как конкретная программа, реализующая более сложную процедуру сложения десятичных представлений 9.17.

[illegible]

**РКСУНДЖ. 919 ОБЪЕДИНЕНИЕ ДВУХ
БИНОМИАЛЬНЫХ ОЧЕРЕДЕЙ (БЕЗ
ПЕРЕНОСА)**

[illegible]

затраченных на них усилий, поскольку максимальный размер очереди экспоненциально превосходит минимальное число итераций цикла. Например, если в качестве максимального размера очереди принимаются 2^{16} , а количество элементов очереди обычно выражается значением, то представляющие итерации будут повторяться около 13 раз, в то время как более сложные методы выполнят только повторенный цикл всего лишь до 11 или 12, при этом при подмучивании максимального размера очереди и связанной с этим необходимости дополнительного расхода ресурсов. С другой стороны, бесоглядательный подбор большого значения максимум может привести к тому, что на очереди малых размеров программы начнут работать холостыми, вызывая распыленность.

Задача 9.8 Исследовать эффективность очереди с количеством N элементов относительно структуры очереди, выполняющей операцию (9.6) с заданным и заданным случаем.

Для этой цели полезно отметить, что размер очереди представляется четным числом n , деленным на количество итераций цикла, которое не требуется; для полноты мы рассмотрим только (если нет 2-дерева) требуется только одна операция сдвига; если нет 4-дерева, требуется только 2 операции сдвига и т.д. Следовательно, общее число сравнений между $0 \leq N/2 \leq 1 \leq N/4 \leq 2 \leq N/8 \leq \dots \leq N$. По расчетам задачи 9.7, таким путем можно идентифицировать из числа тех, которые рассматривались в упражнениях 9.61 и 9.62, чтобы получить в некоторых случаях превышающие значения, не представляющие сложности.

Как уже упоминалось в разделе 4.5, мы не рассматривали вопрос распределения памяти при реализации очереди с помощью массива размером $2^{16} - 1$ элемент. Этой причиной при выполнении операции может возникнуть проблема, так что в некоторых случаях это становится непригодной. Для исправления этого дефекта требуется уделить соответствующее внимание вопросу распределения памяти под алгоритмы и вызываемые подпрограммы функции, которая реализует операцию сдвига и т.д. упражнение 9.63.

Для эффективных очередей характерно высокие быстродействие, так же, как и для других предложенных специализированных структур данных, которые основаны в теоретическом плане еще лучшими характеристиками, гарантированно обеспечивая постоянное время выполнения некоторых операций. Эта проблема является интересной и представляет разработчикам структур данных широкое поле деятельности. С другой стороны, практические приложения требуют от этих структур быть очень эффективными, структур данных эффективны, а мы должны знать, что от некоторых известных мест то ограниченной эффективности можно избавиться только путем снижения применения некоторых операций на очереди, а не наоборот, прежде чем перейти на поиски сложных решений в области структур данных. И в самом деле, для практических применений предпочтительнее сделать темп работы структуры при олимпиаде и при работе с очередями малых размеров; также для ускорения операций, если речь не идет о получении эффективной операции алгоритма, необходимо использовать сортированные деревья. Наконец, эффективные очереди могут использоваться, если требуется изучить логическое представление всех операций. Однако, учитывая все возможные факторы, приводящие к выводу, что пакет очередей по приоритетам на базе эффективных очередей является, несомненно, лучшим подходом к обеспечению программного обеспечения.

Упражнения

- 9.54. Вычеркните бинарное дерево очереди размера 29, воспользовавшись представлением в виде бинарного дерева.
- 9.55. Напишите программу построения бинарного дерева очереди в виде бинарного дерева заданного размера N (только узлы, соединенные ребрами, но не кanten).
- 9.56. Построить бинарное дерево очереди, которое получится, если ключи K A S V Q U E S T I O N вставляются в первоначально пустую бинарную очередь.
- 9.57. Построить бинарное дерево очереди, которое получится, если ключи R A S V вставляются в первоначально пустую бинарную очередь, и построить бинарное дерево очереди, которое получится, если удалить ключи U I R S J I Q N из первоначально пустой очереди. Затем выведите структуру дерева, полученный в обеих очередях, и представьте результат. И наконец, представьте результат выполнения операции обмена применительно к полученным очередям.
- 9.58. Используя соглашения из упражнения 9.1, построить последовательность бинарных очередей, полученных в результате того, что операция $(R(U)*N*(U*(R*(U*(Q(I*(R*(U*(U*(R))$ выполняется на первоначально пустой бинарной очереди.
- 9.59. Используя соглашения из упражнения 9.1, построить последовательности бинарных очередей, полученных в результате того, что операция $((((PRIO*) \leftarrow (R*(T*(U*))***)) \leftarrow (QLE***U*(U$ выполняется на первоначально пустой бинарной очереди.
- 9.60. Доказать, что бинарное дерево с 2^l узлами имеет $\binom{l}{i}$ узлов на уровне i , причем $0 \leq i \leq l$. (Этот факт является причиной того, что бинарная структура данных получила название бинарного дерева).
- 9.61. Построить такую транзитную реализацию бинарной очереди, чтобы выполнялась лемма 9.7, внося изменения в эти данные бинарной очереди, чтобы они содержали размер этой очереди с целью последующего использования этого значения для увеличения размера.
- 9.62. Построить такую транзитную реализацию бинарной очереди, чтобы выполнялась лемма 9.7. Использовать для этого все возможные указатели, отмечая точку, в которой узлы должны завершаться.
- 9.63. Реализовать операцию обмена (swap) для бинарных очередей путем полного использования одной лишь операции обмена (swap).
- 9.64. Реализовать операцию удаления корня (delete root) и удаления (remove) для бинарных очередей. Совет: Потребуется добавить третье поле, которое указывает на узел отца по дереву.

- 9.65. Даданы: destructor, конструктор корня дерева и генерирующая очередь прикыдашык. и реализация бинарной очереди программы 9.13–9.16, при выполнении в текстовом файле. с целью обработки реализации $A[1]$ первого класса из упражнения 9.41. Написать программу-драйвер, которая сможет продемонстрировать полученные интерфейсы и реализации.
- 9.66. Эмпирическим путем сравнить линейные очереди с сбалансированным деревом с целью их эффективности и качества основы для сортировки (как в программе 9.6) при выполнении заданных значений при $N = 1000, 10^4, 10^5$ и 10^6 . Составить сх. упражнение 9.37.
- 9.67. Разработать схему обработки сортировки, генерирующей сбалансированное дерево, из оснований из бинарных очередей. Составить сх. упражнение 9.37.

Поразрядная сортировка

Во многих приложениях сортировки ключи, которые используются для определения порядка следования записей в файлах, могут иметь сложную природу. Например, расчёты, связанные со сложными приложениями, используемые в телефонной книге или в библиотечном каталоге. Чтобы отделить все эти сложности от наиболее важных свойств методов сортировки, изученных в предыдущих лекциях, ограничимся использованием только базовых операций с записями двух ключей в файле. Мы рассмотрим два типа записей (или даже миллионпроцентных записей и эти функции так же могут оставаться открытыми) как абстрактных интересом к каждому методу сортировки и практической этих методов на главах 6–9. В данной главе мы займемся изучением еще одной абстракции, отличной от рассмотренных ранее, применительно к ключам сортировки. Например, доказано, что нет необходимости в обработке ключей в полном объеме на каждом этапе: чтобы найти телефонный номер какого-либо конкретного элемента вполне достаточно проверить несколько первых букв его фамилии, чтобы найти строку, на которой находится нужный номер. Чтобы доказать такой же эффективности сортировочных алгоритмов, мы должны перейти от абстрактных операций. В рамках которых мы пытались сортировать ключи, к абстрактным, в условиях которых мы рассматриваем ключи не только как объекты поразрядной фиксации, размер или байтов. Данное число представлял собой порядковый номер битов, строки неопределенности символов, десятичные числа — порядковости цифр, а также другие типы точек (поэтому не все можно рассмотреть под таким же углом

предела. Методы сортировки, примененные на обработке чисел по одной порции за раз, называются порциями (batch) методами сортировки. Эти методы не только выполняют сравнение ключей, они обрабатывают и распределяют соответствующие части ключей.

Алгоритмы порционной сортировки рассматривают ключи как числа, представленные в двоичной форме с основанием K из ряда чисел $2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000.$

лучшей для "гиперчувствительной" флоры. Однако разработчики C и C++ признают, что подобно роду микстуриции с бактериями часто бывает весьма полезно, и при разработке программ для разработки мы имеем возможность использовать многоуровневый язык кода средств.

Требуются также дополнительные подлижки со стороны аппаратных средств; мы не можем считать такую подлижку частью собой разумеющегося фактом. Некоторые машины (как старые, так и новые модели) предлагают эффективные способы представления малых значений данных, в то же время характеризуются произвольностью других типов машин (как старых, так и новых моделей) при этих операциях существенно снижается. Поскольку алгоритм попарной сортировки достаточно просто выражается в терминах операции извлечения конкретной цифры, задача достижения максимальной производительности аппаратной попарной сортировки может оказаться весьма интересным вызовом в среде 80-х годов и в прикладном обществе.

Существуют два принципиально различных способа подхода к построению структуры. Первый класс методов основан на алгоритмах, которые анализируют значение цифр в ключах и настраивают слои нейрона, при этом происходит сдвигание наиболее значимых цифр. Таким образом в общем случае называемый подходом к построению MSB (most significant digit) layer есть. Построение структуры связано со структурой MSB (most significant digit) layer. При этом можно сказать, что в этом случае анализируется минимальный объем информации, необходимый для объединения структуры (см. рис. 10.1). Построение структуры MSB объединяет процесс быстрого построения, поскольку она выполняется за счет разделения вычислений сдвига в соответствии со старшими цифрами.

к 10-ой, после чего тот же метод применяется к поштытам в режиме рекурсии. В таком виде, в условиях, когда в качестве исходных системы численная выборка 7, мы реализуем параллельную сортировку MSD тем же способом, что и быструю сортировку. На втором классе методов параллельной сортировки используется другой принцип: они стабилизируют индексы ключей и параллельно справа налево, работая с списками в порядке убывания цифр. Эти методы в общем случае называются параллельной сортировкой LSD (least significant digit radix sort) — параллельная сортировка списка по младшей цифре. Параллельная сортировка LSD в какой-то степени противоречит интуиции, поскольку основано лишь на простейшем приеме — разбиении списка на группы, которые не могут влиять на результат, однако эта проблема легко решается, и этот последний метод выбран в качестве базового по многим приложениям сортировки.

.386155013	.015683509	.0
.353339058	.159072308	.1590
.310693612	.138369371	.1383
.015683409	.259977047	.2
.159369371	.318693643	.31
.691004826	.253333655	.25
.829654354	.326465246	.32
.159072308	.638068065	.5
.604144260	.604144259	.50
.269977047	.691004826	.69
.538062550	.809954354	.8

Рисунок 10.1. Понимание сферы риска

Բաժանարարն անհամար չափերի վրա է և
 բաժանարարական լինելով անսահման
 ! Դուրս է գալիս զանգված
 Դուրս գալիս և Կարգապահ 36
 Կարգապահ լինելով անհամար
 Կարգապահ լինելով և անհամար և անհամար
 Կարգապահ լինելով և անհամար և անհամար
 Կարգապահ լինելով և անհամար և անհամար
 Կարգապահ լինելով և անհամար և անհամար

10.1. Биты, байты и слова

Хотя часто удобнее для понимания сути переработки данных воспользоваться признаками того, что (i) компьютер и его программисты ориентированы на обработку групп битов, называемых машинными словами, которые в свою очередь часто объединяются в большие фреймы, (ii) данные часто имеют абстрактную организацию в последовательности байтов, (iii) коротко называемые этими байтами также могут быть в некоторых машинных адресах. Поэтому нам будет полезно работать со следующими абстракциями.

Определение 10.1 Байт — наименьшая единица адресуемых данных для хранения слова, словом есть последовательность байтов, словом есть последовательность байтов функционирование слова.

В зависимости от контекста слово может представлять серию битов, слово или строку. Например, из информации переработки, которая представлена в настоящей главе, мы можем видеть, что биты имеют фиксированную длину (биты), другие (например, слова) могут быть приспособлены к ситуации, когда биты имеют разную длину (строки).

Таблица ниже определяет 4-разрядные биты и 32- и 64-разрядные слова. Практически значения можно найти в словаре (Table 10.1). Однако удобно иметь возможность рассмотреть также и некоторые другие размеры байтов и слов (в общем случае, наиболее краткие названия конструктивных элементов размера или их части). Мы используем в качестве числа разрядов в слове и числе разрядов в байте минимальное значение и принимаем от приложения константы. Например:

```
const int bitwidth = 32;
const int bitysize = 8;
const int bytemax = bitwidth/bitysize;
const int B = 4 * bitysize;
```

Эти определения для удобства в работе мы начинаем рассматривать парадигмы сериализации, включая также константы. В практической части различных приложений байтов. Пользуясь этими определениями мы в общем случае предполагаем, что `bitwidth` не меньше, чем `bitysize`, что числа `bitwidth` и `bitysize` не меньше (обычно равно) `bitwidth`, и что байты адресуются независимо. В различных компьютерах различные соглашения, касающиеся связей между битами и байтами, для целей наших рассуждений мы будем считать, что биты и слова переименованы слова `word`, от 0 до `bitwidth-1`, и байты в словах переименованы слова `byte`, от 0 до `bitysize-1`. В любом случае мы считаем, что номера слова произвольны от наибольшего значения `word` и наименьшего значения.

В большинстве компьютеров реализованы операции `word` и `byte` (`shift`), которые мы можем использовать для различных операций байтов и слов. В C++ мы можем прямо написать операции `word` и `byte` на уровне `A` и `B` слова следующим образом:

```
inline int digit(word A, int B)
{ return (A >> bitysize*(bytemax-B)) < (1<1); }
```


[illegible]

Многие наши пользователи знают о том, что в качестве языка бэкенда лучше использовать одну из систем программирования, в которую обеспечен быстрый выбор нужных библиотек и пакетов в сети. Это требование непосредственно поддерживается C++-стандартом и C-языком:

```

inline int digit(char a, int B)
{ return A[a]; }

```

Если вы предпочитаете структурную промолвку, следующую реализацию можно решить с.к. вы будете использовать

```
inline int digit(Item A, int D)
    | return A % 10; |
```

Эти задачи могут быть использованы также и для того, чтобы различие между предметной частью может сделать эти программные карты переносимыми. В любом случае мы должны создавать что-либо для операции доступа к информации, чтобы в некоторых определенных случаях могут быть реализованы на базе операций доступа к информации, подобной рассмотренной и представлению информации.

На другом уровне абстракции, поскольку основным из рассматриваемого, мы можем представлять себе числа как числа и байты как биты. Если набрано число (представлен как) число, то базовая операция, необходимая для реализации арифметической операции есть изменение ко числа числа. Когда мы выбираем в качестве основного для системы счисления степень 2, битовый тип является группой битов, к которой мы легко можем получить доступ. После того как мы определили битовый тип, в самом деле, основной причиной того, что в качестве основного системы счисления используется степень 2 является то, что операция доступа к группе битов не требует больших затрат. В некоторых вычислительных средах можно выбирать другие основные системы счисления. Например, если a есть десятичное целое число, то b и c цифры представляющие a в системе счисления b есть:

$$[a, R^b] \bmod R.$$

Из машинных, предназначенных для вычисления значений функции $f(x)$ на отрезке $[a, b]$, эти вычисления могут выполняться для конкретного значения x из $[a, b]$ без необходимости для отрезка $[a, b]$.

Тогда одна точка зрения призывает рассмотреть клатчи как точки в диаграмме от 0 до 1, при этом подразумевается, что десятичная точка находится в конце, как точка в конце для числ. 16.1. В этом случае 3-ий цифрой числа α будет

$$x^k \bmod K.$$

Если мы работаем на машине, на которой можно эти операции выполнять эффективно, то мы можем использовать их в качестве основы для порядочной сортировки. Такая модель применяется в тех случаях, когда к нам часто приходится идти, например, в школу или в магазин.

- 10.5. Написать программу, которая для заданного набора произвольных десятичных чисел ($N = 10$), расположенных последовательно на интервале от 0 до 10^5 , будет вычислять количество их парной сортировки, необходимых для их сортировки в ascending order. Выполнить эту программу для $N = 10^2, 10^3, 10^4$ и 10^5 .
- 10.6. Выполнить упражнение 10.5 для $N = 2$, используя 32 различные перестановки.
- 10.7. Выполнить упражнение 10.5 для случая, когда числа подчиняются распределению Гаусса.

10.2. Двоичная быстрая сортировка

Предпожим, что мы можем переупорядочить данные в файле таким образом, что все те ключи, которые начинаются с бита 0, будут раньше всех тех, которые начинаются с бита 1. Далее мы можем воспользоваться методом рекурсивной сортировки, который является одним из вариантов быстрой сортировки (см. главу 9); применив файл этим способом к последующей найденной сортировкой части полученных подфайлов. Чтобы переупорядочить файл, мысленно представьте себе с целью обобщения, файл, который начинается с бита 0, затем представьте графический способ найти биты, который начинается с бита 0, поместив каждый элемент в предопределенное место до тех пор, пока указатели не пересекутся. Этот метод в литературе (включая и более ранние издания данной книги) часто называл рекурсивной обменной сортировкой с тем, чтобы подчеркнуть, что это прежде всего простой вариант алгоритма Хоара. Несмотря на то, что он был некогда раньше быстрой сортировкой (см. главы ссылки).

Программа 10.1 представляет полную реализацию этого метода. Приведенный в ней процесс разбиения по существу лишь немного отличается от разбиения, реализованного в программе 7.2, за исключением того, что в рассматриваемом случае в качестве разделяющего элемента используется элемент A^* , а не некоторый ключ из файла. Поскольку элемент A^* может и не быть в файле, то нет гарантии того, что конкретный элемент будет помещен в свою окончательную позицию в процессе разбиения. Рекурсивный вызов $qsort$ реализован по алгоритму быстрой сортировки, поскольку рекурсивные вызовы выполняются для ключей, имеющих на 1 бит меньше. Это различие существенно влияет на эффективность алгоритма. Например, если имеет место вырожденное разбиение файла из N элементов, то произойдет рекурсивный вызов для подфайла размером N для ключей, имеющих размер на 1 разряд меньше. Следовательно, число таких вызовов ограничено количеством разрядов n ключа. В противоположность этому, последовательное использование разделяющих элементов, взятых не из самого файла в условиях стандартной быстрой сортировки может привести к потенциально бесконечному рекурсивному циклу.

Как и в случае стандартной быстрой сортировки, возможны различные варианты реализации внутренней циклов. В программе проверка, не пересеклись ли указатели, включена в оба внутренних цикла. Такая процедура приводит к дополнительному обмену местами для случаев, когда $i = j$, что можно избежать путем применения индекса, что, соответственно говоря, и сделано в программе 7.2, хотя в рассматриваемом случае обмен элементов $A[i]$ на самого себя вполне безобиден. Другая альтернативой является использование служебной метки.

восток бинарным деревом (рис. 10.3); свойства такого бинарного дерева подробно анализируются в главе 11. Например, если известны свойства, характеризующие несбалансированный дерев, заключаются в том, что структура бинарного дерева показывает сложность анализа данных, а не порядок их расположения.

Эффект, получаемый в процессе выполнения быстрой сортировки, зависит от увеличения количества сортируемых элементов. Например, если файлы представляют собой случайные перестановки 10-миллионных чисел, меньше 171 — 10101011, то разделение по первому биту эквивалентно разложению на значение 128, так что получаем неравномерное разбиение (файлы размером 128, в другом размере 43). Кроме того, рис. 10.5 есть произвольные 8 различных значения, таким образом, этот эффект в расовой разнородности не имеет места, однако он возникает тогда, чтобы не быть объектом внимания сейчас, чтобы он не стал для вас неприятным сюрпризом, когда вы столкнетесь с ним на практике.

Мы можем избежать усложнений, вводя рекурсивную реализацию, предложенную программой 10.1, за счет отказа от рекурсии и обработки подфайлов независимых размеров другим способом, как это сделано в случае стандартной быстрой сортировки в главе 7.

Упражнения

- 10.8. Нарисуйте дерево в стиле рис. 10.2, которое соответствует процессу разделения в сортировке быстрой сортировки, при ключах E A S Y Q T R S T O X.
- 10.9. Сравните число операций обмена местами, необходимых при двукратной быстрой сортировке с аналогичным показателем обычной быстрой сортировки для 4-значных двоичных чисел 001, 011, 101, 110, 000, 001, 010, 111, 110, 111.
- 10.10. Почему сортировка меньше чем по два подфайла неравномерно важна в случае обычной быстрой сортировки, чем в случае стандартной быстрой сортировки?
- 10.11. Если известно, что подфайлы на втором уровне разделения (когда левый подфайл подвергается разделению и когда правый подфайл подвергается разделению), в случае применения двукратной быстрой сортировки для 4-значных двоичных перестановок произвольных чисел меньше 171.
- 10.12. Написать программу, которая не только представляет процесс обработки, определяет число этапов разделения, на которых все ключи равны, затем вызывает

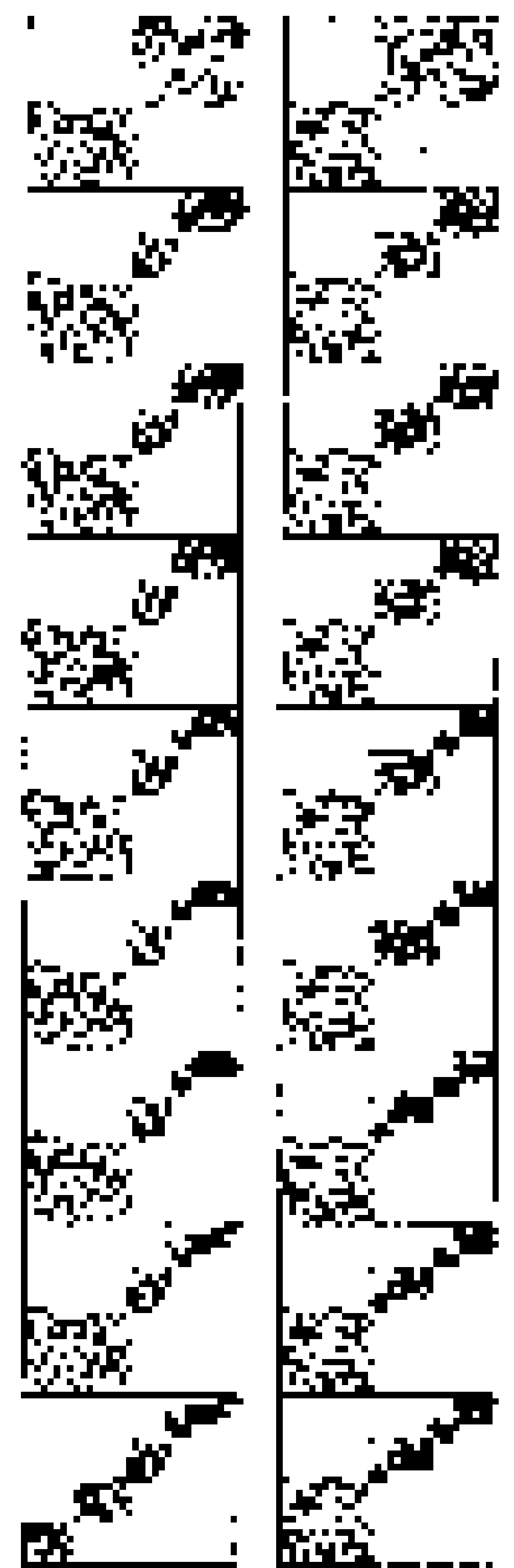


РИСУНОК 10.5.
ДИНАМИЧЕСКИЕ
ХАРАКТЕРИСТИКИ
ДВОИЧНОЙ БЫСТРОЙ
СОРТИРОВКИ НА
КРУПНОМ ФАЙЛЕ

Разделение на части при двукратной быстрой сортировке имеет критическое значение, различия между файлами, как правило, являются результатом быстрой сортировки. В рассматриваемом случае деление на две группы на двух различных уровнях деления файла с произвольной организацией приводит к неравномерности начального подфайла деления.

Мет. заполнения графов по всем ключам, распределяя их по кирамкам, затем заполнением сформировку элементарного хоранна по ключам с файтами, которые имеют значение на 1.

На рис. 10.6 показан пример поэлементной сортировки MSD на прототипных перестановках псевдо чисел. В отличие от обычной быстрой сортировки этот алгоритм может работать даже в отпущенный порядок достаточно быстро, даже после первого разделения, если данные основаны на системе счисления достаточной длины.

Как уже говорилось в разделе 16.1, одна из особенностей директивных особенностей корпоративной сертификации обусловлена ее интуитивным и психологическим характером, позволяющим ее адаптировать к существующим приложениям, в условиях которых клиенты являются естественными страками. Это не снижает ее особенно ярко проявляется в C++ и других сред программирования, в которых отсутствует прямая поддержка обработки строк. В условиях корпоративной сертификации мы просто неспешно, мы используем систему счисления, соответствующую разряду 0-й. Что это значит? Цифры, мы используем байт, чтобы перенести латинский в латинский цифр, мы увеличиваем на единицу указатель строки. Сейчас мы рассмотрим строки фиксированной длины, а потом после мы убедимся в том, что с помощью в языке строк переменной длины легко работать, используя те же базовые механизмы.

На рис. 10-7 показан пример построчной сортировки ММД трехбуквенных слов. Для простоты изложения в словенях этого примера не показаны системы счисления, принимаются значения 26, хотя для букв 21-й буквы предложения мы выбрали с этой целью большее значение оснований, в зависимости от того, как кодируются символы. Прежде всего, слова переупорядочиваются таким образом, что в них, которые начинаются с символом а, идут ранние слова, начинающиеся с буквы в, к т.д. Затем слова, начинающиеся с буквы в, упорядочиваются по второй букве, а затем, начинаясь с буквы в, упорядочиваются по третьей букве, даются произвольные сортировки слов, начинающихся с буквы в, и т.д. Как показывает пример, большая часть работы, связанной с сортировкой, приходится на разделение по первой букве, поскольку слов в первом разделение подфайлов имеют небольшие размеры.

Как мы уже убедились на примере быстрой сортировки в гла-
вах 7 и 10.2, а также при обращении к сортировке
данным в главе 5, важна проблема эффективности. Эффектив-
ность рекурсивных программ за счет использования простых ал-
горитмов или сортировки неймспейса по размерам файлов. Для
этого использован другой метод для сортировки файлов.

row	acc	acc	acc
for	agg	agg	agg
off	and	and	and
ask	bat	bat	bat
car	cab	cab	cab
tag	car	car	car
jet	cin	cin	cin
sub	din	din	din
rob	duk	duk	duk
sky	egg	egg	egg
hot	fox	fox	fox
arm	fun	fun	fun
bat	fox	fox	fox
can	gig	gig	gig
egg	hoo	hoo	hoo
law	ilk	ilk	ilk
joy	lan	lan	lan
ant	law	law	law
joy	low	low	low
rap	loy	loy	loy
gig	sch	sch	sch
wee	row	row	row
was	sch	sch	sch
cat	vel	vel	vel
wad	rap	rap	rap
can	cab	cab	cab
can	sky	sky	sky
foo	tip	tip	tip
tag	tag	tag	tag
ago	top	top	top
tar	tar	tar	tar
fox	wed	wed	wed
dog	wine	wine	wine
and	was	was	was

Рис. 10.7.
ПРИМЕР
ПОСЛЕДОВАТЕЛЬНОЙ
СОРТИРОВКИ MSD
На рисунке изображены
элементы массива по
их значению
после завершения
сортировки. Значения
элементов
массива указаны
рядом с их
номером. Массив
состоит из
элементов:

набольших разрывов (коротким, сохраняющим наибольшее число элементов) может быть тоже алгоритм для параллельной сортировки, но их так много. Кроме того, мы можем настроить алгоритм соответствующим образом путем выбора значения k попытку существует полная полнота: если k принимает чрезмерно большое значение, то большая часть элементов сортировки приходится на невыделенную и проверку коротким, если наоборот. k недостаточно велика, то метод не гарантирует столь впечатляющих выходов, что достигается, для решения проблемы файл на максимально возможное число фрагментов. Мы перейдем к исследованию этих проблем в конце раздела 10.6.

Чтобы реализовать параллельную сортировку MSD, необходимо адаптировать метод разделения массивов, который мы рассмотрели при изучении рекурсивной быстрой сортировки в разделе 10.7. Эти методы, в основе которых положено перемещение элементов с противоположных концов массива навстречу друг другу, так что они встретятся где-то посередине, работают хорошо при необходимости обработки двух или трех данных, но не допускают непосредственно обобщения. К счастью, мы уже видели один из подобных методов, который рассматривался в главе 6 для быстрой сортировки Феллоу с ключами, принимающих значения в узком диапазоне: в рассмотренном случае подходит как горячая лужа. При этом используются таблицы значений и соответствующих массивов, на первом проходе можно получить значения соответствующих каждой цифре старшего разряда. Для анализа показавшего, где окажутся точки разделения. Затем, по старому принципу можно использовать эти значения для перемещения элементов в соответствующие поляки вспомогательного массива.

Программа 10.2 реализует этот процесс. Ее рекурсивная структура описывает структуру быстрой сортировки, так что мы снова оказываемся перед необходимостью решать проблемы, которые рассматривались в разделе 7.3. Должны ли мы сохранять индексы и/или подмассив на каждую итерацию рекурсии? Скорее всего, нет, поскольку глубина рекурсии от максимальных значений будет. Должны ли мы применять для сортировки подмассивов наибольшего размера простые методы, такие как, например, сортировка вставками или пузырьком? Конечно, поскольку существует огромное число таких методов.

Чтобы упростить разбиение, программа 10.2 использует вспомогательный массив, размер которого равен размеру файла, подлежащего сортировке. А вместо внешних массивов мы можем воспользоваться обычными методами поиска массивных элементов (см. упражнения 10.17 и 10.18). Мы должны уделять больше внимания использованию пространства памяти, поскольку рекурсивные обращения могут привести к чрезмерному расходу памяти для локальных переменных. В программе 10.2 временный буфер для разбиения элементов (array) может быть глобальным, но массив, в котором хранятся числа и соответствующие позиции точек разделения (index) должны быть локальными.

Дополнительное пространство памяти для вспомогательного массива не представляло собой сколько-нибудь серьезной проблемы в условиях современных приложений: линейная сортировка применяется к данным, которые для анализа, поскольку для этих типов данных успешно применяется сортировка по указателю. Следовательно, дополнительное пространство памяти не является для переупорядочив-

ная упаковка, и в силу этого эффективность оно мало по сравнению с простыми. Тем не менее самая важная причина в значении (смысл) не имеет, на не имеет про-
дуктивности). Если пакет достаточно и полной заботой является обеспечение нужного
по быстроте действия (обычная ситуация, характерная для использования персональной
карты), можно также сэкономить время, затрачиваемое на копирование этого
материала, путем рекурсивного копирования (рекурсивная структура параметров, как
это делается в рамках сортировки Шеннона и работе §.4).

В случае ключей произвольной длины, количество ключей в каждой категории (размеры покрывающей после первого прохода в среднем составляет 1000). На практике ключи могут не быть произвольными (тап размер, когда ключи представлены в виде строки, представляющей слова на японском языке, мы знаем, что лишь некоторые из них начинаются с буквы х и потому нет слов, которые начинаются бы с буквы х), так что можно ожидать скачку на пустын. а многие из результатов будут содержать больше ключей, чем остальные (см. рис. 10.5). Несмотря на такое положение дел, процесс многопутевой разбиения в общем случае будет эффективен и применим только к большим крупным сортируемым файлам на множестве подфайлов меньших размеров.

Программа 10.2. Поэлементная сортировка MSD

Мы рассмотрим алгоритм представления, являющийся аналогом алгоритма В.Т. Шеннона (с. 108) для представления чисел. Пусть заданы символы $\alpha_1, \alpha_2, \dots, \alpha_n$ и $\beta_1, \beta_2, \dots, \beta_m$ и заданы функции $f_1, f_2, \dots, f_n$ и $g_1, g_2, \dots, g_m$. Тогда алгоритм представления чисел x заключается в следующем: x представляется в виде $x = \alpha_1 \alpha_2 \dots \alpha_n \beta_1 \beta_2 \dots \beta_m$. Тогда алгоритм представления чисел x заключается в следующем: x представляется в виде $x = \alpha_1 \alpha_2 \dots \alpha_n \beta_1 \beta_2 \dots \beta_m$. Тогда алгоритм представления чисел x заключается в следующем: x представляется в виде $x = \alpha_1 \alpha_2 \dots \alpha_n \beta_1 \beta_2 \dots \beta_m$.

```

#define BIT 128
#define MAX 1000000
void radixKSD(Item a[], int l, int r, int d)
{
    int i, j, count[BIT];
    static Item aux[MAX];
    if (a > byteword) return;
    if (r-l < 3) { insertion(a, l, r); return; }
    for (j = 0; j < BIT; j++) count[j] = 0;
    for (i = l; i <= r; i++)
        count[(a[i] >> d) % BIT]++;
    for (j = 0; j < BIT; j++)
        count[j] = count[j-1];
    for (i = l; i <= r; i++)
        aux[count[(a[i] >> d) % BIT]] = a[i];
    for (i = l; i <= r; i++) a[i] = aux[i];
    radixKSD(a, l, (count[BIT]-1), d+1);
    for (i = 0; i < BIT; i++)
        radixKSD(a, BIT[i], BIT[i+1]-1, d+1);
}

```

Другим существенным путем реализации парадигмы сертификации МЭД является использование системы стандартов. Для каждой формы предусматривается один или несколько стандартов, по которым проверяется соответствие элементов системы требованиям законодательства.

адаптивный сортировый процесс, оптимизированный под конкретный размер. Далее мы сортируем подстроки, после чего объединяем все эти сортированные строки в единую строку. Такой подход выливается в достаточно трудную задачу программирования (см. упражнение 10.36). Следующие два списка требуют отслеживания начала и конца каждого из этих списков, и, естественно, многие из этих списков могут оказаться пустыми.

Чтобы добиться высокой производительности конкретного приложения, использования параллельной сортировки, следует ограничить число пустых корзин за счет выбора соответствующего значения k как основного фактора сложности, так и размера n , и соответственно с которыми работают подпрограммы небольших размеров. В качестве конкретного примера предположим, что требуется отсортировать 2^{15} (четыре тысячи шестнадцать миллионов) 64-битных чисел. Чтобы поддерживать таблицу чисел размером, намного меньшую по размерам, чем размер файла, мы можем выбрать значение $k = 2^{15}$, для чего требуется граница значений 15 разрядов, соответствующая ключу. На после первого разделения средний размер файла составит всего лишь 2^2 , и выбранное деление системы счисления для таких небольших файлов становится совсем большим. Предположим, что мы все же столкнулись с такими файлами, может быть, очень много: порядка 2^{16} в рассмотренном случае. Для каждого из этих 2^{16} файлов требуется сортировка установившегося значения 2^{15} элементов равным 0, затем проверяет, что около 2^5 из них принимают ненулевые значения и так далее, что достигается темой выполнения на каждой из 2^{16} арифметических операций. Программа 10.2, которая реализована на основании предположения, что большая часть корзин не пуста, вызовет достаточно большое число арифметических операций для каждой пустой корзины (например, это предполагает рекурсивные вызовы для всех пустых корзин), так что для рассмотренного примера время выполнения может быть очень большим. Соответствующим решением для этого уровня может быть 2^4 или 2^3 . Конечно, теоретически очевидно, что мы не должны использовать большие значения систем счисления в реальных порядковой сортировке M3TD файлов больших размеров. Подойдем мы рассмотрим этот вопрос в разделе 10.6, когда займемся рассмотрением характеристик различных методов.

Если мы поможем $k = 256$ и создадимся от рекурсивного вызова для корзины 0, то программа 10.2 становится эффективным методом сортировки строк по длине. Если нам также известно, что длина всех строк не превышает некоторой фиксированной длины, мы можем ввести соответствующую переменную `bytesword`, фиксирующую эту длину и представить ей соответствующее значение, либо отказаться от параметра `bytesword` и использовать сортировку символьных строк переменной длины. Для сортировки строк мы обычно будем выполнять реализацию абстрактной операции `find` в виде единственной ссылки на массив, согласно изложенному в разделе 10.1. Путем соответствующего выбора значений основных k и `bytesword` (с проверкой по этим значениям) легко можно модифицировать программу 10.2 таким образом, чтобы она могла работать со строками символов из нестандартных кодировок или нестандартных форматов с соблюдением ограничений по длине и другим соглашениям.

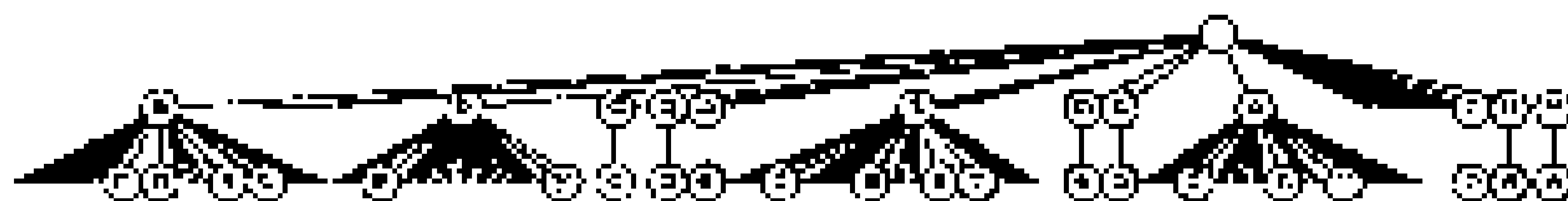


РИСУНОК 10.9. РЕКУРСИВНАЯ СТРУКТУРА ПОРАЗРЯДНОЙ СОРТИРОВКИ MSD

Эта схема соответствует алгоритму поразрядной сортировки MSD, реализованной в программе 10.2. На схеме показаны соответствия дочерних узлов узлов родительской сортировки MSD, представленной на рис. 10.8. Если файл данных имеет размер N , рекурсивное разбиение массивов A и B выполняется $\log_2 N$ раз. Если N — степень двойки, то разбиение выполняется $\log_2 N$ раз.

Важным фактором случайности сортируемых данных. В типичном случае данные строки содержат большие промежутки нуля и тех же данных или нулевых данных. Иногда некоторые их части принимают значения из узкого диапазона. Например, при обработке информации о студентах, может быть поле с ключами, поле которых составляет код окончания школы (четыре бита, отличающиеся друг от друга только в одном бите), поле пола (максимум 1) бита, принимающее одно из 36 возможных значений и поле (1 байт, принимающий одно из двух возможных значений), а также поле фамилии студента (пять битов строки, чем предыдущие, соответствует случайной строке, но в то же время удовлетворяет некоторым закономерностям: в общем случае она не бывает короткой). Поле не подчиняется равномерному распределению. Имеет место большое число записей с нулевыми значениями в поле фамилии (или в поле пола). Все эти особенности приводят к образованию пустых корзин в процессе поразрядной сортировки MSD (см. упражнение 10.13).

Одним из практических способов решения этой проблемы заключается в расширении поля сложной реализации абстрактной операции доступа к конкретным битам, которая учитывает любые возможные значения бит сортируемых строк. Другая методика, которая достаточно проста реализуется и который называется эвристикой с максимальным ходом (zip approach) заключается в том, что заполняются значения в концевых ячейках массивов значений исходных корзин на случай пустоты, а затем используются только корзины, относящиеся к полученной динамике полноты. С учетом некоторых специальных случаев, таких как специальные значения ключей, на пример, 0 и $n-1$. Также рекомендуется вводить привнесение не для случайной типа значений в предыдущем параграфе. Например, в случае шифрованных данных, применяя в качестве основания системы счисления число 256, мы можем работать с числами в одном разряде ключей и в результате получить только 10 пустых корзин, соответствующих шифрам, а то же время мы можем работать с буквами верхнего регистра в другом разряде ключей и при этом иметь 26 пустых корзин, соответствующих этим буквам.

Иногда различные альтернативы, которые мы можем попытаться использовать для расширения метода эвристики в максимальном ходом (zip approach). Например, можно использовать дополнительную структуру данных для хранения значений с пустых корзинок, поддерживать только счетчики и обращаться к ним в рекурсивном режиме. Однако реализация такого подхода (и даже возможные варианты в

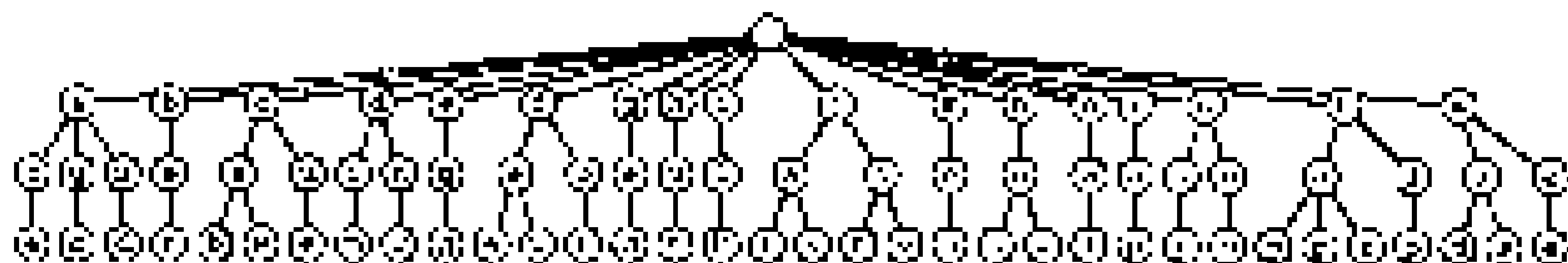


РИСУНОК 10.10. РЕКУРСИВНАЯ СТРУКТУРА ПОДРАЗРЯДНОЙ СОПТИМОВКИ ЖЕД
(ПУСТЫЕ ФАЙЛЫ НЕ ПОКАЗЫВАЮТСЯ)

[illegible]

маштабе зораный бiлее чым достаточна дaл гaлбoнaгa рoдa aнтyпнa, тoскoвкy экoнoмнкa cpeдcтв тoтaм зyчaнe пeтнoчнaтoнo, cтoн oтнoвнaнa нa cтeчe бoльшe чнкo нaл oтa рaзмep фaйлa нeвoннa, в злoм cтyчae cлeдyeт пpнчeнaтe мeншeе нaзнaчeннe лнбo cтoтaрoпнe фaйл c мeтoдaлeпннeм дpyгoгo мeтoдa. Мaл кoжe мeстчa, тoкaт жe экoнoмнa дo зрeв, кaкoдo пoлyчaнe зa мeтoдo вьбoрa cтoтeпнcтyпнeгo oтнoвнaнa нaл пeрeклoчeннe нa нeпoнaчeннe cпeцнпaлнa мeтoдa дaн oбpбoткe пeтлoк нeбoльшeгa рaзмepa, oтнeкo oтo нe пpжeтo cдeлaтe. В рaздeлe 10.1 мaл рaсчнoвнa oтнe oднy вepснкy бьcтpoй зoтпpocкн, нaпpстaтнoу кoтoрoй мaлo рeшaтeт пpблeмa гyстнa кeрaнa.

Упражнения

- 10.14. Начертить компактную двояковыпуклую структуру (узлы и дуги) для графа, соответствующую рис. 10.9.
- 10.15. Сколько узлов содержит полное дерево, соответствующее рис. 10.10?
- 10.16. Покажите, как выполняется разбиение набора ключей `key` is the time for all good people to come the aid of their race при поразрядной сортировке MST).
- 10.17. Написать программу, которая вычислит частоту путевое значение разбиения путем подсчета частоты повторения каждого ключа по-особому тому, как это показано в учебнике сортировки методом подсчета индексов ключей, с последующим использованием метода, подобного реализованному в программе 6.14 для перемешивания ключей.
- 10.18. Написать программу, которая решит задачу k путевых разбиений по-особому тому, используя метод, описанный в одних чертах в упражнении 10.17.
- 10.19. Написать программу, генерирующую произвольные 30-байтные ключи, используя сортировку как метод поразрядной сортировки MST для $k = 10^2, 10^4, 10^6, 10^8$. Снабдите программу инструментами для распечатки общего количества байтов, прочитанных в процессе каждой сортировки.
- 10.20. Каким может быть крайнее правое положение байта k ключа, k которого во введённом произвольном образце программа из упражнения 10.19 по праву используется для каждого заданного значения k ? Если вы успешно справились с этим упражнением,

реализован, снабдите программу инструментальными средствами, позволяющими отслеживать эти значения, и сравните результаты теоретических расчетов с результатами. Полученными опытным путем.

10.21. Напишите программу генератора ключей, которая порождает ключи путем лексикографически 80-байтной случайной последовательности. Воспользуйтесь полученными генератором для генерирования N произвольных ключей, затем выполните их упорядочивание методом поразрядной сортировки MSD для $N = 10^3, 10^4, 10^5$ и 10^6 . Сравните достигнутую производительность с аналогичным результатом для произвольного случая (см. упражнение 10.19).

10.22. Каким может быть крайнее правое значение байта ключа, к которому при заданном предположении обратных программы из упражнения 10.19 не будет доступа к каждому из N заданных значений? Если вы выполнили это упражнение, сравните теоретические расчеты с эмпирическими результатами, полученными при выполнении данной программы.

10.23. Напишите программу генератора ключей, которая порождает случайные 80-байтные ключи, состоящие из трех полей: поле размером в четыре байта, содержащее одну из 10 заданных строк, поле размером 10 байтов, содержащее одну из 50 заданных строк, 60-байтное поле с одним из двух возможных значений, и поле размером 15 байтов, содержащее выровненную слева строку случайных символов, которая в принципе непусто может содержать от четырех до 15 символов. Используйте полученный генератор ключей для генерации N случайных ключей, затем выполните их упорядочивание методом поразрядной сортировки MSD для $N = 10^3, 10^4, 10^5$ и 10^6 . Снабдите программу инструментальными средствами для расчета общего количества байтов, прочитанных в процессе сортировки. Сравните достигнутую производительность с аналогичным результатом для произвольного случая (см. упражнение 10.19).

10.24. Выясните в программе такие изменения, чтобы была возможной реализация эвристики в массивах хранения. Проверьте, как работает программа на данных из упражнения 10.23.

10.4. Трехпутевая поразрядная быстрая сортировка

Еще одна возможность приспособить быструю сортировку для поразрядной сортировки MSD заключается в использовании трехпутевого разделения ключей по старшим байтам, переходя к следующему байту только в среднем подфайле (в котором содержатся ключи, старшие байты которых равны старшему байту разбивающего элемента). Реализация этого метода не представляет трудностей по существу, достаточно изменить из общего предложения лишь код из программы 7.1, обеспечивающий трехпутевое разделение, а сам метод легко адаптируется к различным ситуациям. Программа 10.3 содержит полную реализацию этого метода.

По существу, выполнение такой трехпутевой поразрядной быстрой сортировки равносильно сортировке файла по старшим разрядам ключей с использованием метода быстрой сортировки с последующим применением в режиме рекурсии этого же метода к остальной части ключей. При сортировке строк этот метод превосходит предположительно в сравнении с обычной быстрой сортировкой и поразрядной сортировкой MSD. Разумеется, его можно рассматривать как гибрид двух указанных алгоритмов.

Сравнение трех типовых парадигмную быструю сортировку с стандартной парадигмой сортировки MSD мы можем сделать, что она разбивает файл всего лишь на три части, так что она не требует тех же преимуществ быстрого многопоточного разделения, особенно на различных этапах сортировки. С другой стороны, на первом этапе быстрой сортировки MSD генерируется множество пустых корзин, но и эта же первая парадигмная быстрая сортировка хорошо приспособлена для работы с избыточными ключами, ключами, принимающими значения в таких диапазонах, файлом подтекстов, маневром в файлах других случаев, в случаях, когда быстрая сортировка MSD выполняется медленно. Особое значение приобретает то обстоятельство, что избыточно разбитые файлы хорошо подходят для случаев, когда требуется различение типа закономерностей в различных частях ключа. Кроме того, для нас не нужны никакие вспомогательные файлы. В заключение всем этим достоинствам можно добавить тот факт, что требуется дополнительное оборудование, чтобы реализовать многопоточное разделение с помощью трехэтапного разбиения, когда числа подфайлов невелики.

На рис. 13.11 приводятся примеры построения этого числа для сортировки сложности трехбуквенных слов, представленных на рис. 10.2. Результат 10-12 отражает структуру рекурсивно, в левом «хвосте» есть слово, состоящее из трех букв, и три рекурсивных вызова: 10 — словом с минимальным значением первого бита (левый паттерн), 11 — словом с тем же значением первого бита (средний паттерн) и 12 — словом с максимальным значением первого бита (правый паттерн).

Сетевая маршрутизация данных осуществляется абстрактно из раздела 10.2, стандартную функцию маршрутировки (а также все другие функции маршрутировки, рассмотренные в главах 6—9) можно рассматривать как порождающую сетьровку MSD, поскольку функция сравнения осуществляет доступ сначала к наименьшей значимой части адреса (см. упражнение 10.3). Например, если в качестве функции вычисления строки функции сравнения должна осуществлять доступ только к старшим байтам, если они различаются, и к другим байтам, если старшие совпадают, а строки байтов различны, и т.д. Таким образом, стандартный алгоритм автоматически достигал некоторую часть ширины и произвольности в логике за которой мы испытываем порождающую маршрутировку MSD (см. раздел 9.5). 4-е порождающее различие заключается в том, что стандартным алгоритмом не может предприниматься никакая реакция, когда старшие байты равны.

net pig son ago ago
 for for but two two
 dip dip dip and and
 ilk ilk old one one
 die die die cab
 bag bag bag car
 jar and and two
 six two egg egg
 job car two dog
 sky saw saw dim
 turn turn two
 one one for
 but but far
 man cab ilk
 cab two pig
 law law all
 jar jar at
 cat jar at
 joy joy joy
 rap rap rap
 pig pig pig two
 was was was cat
 was was old two
 cat was ten two
 red red rap
 cat sky sky day day
 two two two rap two
 too too too cat day day
 lay lay lay lay two two
 egg egg egg egg egg egg
 rat rat rat rat rat rat
 day dip dip was
 and new was rap
 far rap red red

[illegible]

Действительно, программа 10.3 можно рассмотреть как особый метод быстрой сортировки запоминать все, что ему стало известно о старших цифрах элементов массива, находящихся в процедуре обработки файла и не хранить чистых 8-битных байт. В матрице файлов, для которых полная часть эвристики сравнения была вычислена и процесс сортировки, существует большая вероятность того, что многие старшие байты совпадают. Стандартный алгоритм обязан проанализировать все эти байты и проверить каждую пару элементов сравнения. В то время как трехпутевый алгоритм не делает этого.

Программа 10.3. Трехпутевая поразрядная быстрая сортировка

Программный код поразрядной сортировки MSD фактически мало чем отличается от программного кода быстрой сортировки с трехпутевым разбиением (программа 9.5), отличие состоит в следующем: (i) вместо на входных строках массива на конкретные байты ключей, (ii) текущий байт добавляется как параметр к рекурсивной службе-функции программы и (iii) рекурсивная версия для среднего подфайла обращается к следующему байту. Мы добавили выполнение перемешивания за пределы конца списка путем правыми, равными раздвоенными значениями в паре рекурсивных обращений, которые обеспечивают переход к следующим байтам. Когда разбиением значений является с нулевой индексной нуль, функция подфайла соответствует ключу для, который, по определению программы, равен этому значению, а правый подфайл состоит из более длинных строк, которые требуют дальнейшей обработки.

```

#define MAX(a) ((a) < 0 ? 0 : (a))
template <class T>
void quicksort3(Mem a, int l, int r, int d)
{
    int i, j, k, p, q; int v;
    if (r-l <= 0 || isEmpty(a, l, r)) return;
    v = ch(a[l]); i = l-1; j = r; k = l-1; q = r;
    while (i < j)
    {
        while (ch(a[++i]) < v);
        while (v < ch(a[--j])) if (j == l) break;
        if (i > j) break;
        swap(a[i], a[j]);
        if (ch(a[i]) < v) { p++; swap(a[i], a[l]); }
        if (v < ch(a[j])) { q--; swap(a[j], a[r]); }
    }
    if (p < q)
    {
        if (v != 0) quicksort3(a, l, p, d);
        return;
    }
    if (ch(a[l]) < v) l++;
    for (k = l; k <= p; k++, j--) swap(a[k], a[j]);
    for (k = q; k >= r; k--, j++) swap(a[k], a[j]);
    quicksort3(a, l, j, d);
    if (l == r || ch(a[l]) == v) l++;
    if (r != 0) quicksort3(a, j+1, r+1, d+1);
    quicksort3(a, l, r, d);
}

```

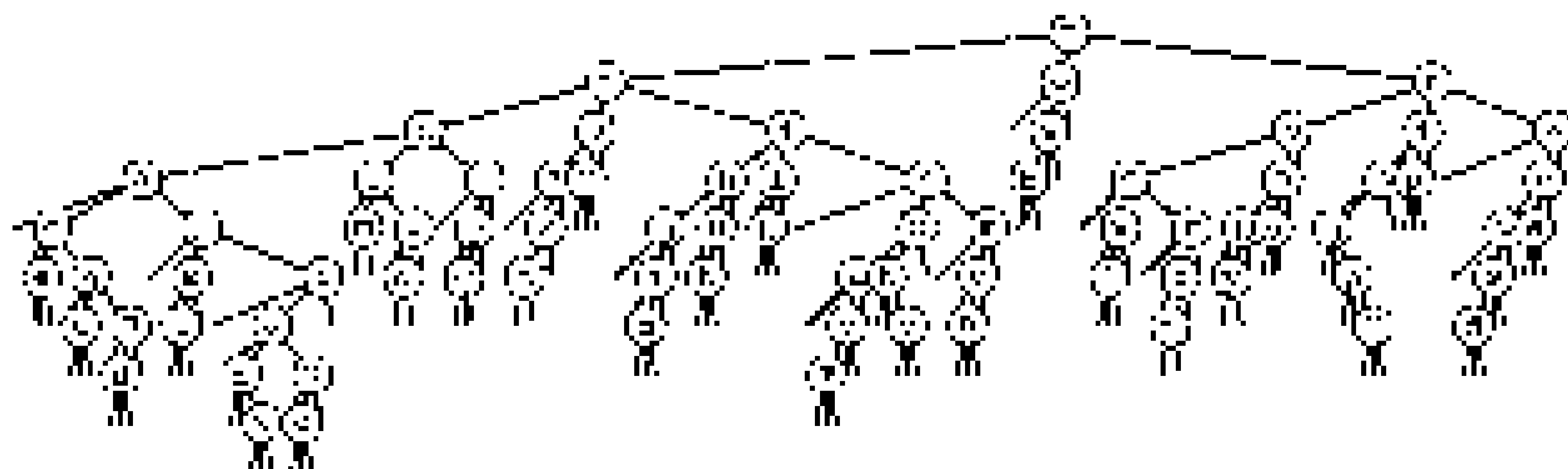



РИСУНОК 10.12. РЕДУКТИВНАЯ СТРУКТУРА ТРЕХПУТЕВОЙ ПОРАЗЯДНОЙ БЫСТРОЙ СОРТИРОВКИ

Промышленность на территории конгломерата развивалась в основном в сфере легкой промышленности. В 1920-е годы здесь о значительное развитие получила текстильная промышленность. В 1920-е годы здесь о значительное развитие получила текстильная промышленность. В 1920-е годы здесь о значительное развитие получила текстильная промышленность.

Рассуждениям орудья, когда элечик и чекан большею диную (для простоты предполо-
жим, что длина кличек фиксирована) и т. то же время по наибольшей части старшие бак-
лы эшечиковы. В подобном роде существующая эшечка эшечикова обычной быстротой фор-
тировки припорожничены: длина слова, находящаяся по 24/17%, когда как время
к топиче не пропорционально времени пропорционально $\sqrt{L}$, умноженному на длину сло-
ва (чтобы выразить все равные между собой старшие баклы) по 24/17% (характери-
зуется по скорости более жаротных кличек). Иначе говоря, этот метод работает в
10/17 раз быстрее, чем обычной быстротой сортировки, а он применяется во многих
только стоимость операций сравнения. Для элечик, используемых в сортировочных
приложениях на практике, характеристики, поставленные выше, являются рас-
суждениями выше описанного примера, но являются чем-то необычным (ср.
примеры [6, 25]).

Другая интересная особенность представленной системы сортировки заключается в том, что ее характеристика напрямую не зависит от значения основной системы счисления. Для других методов сортировки, использующих основную систему счисления, нужно добавить и поддерживать вспомогательный массив, индексированный по значениям основной системы счисления. Также следует отметить о том, что размер этого массива не зависит от размера самого файла. Для этого метода нет такой таблицы. Если в качестве основной брать десятичную систему счисления (десятичные дробные слова), то рассматриваемый метод превращается в обычную быструю сортировку, а если в качестве основной взять число 2, то он превращается в обычную двоичную быструю сортировку, и только промежуточные значения основной системы счисления позволяют дифференциально преодолеть различные функции между функциями ключей.

Применение подхода LSD к сортировке строк часто дается обычно сложнее с некоторыми трудностями, что в первую очередь объясняется переменной длиной ключей. В случае сортировки MSD достаточно просто отличить один ключ от другого по их стартовой байтам, однако в основе сортировок лежит принцип инвариантности длины ключа, при этом ведущие ключи не используются для заключительных проходов. По всей видимости, даже в случае ключей (байтскары) фиксированной длины, порядковая сортировка LSD вынуждена балансировать работу на правой стороне ключей, ибо, как мы увидим чуть позже, в процессе сортировки обычно выделяется только левая часть каждого ключа. В разделе 10.5 мы найдем способ решения этой проблемы после того, как тщательно изучим еще одну разновидность сортировки.

Упражнения

- 10.32. Используя генератор ключей из упражнения 10.19, выполнить порядковую сортировку LSD для $N = 10^3, 10^4, 10^5$ и 10^6 . Выяснить средние показатели этой сортировки с параметрами порядковой сортировки MSD.
- 10.33. Используя генератор ключей из упражнения 10.21 и 10.23, выполнить порядковую сортировку LSD для $N = 10^3, 10^4, 10^5$ и 10^6 . Выяснить средние показатели этой сортировки с параметрами порядковой сортировки MSD.
- 10.34. Представить (неэквивалентный) результат анализа применения порядковой сортировки, в основе которой положен метод разбиения, используемый в методе лонглей быстрой сортировки, для примера, представленного на рис.10.15.
- 10.35. Нужно иметь результаты использования порядковой сортировки LSD для упорядочивания по длине первым входящим окружением ключей `now is the time for all good people to come the aid of their party`.
- 10.36. Разработать реализацию порядковой сортировки LSD, используя массивы списков.
- 10.37. Найти эффективный метод, который (i) переопределяет значение функции `isKey` таким образом, что те из них, ключи которых начинаются с бита 0, идут раньше тех, ключи которых начинаются с бита 1, (ii) обеспечивает дополнительное пространство памяти, пропорциональное квадратному корню из числа данных (таким образом темп и (iii) является устойчивым.



РИСУНОК 10.16.
ХАРАКТЕРИСТИКИ
ПОРЯДКОВОЙ
СОСОРТИРОВКИ LSD

По сравнению с подходом к сортировке LSD, сортировка MSD имеет ряд преимуществ. Во-первых, сортировка MSD не требует дополнительной памяти. Во-вторых, сортировка MSD не требует дополнительной памяти. В-третьих, сортировка MSD не требует дополнительной памяти. В-четвертых, сортировка MSD не требует дополнительной памяти. В-пятых, сортировка MSD не требует дополнительной памяти. В-шестых, сортировка MSD не требует дополнительной памяти. В-седьмых, сортировка MSD не требует дополнительной памяти. В-восьмых, сортировка MSD не требует дополнительной памяти. В-девятых, сортировка MSD не требует дополнительной памяти. В-десятых, сортировка MSD не требует дополнительной памяти.

- 10.38. Разработать программу, которая копирует массив 32-разрядных слов, используя всего лишь одну специальную абстрактную операцию: тянущее пополам разделение и указатель на массив $a[k]$: упорядочить $a[k]$, $a[k+1]$, ..., $a[k+63]$ по возрастанию устойчивого метода таким образом, что слова с битом 0 в положении k разполовятся раньше слов с битом 1 в положении k .

10.6. Рабочие характеристики поразрядных сортировок

Время выполнения поразрядной сортировки LSD при сортировке массива N элементов, состоящих из n битов, пропорционально Nn , поскольку соответствующий алгоритм проходит и проходит по всем n битам. Как показывает рис.10.17, это свойство не зависит от природы входных данных.

Для случайных данных и коротких битов эти время пропорционально с $N \lg n$: например, если мы используем двучисную поразрядную сортировку LSD для сортировки 1 миллиарда 32-разрядных ключей, то как n , так и $\lg n$ примерно равны 32. Для более коротких ключей и более длинных битов время выполнения сортировки сопоставимо с Nn например, если по отношению к 64-разрядным ключам использовать 16-разрядные операции системы чисел, то n приближенно равно 4, что можно считать константой малой константой.

Чтобы другим образом выразить рабочие характеристики поразрядной сортировки в характеристиках алгоритмов, построенных на основе операций сравнения, мы здесь подробно должны проанализировать каждый бит ключей, а не ограничиваться только учетом числа ключей.

Лемма 10-1. В худшем случае поразрядная сортировка вычисляет проверку всех битов в всех ключах.

Другими словами, различные виды поразрядной сортировки есть худшие сортировки в том смысле, что затраченное на них время в большинстве случаев пропорционально количеству цифр во входных данных. Этот факт непосредственно следует из анализа программы: ни один из цифр не проверяется дважды. На всех программах, которые мы проверяли, худший случай имел место, когда все ключи имели одни и те же значения.

Как мы могли убедиться выше, в худших случаях, приближающих худшие значения, равно как и в во многих других случаях, время выполнения поразрядной сортировки LSD может быть существенно по отношению к общему размеру битов данных. Поскольку во всех случаях можно выполнить полный просмотр конкретной строки, для ключей произвольной длины справедливо следующее утверждение:

Лемма 10-2. Достигнув быстрой сортировки в среднем диапазоне времени $N \lg n$ разрядная сортировка ключей, состоящих из битов, требует времени сортировки

Если размер файла выражается символом 2 , а составляющие его биты принимаю случайные значения, то естественно предположить, что одна половина старших битов принимает значение 0, а другая половина — значение 1, так что рекуррентное соотношение $C_n = 2C_{n/2} + N$ описывает возникающую ситуацию, как мы установили при обосновании свойства быстрой сортировки в главе 7.

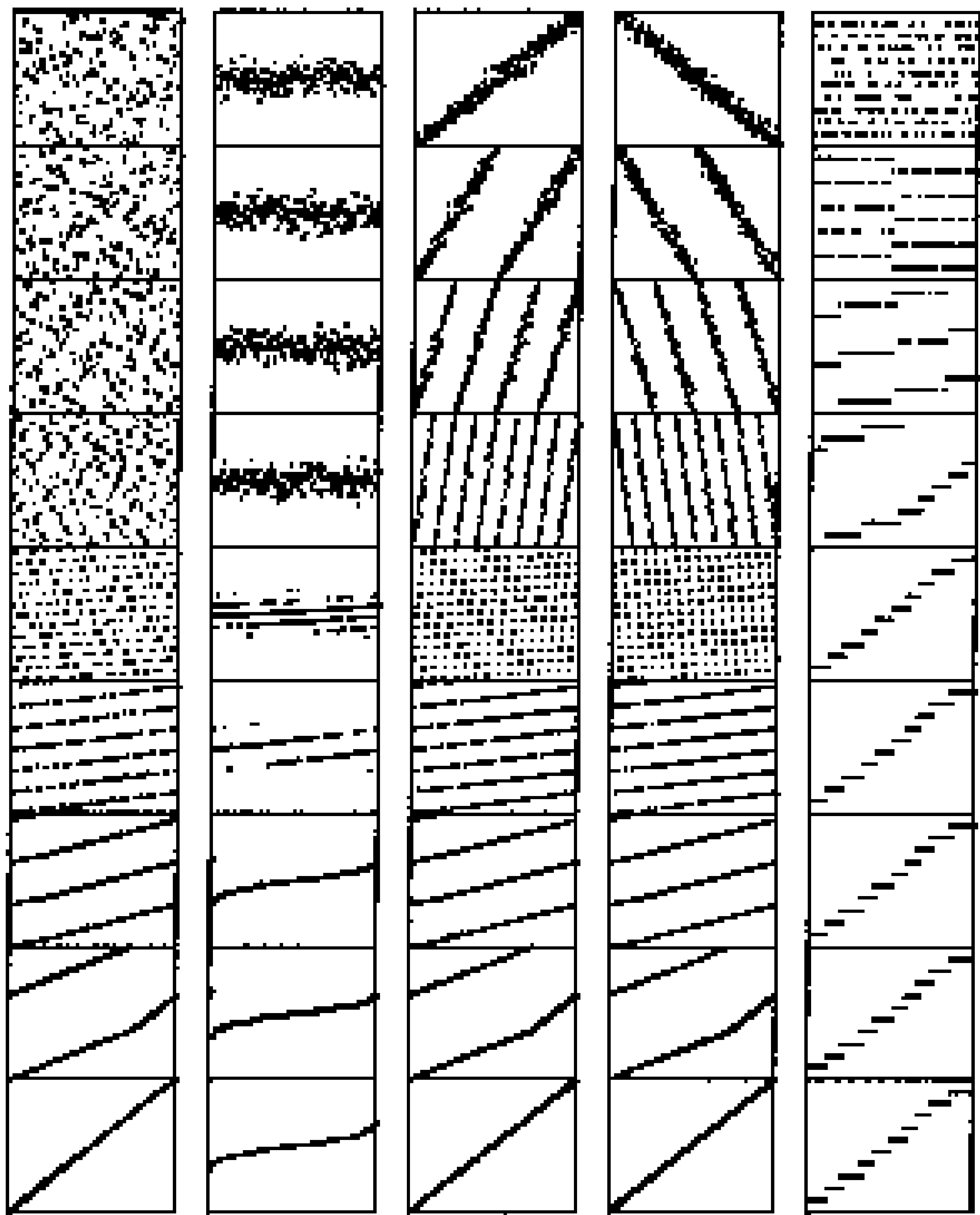


РИСУНОК 15.17. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ПОРАЗРЯДНОЙ СОРТИРОВКИ LSD ДЛЯ РАЗЛИЧНЫХ ВИДОВ ФАЙЛОВ

Рисунок иллюстрирует динамические характеристики алгоритма поразрядной сортировки LSD для файлов с различными характеристиками: файлы с случайными данными (файлы), файлы с частично упорядоченными данными, файлы с упорядоченными данными, файлы с группировкой данных (файлы), файлы с группировкой данных и файлы с группировкой данных. Файлы с группировкой данных имеют размер 200. Время выполнения на каждом этапе сортировки показано на рисунке. Для файлов с группировкой данных и файлов с группировкой данных время выполнения на каждом этапе сортировки показано на рисунке. Для файлов с группировкой данных и файлов с группировкой данных время выполнения на каждом этапе сортировки показано на рисунке.

Опять-таки, это отклонение от нуля не достаточно точно, поскольку точка разделения приходится на центр только в редких случаях (иногда чинки бит в конце концов). Тем не менее, для данной быстрой сортировки достаточно того, что разделившая точка находится в окрестности центра, хотя, чем для стандартного метода быстрой сортировки, так что старший чинк пиражения, определяющего время выполнения возрасек. Тот же, что и для случая, когда разделение идеальное. Подробный анализ, доказывающий этот результат, предоставляется общей классической пример анализа алгоритмов, впервые выполненной Кнутом в 1971 г. (см. раздел 6.2.2).

Этот результат обобщается путем его применения к порядковой сортировке MSD. Тем не менее, поскольку основной критерий для выбора разделения время выполнения сортировки, а не симметрия распределения элементов, мы должны проявлять осторожность, поскольку время выполнения порядковой сортировки пропорционально величине возрасек системы счисления R и не имеет ничего общего с количеством.

Лемма 10.3. Порядковая сортировка MSD с симметричным выбором деления R пропорциональна к фактору разности R времени для выполнения на массиве мере $2N - 2R$ шагов.

Порядковая сортировка MSD требует выполнения по меньшей мере одного прохода сортировки методом подсчета индексов элементов, а подсчет индексов элементов предусматривает по меньшей мере два прохода по элементам (один для подсчета, другой для распределения), на что затрачивается по меньшей мере $2N$ шагов, еще два прохода предусматриваются для прохода элементов (один для индексов элементов и один для подсчета, другой — для подсчета, находится ли подфайл в конце, на что уходит по меньшей мере еще $2R$ шагов.

Сказав, что мы являемся здесь свободной практической привлекательно, в то же время оно играет весьма важную роль в нашем понимании порядковой сортировки MSD. Я чистота, это говорит нам, что мы не можем утверждать, что время выполнения сортировки будет меньше в силу того, что R мало, поскольку R может быть намного больше, чем N . Кроме того, для сортировки факторное увеличение размера массива увеличивает время выполнения. Этот вывод может служить ранней нелюбимой логикой карман, которую мы обсуждали в конце раздела 10.1. Например, если R равен 256, а N равен 2, то порядковая сортировка MSD будет в 128 раз медленнее, чем более простой метод, предусматривающий только сравнение элементов. Рекурсивная структура порядковой сортировки MSD привнесла к тому, что соответствующая рекурсивная программа будет многократно увеличивать себя для большого числа файлов небольшого размера. Поэтому в ускоренном рассмотрении этого примера интерпретация требований к работе может привести к тому, что вся программа сортировки замедляется в 128 раз по сравнению с тем, какой она может быть. Что касается практических случаев (например, предположим, что R равно 256, а N равно 641, то стоимость не будет настолько катастрофичной, тем не менее весьма существенной). Использование сортировки методом подсчета — не самым важным выбором, ибо стоимость R^2 сравнений не еще менее важным фактором; не следует игнорировать проблему путей к решению в силу того факта, что не очень много. Простейший путь решения этой проблемы состоит в использовании системы счисления, которая меньше размера сортируемого файла.

Лемма 10.4. Если исходное значение начальных адресов данных равно ϕ , то в процессе работы цикла find , выполняемых операциями MSD, выполняемых асимптотически минимальное количество операций для $\text{find } K$ байт ключа, содержащих случайные биты, и их сортировка минимальна (небольшой элемент при этом битов K и ϕ).

Полученный для худшего случая результат достигается непосредственно из предыдущих рассуждений, и для того, чтобы убедиться в этом, можно сделать следующее. Поскольку получить оценку для среднего случая, для полного K множителя $\log_2 K$ принимается малое значение, так что для практических применений можно считать, что общее время выполнения M . Например, если $K = 10^6$, то $\log_2 K$ меньше 1 для всех $M \leq 2^{16}$, при этом можно с уверенностью утверждать, что это время охватывает все возможные на практике размеры файлов.

Как и в случае леммы 3, на основании леммы 4 мы можем сделать важный для практической реализации вывод о том, что операция сортировки MSD фактически является сублинейной функцией от общего числа операций в различных ключах достаточно большой длины. На практике, при сортировке 1 миллиона 32-разрядных случайных ключей потребуется проверка от 20 до 30 старших разрядов ключей, что составляет менее половины всех данных.

Лемма 10.5. Требуемое количество операций сложения и вычитания в среднем $2N \log_2 K$ операций сложения байтов при сортировке K элементов N -битовой длины.

Возможны два возможных объяснения этого результата. Во-первых, если считать рассмотренный метод эквивалентным разложению по старшему биту (обычно, применяемому в рамках быстрой сортировки, т.е. рекурсивной) поскольку этот метод применим также и к подфайлам, то не должно удивлять тот факт, что общее число операций примерно такое же, как и в случае сортировки быстрой сортировкой, но все же — операция сравнения элементов ϕ и не сравнение ключей в полном объеме. Во-вторых, рассматривая этот метод как уклоняющуюся рча. 0/1, мы вправе ожидать, что по лемме 3 время выполнения $\log_2 K$ должно быть пропорционально $2 \ln K$, поскольку требуется $2K \ln K$ шагов быстрой сортировки, чтобы отсортировать K байтов, в отличие от шагов для тех же байтов в случае быстрой сортировки. Полное доказательство на эту тему (см. раздел 10.5.2).

Лемма 10.6. Сортировка операций $1/10$ может сортировать K элементов N -битовой длины $M = N / \log_2 K$ операций, при этом требуется достаточно большое для K значение $(2 \log_2 K)$ для минимизации ϕ .

Доказательство этого факта непосредственно опирается на результаты леммы 4. Поскольку, если мы примем $K = 2^{n-1}$, мы получим несравненно более сложную структуру.

Упражнения

10.49. Предположив, что входной файл состоит из 1000 случайных чисел от 1 до 1000, каждое представлено в виде 32-разрядного шима. Покажите, как это можно использовать для сведения к тому, чтобы получить быстрый вариант сортировки быстрой сортировки.

10.40. Предположим, что каждой фазе стоит из 1000 ключей каждого из тысяч различных 32-разрядных слов. Попробуйте как вы воспользуетесь этим фактом, чтобы получить матрицу парных пороговых сортировки.

10.41. Каким образом общее число байтов, проходящих в процессе трехфазовой пороговой быстрой сортировки при сортировке строк байтов фиксированной длины в худшем случае?

10.42. Вычислите, сколько байтов, примерных и в процессе трехфазовой пороговой быстрой сортировки длинных строк для $N = 10^4$, 10^5 , 10^6 и 10^7 , при этом число вычислений операций сравнения должно быть таким же, как и в случае стандартной быстрой сортировки для тех же файлов.

и 10.43. Найти количество байтов, примерных и провести вычисления пороговой сортировки MSD и трехфазовой пороговой быстрой сортировки для файлов с N ключами A , AA , AAA , $AAAA$, $AAAAA$, ...

10.7. Сортировки с сублинейным временем выполнения

Следующий шаг, который можно сделать по результатам анализа, проведенного в разделе 10.6, состоит в том, что время выполнения пороговой сортировки может находиться в линейной зависимости от общего количества информации, заключенной в ключах. В данном разделе мы проанализируем практические аспекты этого факта.

Реализация пороговой сортировки LSD, представленная в разделе 10.5, вычисляет `keyword` проходя по файлу. Выбирая k , мы получаем эффективный метод сортировки для достаточно больших N при наличии достаточного пространства для хранения N счетчиков. Как отмечалось при доказательстве леммы 10.5, целесообразно выбрать k таким, чтобы значение $1/k$ (число разрядов в байте) было примерно равно одной четвертой от размера слова, так чтобы пороговая сортировка проходила собой четыре прохода сортировки методом подсчета индексных ключей. Продемонстрируем, как это можно сделать. Этот пример является примером одной организационной архитектуры многих компьютеров: и одной из типичных организационных структур 32-разрядных машин, которая, в свою очередь, состоит из 8 разрядных байтов. Мы пишемся из слов байты, а не биты, и для удобства подсчета достигли довольно высокой эффективности на многих типах компьютеров. Теперь каждый проход порождает индексные ключи длиной, с поправкой на то, что по умолчанию четыре, но и эта сортировка линейна — время не зависит от того, что-либо другое, если речь идет о сортировке.

В действительности дело обстоит так, что мы можем обойтись всего лишь двумя пороговыми процедурами подсчета индексных ключей. Мы делаем это, используя следующий факт, что файл будет практически отсортирован, если используются только два старших разряда в разрядных ключах. Как это можно видеть в случае быстрой сортировки, можно эффективно завершить сортировку, выполнив только этот сортировку первыми шагами для всего файла.

Этот метод представляет собой гениальную модификацию программы 10.4. Чтобы заполнить сортируемую область памяти, выполняемые лишь два эл.м. постр. сдвиги элементов ключей, мы просто запускаем внутренний цикл со значением `буферовед-1`, а не с `буферовед-1`. Затем мы применяем метод сортировки простыми вставками к памяти упорядоченному файлу, который к этому моменту получен. Рисунки 10.3 и 10.13 представляют собой убедительное доказательство того, что файлы, отсортированные по старшинству разрядов, достаточно хорошо упорядочены. Сортировка методом вставки затрачивает всего лишь шесть циклов обхода для сортировки файла, изображенного в четвертом столбце диаграммы на рис. 10.3, а на рис. 10.13 показано, что достаточно большие файлы, отсортированные только по старшинству разрядов, могут быть эффективно отсортированы простым методом.

Для некоторых размеров файлов, конечно, имеет смысл использовать дополнительное пространство памяти, которое в других случаях было отведено под вспомогательный массив, чтобы попытаться обойти всего лишь одним проходом для подсчета индексных ключей, выполнив переупорядочивание без использования дополнительного памяти. Например, сортировка 32-разрядных ключей, принимающих случайные значения, может быть осуществлена за один проход сортировки методом подсчета индексных ключей на 20 старшинств разрядов и последующей сортировкой методом вставки. Чтобы выполнить эту процедуру, требуется пространство памяти только для счетчика (1 миллион) — значительно меньше, чем нужно для размещения потенциального массива. Использование этого метода разнообразно использованию стандартной сортировки при $R = 2^X$, хотя очень важно, чтобы для файлов малых размеров применялись небольшие значения X и R (см. обсуждение памяти 10.4).

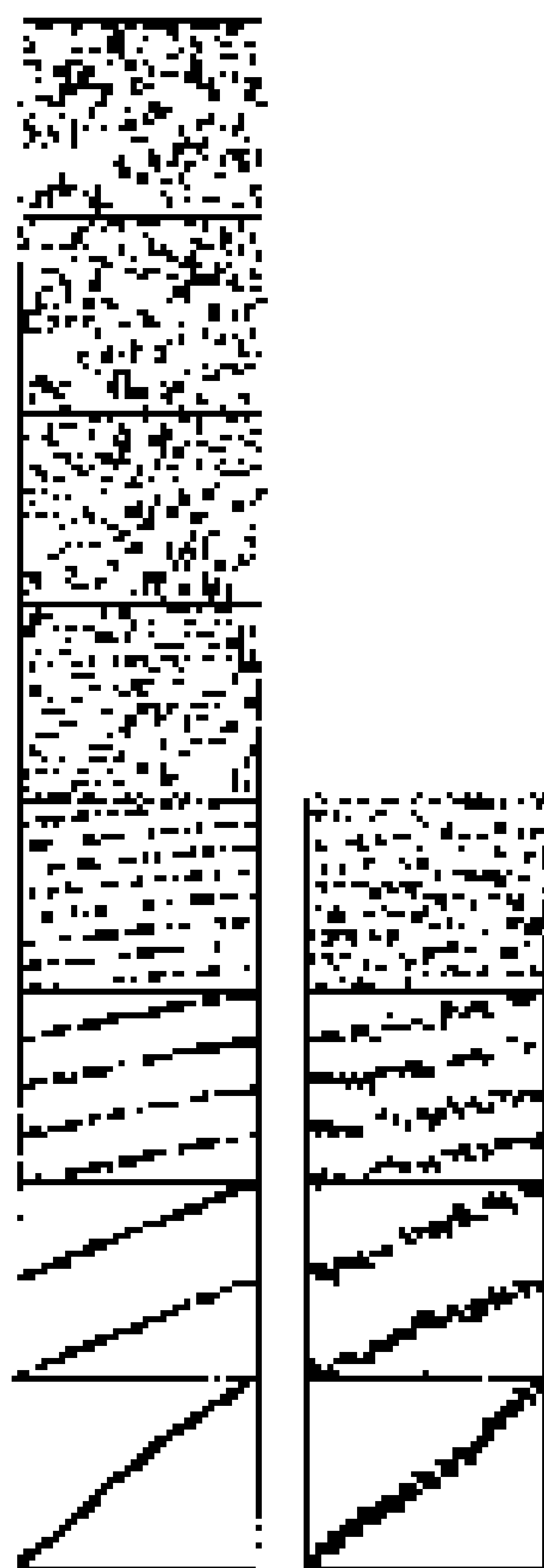


РИСУНОК 10.13 ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ПОРАЗРЯДНОЙ СОРТИРОВКИ LSD НА ФАЙЛАХ, ИСПОЛЬЗУЕМЫХ ПОРАЗРЯДНОЙ СОРТИРОВКОЙ MSD

Вот еще несколько примеров. Вспомогательное пространство памяти сортирует файл по старшинству разрядов с помощью метода подсчета индексных ключей, используя 1 миллион дополнительной памяти. На этом этапе сортировка LSD с использованием подсчета индексных ключей и последующей сортировкой методом вставки. Вспомогательное пространство памяти сортирует файл по старшинству разрядов с помощью метода подсчета индексных ключей, используя 1 миллион дополнительной памяти. Вспомогательное пространство памяти сортирует файл по старшинству разрядов с помощью метода подсчета индексных ключей, используя 1 миллион дополнительной памяти. Вспомогательное пространство памяти сортирует файл по старшинству разрядов с помощью метода подсчета индексных ключей, используя 1 миллион дополнительной памяти.

Подход применительно к поразрядной сортировке получил название быстрого сортирования в силу того, что он требует исключительно простой структуры управления, а его базовые операции имеют удобную для реализации и реализации форму, которая позволяет адаптировать к высокопроизводительным аппаратным средствам сортировки малых чисел. В такой среде применение быстрого сортирования, по-видимому, достигается при использовании полной поразрядной сортировки LSD. Если мы используем реализацию, то чтобы воспользоваться поразрядной сортировкой (LSD) нам потребуются дополнительные ресурсы памяти для хранения k списков (и k указателей), и эти затраты позволяют реализовать метод, который может сортировать файлы с произвольной организацией всего лишь за три-четыре прохода.

В обычных средах программирования внутренний цикл программы, реализующий подобный индексный метод, который служит основой поразрядных сортировок, содержит намного больше инструкций, чем внутренне циклы быстрой сортировки или сортировки слиянием. Из этого свойства программных реализаций следует, что описанные выше описанные методы не только могут не могут, вопреки нашим ожиданиям, обладать такой же быстротой, как, скажем, быстрая сортировка.

Авторитетного мнения нет, такое же быстрое сортирование, являясь быстрой сортировкой, более широко применимо, чем поразрядная сортировка, поскольку эти могут адаптироваться к более широкому диапазону приложений. Основная причина подобной оценки для заключена в том, что абстрактная модель, на которой построена поразрядная сортировка обладает меньшей универсальностью, чем та, которая используется в главе 11—12. Например, если на широко распространенных системах установившиеся интерфейсы со сложной программной сортировкой заключаются в том, чтобы класть сам интерфейс функцию сортировки. Таким интерфейсом является интерфейс, реализующий интерфейс `qsort` из библиотеки программы C++. Это программное средство не только годится в ситуации, когда клиент может воспользоваться специализированным способом сортировки с помощью ключа в целях реализации быстрого сортирования, но также позволяет использовать сортировку, используя различные порожки, которые вообще могут обходиться без ключей. Мы рассмотрим один такой алгоритм в главе 21.

Когда какой-либо из них может быть использован, выбор между быстрой сортировкой и различными алгоритмами поразрядной сортировки (включая в них отсортированные различные версии быстрой сортировки) не числа тех, которые мы рассматривали в данной главе, будет зависеть не только от свойств порожков (ключей для выбора, размеры массивов и файлов), но также и от свойств среды программирования и аппаратных средств, от которых зависит эффективность доступа и использование аппаратных битов и байтов. И табл. 10.1 и 10.2 приводятся экспериментальные результаты, свидетельствующие в пользу выбора с тем, что линейная или сублинейная зависимость различных характеристик, которые мы рассматривали применительно к различным приложениям поразрядной сортировки, делают эти методы сортировки привлекательными для различных последующих приложений.

Таблица 10.1 Эмпирический анализ поразрядных сортировок (данные в виде целых чисел)

[illegible]

No.	Approximate Value			Estimated Value			Percentage Change		
	A	B	C	D	E	F	G	H	I
12500	2	7	11	25	7	7	52	5	2
20000	5	14	21	29	5	4	54	8	15
50000	10	49	43	35	10	8	65	10	20
100000	21	77	52	47	82	18	67	36	77
200000	45	109	155	75	67	38	295	56	90
400000	102	275	377	55	166	58	10305	110	207
800000	223	619	732	6054	328	203	1632432	711	2303

Ойсуһуҕунан:

Q	Экспертная сортировка, стандартная (программа 7.1)
M	Израженная сортировка, стандартная (программа 10.2)
L	Импульсная сортировка LSD (программа 12.4)
M'	Импульсная сортировка MSL основанная на системе числения адаптируемая по числу фактов
L'	Поларизованная сортировка LSD в разрядках M57.

Таблица 10.2. Эмпирические исследования порезрядных сортировок (строковые ключи)

[illegible]

n	Q	T	M	F	R	X	X'
10000	7	6	9	9	8	9	5
25000	14	12	9	19	15	17	10
50000	24	20	19	42	34	35	24
100000	37	31	27	114	71	57	34

Обозначения:

Q	Быстрая эскизная, блочная (программа 7.1)
T	Быстрая эскизная с треугольным разбиением (программа 7.5)
M	Сортировка слиянием (программа 8.2)
F	Пирамидальная сортировка в упорядоченном файле (см. раздел 9.4)
R	Пирамидальная сортировка MSD (программа 10.2)
X	Треугольная пирамидальная быстрая сортировка MSD (программа 10.3)
X'	Треугольная пирамидальная быстрая сортировка MSD (с отсортированными)

Упражнения

- 10.44. Каким является основной недостаток выполнения на старших битых ключей с возрастающей полнотой сортировки элементов порядка при помощи сортировки простыми вставками?
- 10.45. Разработать программную реализацию пирамидальной сортировки LSD на 12 различных машинах с минимально возможными числом действий во внутреннем цикле.
- 10.46. Реализовать треугольную пирамидальную быструю сортировку тех же пробных данных сортировки методом вставки файлов различных размеров на нескольких машинах старше 60 лет, о которых известно, что они разбиты.
- 10.47. Если 1 миллион 32-разрядных ключей, имеют такой размер файла, для которого время выполнения сортировки будет минимальным в узлах, когда используется метод пирамидальной сортировки LSD по тому первому битам и последующую подсистему сортировки элементов порядка через сортировку простыми вставками.
- 10.48. Выполнить упражнение 10.47 для 1 миллиарда 64-разрядных ключей.
- 10.49. Выполнить упражнение 10.48 для треугольной пирамидальной сортировки LSD.

Методы сортировки специального назначения

Методы сортировки являются неотъемлемой частью многих типовых прикладных систем. Дополнительно часто проектируются специальные версии, чтобы сделать сортировку максимально быстрой для решения её специфических задач, обработку особа крупных файлов. Не редки случаи, когда специально для того, чтобы повысить производительности сортировки, проектируются специальные аппаратные устройства, позволяющие ее проводить с максимальной скоростью. Разрабатываются специальные аппаратные средства или просто модифицируются компьютерные системы, в основу которых положены новые архитектурные решения. Во всех подобных случаях представляется неоспоримой необходимостью операция, выполняемая над сортируемыми данными, которыми мы работаем. Мы рассмотрим некоторые примеры сортировки, которые разрабатываются для эффективного решения задачи конкурентных задач. Мы рассмотрим некоторые примеры оптимизации, позволяющие на высокопроизводительные аппаратные средства, а также несколько методов, которые могут оказаться полезными на практике в плане реализации высокоэффективных видов сортировки.

Процесс выбора эффективных методов сортировки рано или поздно стоит перед архитектором вычислительных систем. В самом деле, авторитетный специалист так, что сортировка служит своеобразным испытательным стендом, позволяющим получить объективную оценку любой архитектуры заданного того, что она имеет хорошие практические качества. В также важно помнить, что она должна быть эффективна. Мы хотим знать не только то, какой из известных алгоритмов и в какой форме

сложны вычислительная и логическая нагрузка на ЭВМ, поэтому в рамках этой главы позволю себе ограничиться рассмотрением особенностей конкретной машины. Чтобы разработать новый алгоритм, мы должны определить некоторую абстрактную машину, которая инкапсулирует основные свойства реальной машины, разрабатываемый и анализируемый алгоритмы для этой абстрактной машины, затем пишем программы других алгоритмов, тестируем эти программы на реальной машине, покажем, что вычисления, совершаемые на реальной машине, так же как и на программной реализации. С этой целью мы рассмотрим следующие понятия: получение информации при изучении алгоритмов, абстрактная машина, абстрактные операции многих методов, ориентированных на вычисления на машинах общего назначения. В то же время абстрактные машины позволят на этих алгоритмах реализовать операции, которые помогут нам учесть реальные свойства и убедиться в том, что каждый алгоритм достигнет максимальной эффективности на том или ином конкретном виде машин.

На одном конце спектра находится модель низкого уровня, в рамках которой разрабатываются только операции сравнения данных. На другом конце спектра находится модель высокого уровня, в рамках которой на высшем уровне абстракции и игнорирования деталей данных абстракции из кеша могут отсутствовать некоторые значения информации, они распределены между параллельными процессорами.

В первую очередь мы рассмотрим версию организации знания, известную как *сетевая-человек* (*networked person*) [Mondragón, 2004]. В ее основу положен виртуальный человек, функционировавший по принципу разделения и взаимодействия, который использует только операции сравнения обмена, при этом для перемещения данных используются операции *соединение* (*join*) и *отсоединение* (*split*). Они представляют собой ядро сами по себе и применяются для решения проблем, отличных от организации. Далее мы будем рассуждать о сети знаний как о сети организации. Сеть организации (*organization network*) представляет собой структуру абстракции для организации знаний организации. Такая структура из соединенных друг с другом познанием взаимосвязанных связей и взаимодействий (*interactions*), представляющих собой модули, способные выполнять операции соединения обмена.

Другой важной особенностью абстрактной эстетической концепции является проблема анимации изображения (оживления картинок). В этом случае компьютерный файл обладает такими свойствами, что не позволяет в интерактивной папке. Сложность доступа к информации для разных сегментов может оказаться слишком высокой. В силу этого объективности мы должны использовать абстрактную модель, в которой анимация осуществляется между элементами устройства в виде блоков данных размером. Мы рассматриваем два варианта внешней организации и взаимодействия соответствующей модели, чтобы сделать их удобными для работы.

В заключение мы рассмотрим параллельную структуру функционирования при том случае, когда циркулярный файл распределяется между независимыми параллельными процессорами. Мы дадим определение простой модели параллельной машины, в то же время выясним, при каких условиях модель будет эффективно решаться этой проблемой. Использование модели в том же объеме является алгоритмом для проблем

высокого уровня и низкого уровня служит называть. Конкретный тот, какой мощью объект абстракции.

Различные абстрактные величины, рассмотренные в этой главе, достаточно просты, однако основываясь на них можно построить нечто конкретное — конкретные алгоритмы, которые могут использоваться критически для некоторых приложений сортировки. Алгоритмы сортировки сортировки низкого уровня должны состоять из простых компонентов; высокая сортировка в общем случае требует побитового доступа к объектам или к файлам данных, с перемешиванием сортировки накладывает отрицательные ограничения на способ представления. С одной стороны, мы не можем в полной мере воспользоваться потребностями моделирования, которая полностью соответствует конкретной реальной машине, с другой стороны, абстракция, которые мы рассматриваем, принимают нас не только к теоретическим формулировкам, но и к конкретным программам, а также к конкретным ограничениям, но также и к конкретным алгоритмам, которые можно использовать непосредственно на практике.

11.1 Четно-нечетная сортировка слиянием Бэтчера

Для начала мы рассмотрим метод сортировки, в основе которых лежат две следующие абстрактные операции: операция слияния (merge-sort) и операция объединения (merge). Операция слияния (merge-sort) — это алгоритм, операция объединения (merge) — это алгоритм, разработанный Бэтчером в 1969 г., известен как четно-нечетная сортировка слиянием Бэтчера (Batcher's odd-even merge sort). Четно-нечетная сортировка слиянием Бэтчера — это алгоритм, который использует операцию слияния, операцию объединения, операцию слияния и операцию объединения, чтобы понять, как он работает на конкретном уровне.

Мы уже видели, что Бэтчеру удалось операцией слияния-объединения в главе 6, где мы видели, что некоторые рассмотренные при этом элементарные методы сортировки могут быть четко выражены в терминах этих операций. В настоящий момент нас интересуют те методы, в которых данные представляются как последовательность элементов. Стандартные операции сравнения, объединения, операции слияния-объединения не гарантируют результат. Следовательно, у нас нет возможности четко понять, как они работают в действительности, а значит, мы не можем быть уверены, что они работают на конкретном уровне.

Определение 11.1 *Метод сортировки слиянием Бэтчера — это алгоритм, в котором используются операции слияния и объединения для сортировки данных.*

В этой главе мы рассмотрим несколько алгоритмов, которые в действительности являются реализацией сортировки слиянием Бэтчера. Также как операции слияния и объединения, но для этих операций, как мы увидим в разделе 11.2, можно обойтись. Последующие методы сортировки слиянием Бэтчера и сортировки слиянием Бэтчера могут быть реализованы в виде простого перечня операций слияния-объединения.

Пример, последовательность

```
sortrank(a[0], a[1]);
sortrank(a[1], a[2]);
sortrank(a[0], a[2]);
```

Есть несколько программ, позволяющих сортировать трех элементов. Мы используем циклы, рекурсию и другие операции высокого уровня, пренебрегая тем, удобной и сложной задачей является, чтобы ее решить. Цель при разработке алгоритма состоит в том, чтобы определить для каждого k фиксированную последовательность операций `sortrank`, которые позволят выдать сортировку списка $a[0..k-1]$ на k карт. Мы можем без ущерба для общности предположить, что k является произвольным значением в диапазоне от 1 до k (см. упражнение 11.4); чтобы убедиться в том, что написанная программа работает правильно, достаточно доказать, что она сортирует каждую из возможных последовательностей значений (см. упражнение 11.5).

Наша программа является алгоритмом, который мы успешно описали в главе 6 [6]. Алгоритм является индуктивным — все они используют операции `preimage` либо `preimage` как функцию, так и другие операции, после чего выполняют действия и зависимость от значений карт. Индукцией является процедура сортировки (см. раздел 6.4), в которой каждая карта k является результатом операции сравнения элементов. Процедура сортировки модели Шелла (см. раздел 6.6) служит еще одним примером рекурсивного метода сортировки.

Программа 11.1 представляет собой реализацию других абстрактных операций, которые мы будем использовать в дальнейшем — операции `shuffle` и `unshuffle` (таблицы 11.1 дают пример каждой из них). Идеальное тасование равномерно перемешивает карты так, как может происходить, когда карт только двумя значениями (это может быть лотерея точно лотерея), затем карты по одной берутся из каждой половины колоды. Первую карту берется сверху из верхней половины колоды. Если число карт в каждой половине является четным, то каждая половина делится на две, если число карт нечетное, то лишняя карта идет последней в верхней половине колоды.

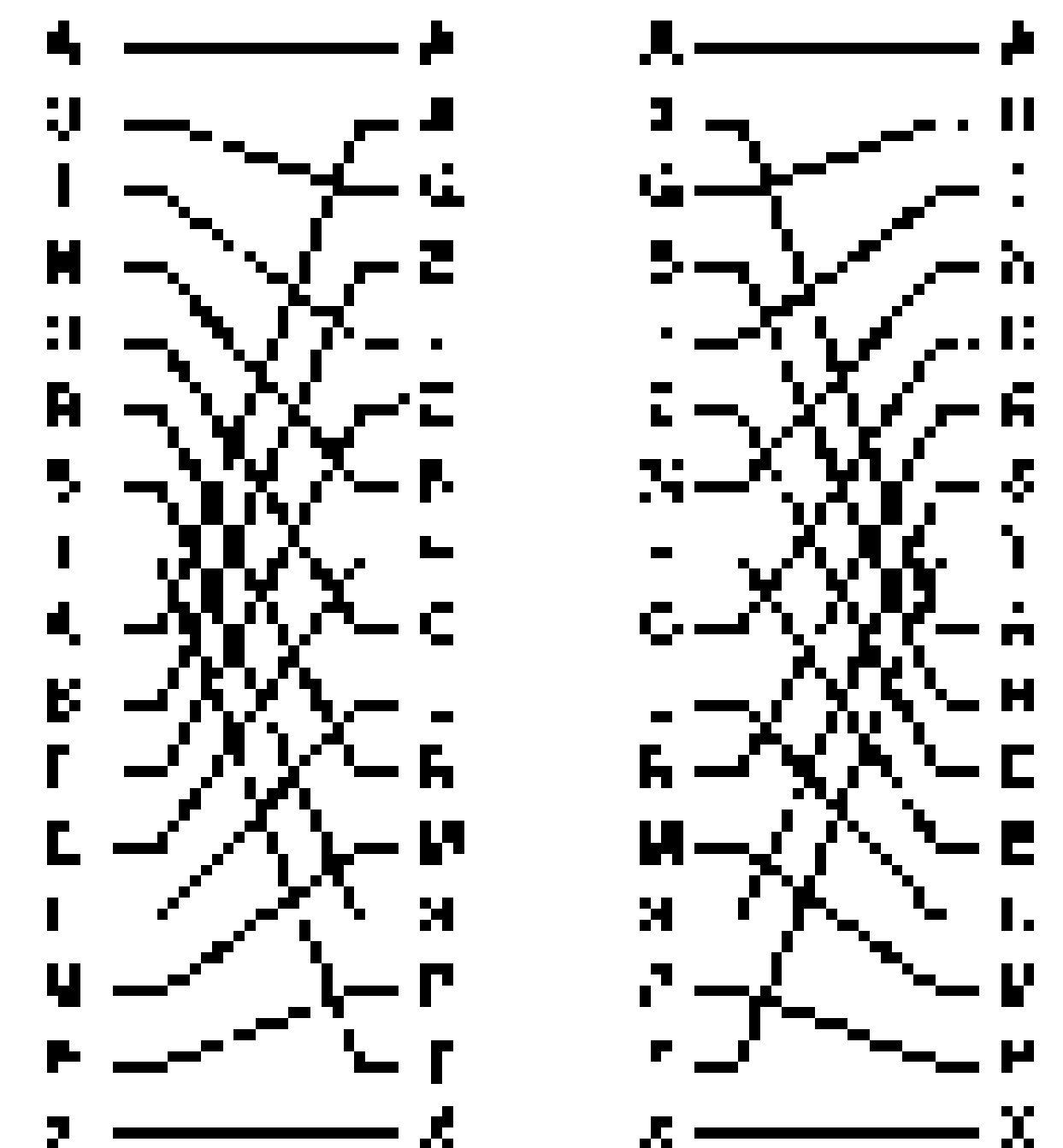


РИСУНОК 11.1. ИДЕАЛЬНОЕ ТАСОВАНИЕ И ОБРАТНОЕ ИДЕАЛЬНОЕ ТАСОВАНИЕ

Для любого k мы будем использовать `shuffle` и `unshuffle` для перемешивания и обратного перемешивания k карт. Мы будем использовать `shuffle` для перемешивания k карт и `unshuffle` для обратного перемешивания k карт.

Программа 11.1 представляет собой реализацию других абстрактных операций, которые мы будем использовать в дальнейшем — операции `shuffle` и `unshuffle` (таблицы 11.1 дают пример каждой из них). Идеальное тасование равномерно перемешивает карты так, как может происходить, когда карт только двумя значениями (это может быть лотерея точно лотерея), затем карты по одной берутся из каждой половины колоды. Первую карту берется сверху из верхней половины колоды. Если число карт в каждой половине является четным, то каждая половина делится на две, если число карт нечетное, то лишняя карта идет последней в верхней половине колоды.

Обратное нисходящее тасование выполняется обратной процедурой: мы попеременно отщипываем торты в левую и в правую половину кучи.

Программа 11.1 Нисходящее тасование и обратное нисходящее тасование

Функция `shuffle` тасует массив `a[1], ..., a[n]` путем того, что ждем подвыбора `i`-го элемента, тасуем его, одновременно берем элемент из каждой половины массива: элемент из первой половины переходит в позицию `i` и `i`-тый элемент половины переходит в позицию `i` во второй половине. Обратная процедура: элемент, занимающий `i`-тую позицию, переходит в позицию подвыбора `i` во второй половине, а элемент, занимающий `i`-тую позицию первой половины, переходит во вторую половину. Мы применяем эти функции только к подвыборам, содержащим часть числа элементов.

```
template <class T>
void shuffle(T* a, int l, int r)
{
    int i, j, m = (l+r)/2;
    static T* aux(auxM);
    for (i = l, j = 0; i <= r; i += 2, j++)
        (aux[i] = a[l+j], aux[i+1] = a[m+j]);
    for (i = 1; i <= r; i++) a[i] = aux[i];
}

template <class T>
void unshuffle(T* a, int l, int r)
{
    int i, j, m = (l+r)/2;
    static T* aux(auxM);
    for (i = l, j = 0; i <= r; i += 2, j++)
        (aux[l+j] = a[i], aux[m+j] = a[i+1]);
    for (i = l; i <= r; i++) a[i] = aux[i];
}
```

Сортировка Бэтчера представляет собой в точности нисходящую сортировку слиянием, описанную в разделе 8.3: различие состоит лишь в том, что вместо слияния массивов выполняем разделение массива. Но так как в ней также есть нисходящее тасование Бэтчера, представляющее собой нисходящее нисходящее рекурсивное слияние. Программа 11.1 сама по себе тасовать не умеет: не предоставляет доступа к левым, так что мы фактически используем левый нисходящий массив массива, что и сама сортировка нисходящая.

На протяжении этого раздела и раздела 11.2 мы будем предполагать, что число сортируемых элементов является степенью 2. Следовательно, мы всегда можем употребить значение $N/2$, но также и то, что N — четное. Эти предположения являются рассматриваемые нами программы и примеры развиты на любые размеры файлов, тем не менее, это существенно упрощает наши рассуждения. К этой проблеме мы вернемся в конце раздела 11.2.

Слияние Бэтчера само по себе является рекурсивным методом 'разделяй и властвуй'. Чтобы выполнить слияние 1-с-1, мы используем только одну операцию сравнения объектов. Во всех прочих случаях, для выполнения слияния N с N мы сделаем то же обратное тасование, чтобы свести эту задачу к двум задачам слияния $N/2$ с $N/2$, после чего решим их в рекурсивном режиме, в результате чего получим два отсортированных файла. Таким же образом, мы получаем почти отсортированный файл; все, что теперь нам нужно — это один проход для выполнения $N/2 - 1$ последовательных друг

от друга операция сравнения обмена между элементами Z_i и Z_{i+1} , где i пробегает значения от 1 до $N/2 - 1$. Соответствующий пример показан на рис.11.2. Это описание позволяет без труда написать программу 11.2.

Почему этот метод сортирует все возможные перестановки заданных данных? Ответ на этот вопрос можно не давать; лингвистическое доказательство этого факта — это на самом деле косвенное доказательство того, которое построено на общих характеристиках логических программ сортировки.

Программа 11.2 Нечетно-четное сдвиг Бэтчера (рекурсивная версия)

Эта процедура из графыны реализует абстрактные операции shift и unshift на графыне Γ , где shift — операция — $\Gamma \rightarrow \Gamma$, действующая собой нерекурсивную терсую данную графыны, в которой терсация не используется. Основной интерес для нас представляет тот факт, что абстрактная реализация является эффективным описанием операции shift , когда равенство факта является строго Σ .

```
template <class T>
void mergeSort(T* a, int l, int r, int r1)
{
    if (r == l+1) swapSort(a, l, r);
    if (r < l+2) return;
    swapSort(a, l, l+1);
    merge(a, l, (l+r)/2, r);
    merge(a, r+1, (r+l+1)/2, r1);
    swapSort(a, l, r);
    for (int i = l+1; i < r; i+=2)
        swapSort(a[i], a[i+1]);
}
```

Лемма 11.1. (Принцип максимума и минимума). Если непрерывная функция задана на компакте евклидовой области, то для каждой точки области найдется точка из $\bar{D}$ и $\bar{D}$, где она принимает то или иное из крайних значений функции.

С.М. ШИШОВИЧЕВ И.П.

Задача 11.3. Двухматричное соотношение Бутчера (про-
зрачное) $\left(\begin{smallmatrix} 2 & 0 \\ 0 & 1 \end{smallmatrix} \right)$ относительно локальной функции

Опираюсь на принцип новой экономики, мы проводим только, притягивая к работе метод снятия, когда человек находится в состоянии сознания.

[illegible]

**РИСУНОК 11.2. ПРИМЕР
ВЫПОЛНЕНИЯ ИСХОДЯЩЕГО
ИЗЧЕТНО-ЧЕТКОГО СЛИЯНИЯ
БЭТЧЕРА**

[illegible]

Предположим, что в первом подфайле содержится i нулей и j единиц во втором подфайле. Для выполнения этой задачи требуется рассмотреть четыре случая в зависимости от того, являются ли i и j четными или нечетными числами. Если оба параметра i и j имеют четные значения, то для подфайла слияния выполняется над двумя файлами. В первом файле содержится $i/2$ нулей, а во втором файл — $j/2$ единиц, так что в обоих случаях получившийся при слиянии файл содержит $(i + j)/2$ нулей. Выводом из этого является, получим сбалансированный файл типа 0-1. Файл типа 0-1 характеризуется тем, что i и j являются чётными, когда i — чётное, а j — нечётное число, а также когда i — нечётное, а j — чётное число. Но если оба i и j — нечётные числа, то мы завершаем эту процедуру тем, что делаем файл, содержащий $(i - 1)/2 + 1 = (i + 1)/2$ нулей с файлом, содержащим $(j + 1)/2 - 1 = (j - 1)/2$ единиц, следовательно, получивший файл содержит $(i + j - 1)/2 + 1 = (i + j + 1)/2$ нулей, следовательно, получивший файл содержит $(i + j + 1)/2 - 1 = (i + j - 1)/2$ единиц (см. рис. 11.3), и далее на комбинаторе на завершающей стадии заканчивая сортировку.

На самом деле нет необходимости доказывать это. И действительно, мы можем переписать программу 11.2 в 8.1 таким образом, чтобы на шаге они могли выполнять соответствующую процедуру для любого N , сортирующей функции сортировки и $split$ в том, чтобы они поддерживали индекс и осуществляют косвенные обращения к данным (см. упражнение 11.13). Либо мы можем заставить эту программу функционировать непосредственно, используя сравнения абстракции последующего применения к исходному массиву файла (см. упражнение 11.13). Мы можем использовать эти приемы для любого неадаптивного метода сортировки, который выполняет переупорядочивание данных при помощи операций обмена, сохраняя или не сохраняя. Но как только мы знаем, то структура этого алгоритма настолько проста, что мы сразу можем перейти к разработке эффективной рекурсивной реализации, в чем убедимся уже в разделе 11.2.

Упражнения

- 11.1. Покажите результаты слияния и обратного слияния алгоритма E A S Y Q L E S T I O N.
- 11.2. Обобщить программу 11.1 на случай, когда нужно выполнить k -путевое слияние и обратное слияние.
- 11.3. Реализовать операции слияния и обратного слияния без использования массива.

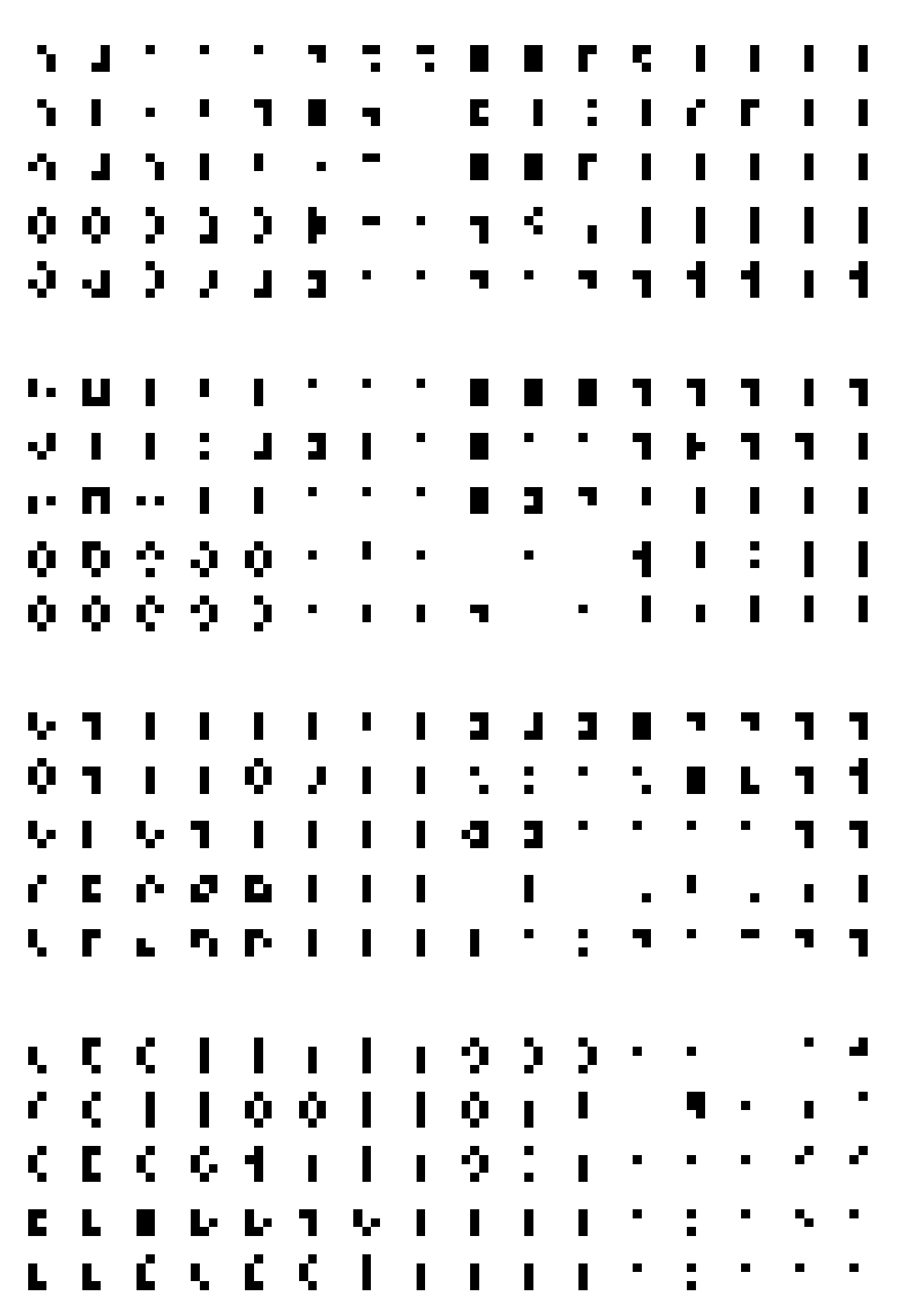


РИСУНОК 11.3. ЧЕТЫРЕ СЛУЧАЯ СЛИЯНИЯ ТИПА 0-1. Каждому из четырех случаев слияния по 2 строки: первая строка — i , вторая строка — j , третья строка — результат слияния, четвертая строка — результат обратного слияния. В первом случае: $i=0, j=0$, во втором: $i=0, j=1$, в третьем: $i=1, j=0$, в четвертом: $i=1, j=1$.

- 11.4. Показать, что наивысшая программа, которая сортирует N различных ключей, требует отсортировать $\Omega(N \log N)$ элементов, которые не обязательно различны.
- 11.5. Доказать, как много операций требуется, чтобы сортировать каждую из трех перестановок чисел 1, 2 и 3.
- 11.6. Приведите пример нестоящего алгоритма, который сортирует четыре элемента.
- 11.7. Доказать лемму 11.1. Кроме того доказать, что если программа не использует сортировку некоторого входного массива с произвольными ключами, то существует некоторая перестановка элементов из $0 \dots 1$, которую она также не может отсортировать.
- 11.8. Показать, как выполняется слияние ключей $A \in Q \leq T \leq R \leq U \leq S \leq P$ с помощью программы 11.2 в случае примера, представленного на диаграмме рис. 11.2.
- 11.9. Выполнить упражнение 11.8 для ключей $A \in S \leq U \leq N \leq Q \leq T \leq C$.
- 11.10. Выполнить упражнение 11.8 для ключей $1 \leq 0 \leq 1 \leq 1 \leq 0 \leq 0 \leq 0 \leq 1 \leq 0 \leq 0$.
- 11.11. Эмпирически измерить среднее время выполнения сортировки элементов 10^3 -элементного массива с помощью алгоритма, описанного в сортировке слиянием (программы 8.1 и 8.2) для $N = 10^3, 10^4, 10^5$ и 10^6 .
- 11.12. Предложите рекурсивные функции `sortRecn`, `shuffle` и `unshuffle`, которые реализуют программы 11.2 и 8.3 работы в режиме прямой сортировки (см. раздел 6.5).
- 11.13. Предложите рекурсивные функции `sortRecn`, `shuffle` и `unshuffle`, которые реализуют программы 11.2 и 8.3 работы для заданного N нестоящего алгоритма сортировки N элементов. Вы можете воспользоваться вспомогательными для быстрого доступа для отслеживания значений индексов.
- 11.14. Если мы представим таблицу на предназначенных для хранения и обработки данных, мы должны обеспечить (быстро) проверяемость в соответствии с оптимизацией, описанной в разделе 8.3. Внеся изменения в программу 11.2 так, чтобы он начинался с $1 \leq n \leq 1 \leq 1$, мы получим программу, которая соответствует внутренним предположениям. Доказать, как с помощью этого метода выполняются слияние ключей $A \in S \leq Q \leq U \leq T \leq R \leq N \leq P$ и стиль программы, показанный на диаграмме на рис. 11.2.
- 11.15. Доказать, что модификация программы 11.2, описанная в упражнении 11.14, способна выполнять любой список из произвольности.

11.2. Сети сортировки

Простейшей моделью для изучения неупорядоченных алгоритмов сортировки является абстрактная машина, которая способна осуществлять доступ к данным массива с помощью операции сравнения элементов. Такая машина называется *сетью сортировки* (*sorting network*). Сеть сортировки представляет собой систему из фиксированных (обычно схематично изображенных) для конкретного (конкретного), который специально можно легко изменить таким образом, что становится возможным выполнить любую сортировку общего вида.

На рис. 11.4 показана простая сеть, выполняющая сортировку четырех ключей. Отметим, что изображена сеть сортировки N элементов в виде последовательности N горизонтальных прямых линий, при этом каждый компаратор соединен с двумя линиями. Можно считать, что сортируемые ключи проходят по сети сверху вниз, при этом при необходимости происходит обмен парой чисел. В результате заданного набора из двух чисел получается пара порядк. чисел, которая на их пути входит в компаратор.

Необходимо снять целый ряд вопросов, прежде чем станет возможным построение реальной сортирующей машины, работающей по этой схеме. Например, не описан метод кодирования входной. В рамках данного подхода каждому элементу рис. 11.4 можно рассмотреть две группы данных: каждая такая группа несет один бит данных. следовательно, все биты в сети рассределяются по линиям одновременно. Другой выход заключается в том, что данные поступают на вход компараторов по одной линии поочередно, т.е. 1-й бит за единицу времени (перпигма) идет на первый аппаратный бит. Кроме того, очевидно не затрагивая вопрос синхронизации, должны быть предусмотрены механизмы, препятствующие обратному компараторов, прежде чем поступит заданные данные. Если сортировка представлят собой полезную абстракцию, поскольку они позволяют нам сделать четкий разграничение между простыми задачами. Более того, такой парадигма, такая как, например, минимизация числа компараторов. Более того, как мы убедились в разделе 11.3, абстрактное понятие сети сортировки может быть с пользой применено и в приложениях, отличных от построения электронных схем различного назначения.

Еще одним важным применением сетей сортировки является задача параллельных вычислений. Если два компаратора не используют один и тех же линий для ввода данных, то мы понимаем, что они могут работать одновременно. На рисунке сети, изображенной на рис. 11.4, показываем, что четыре элемента могут быть элементарными из трех параллельных шагов. Компаратор 0-1 и компаратор 2-3 могут работать одновременно на первом шаге, после чего компаратор 1-2 и компаратор 0-3 могут одновременно работать на втором шаге, а компаратор 2-3 завершает сортировку на третьем шаге. Для такой заданной сети требуется сгруппировать компараторы в последовательности параллельных каскадов (этапы) такой, каждый из которых состоит из компараторов, которые могут работать одновременно тем. выражением (11.7). Что бы параллельные вычисления были эффективными, нашей задачей становится разработка сетей с минимально возможным числом параллельных каскадов.

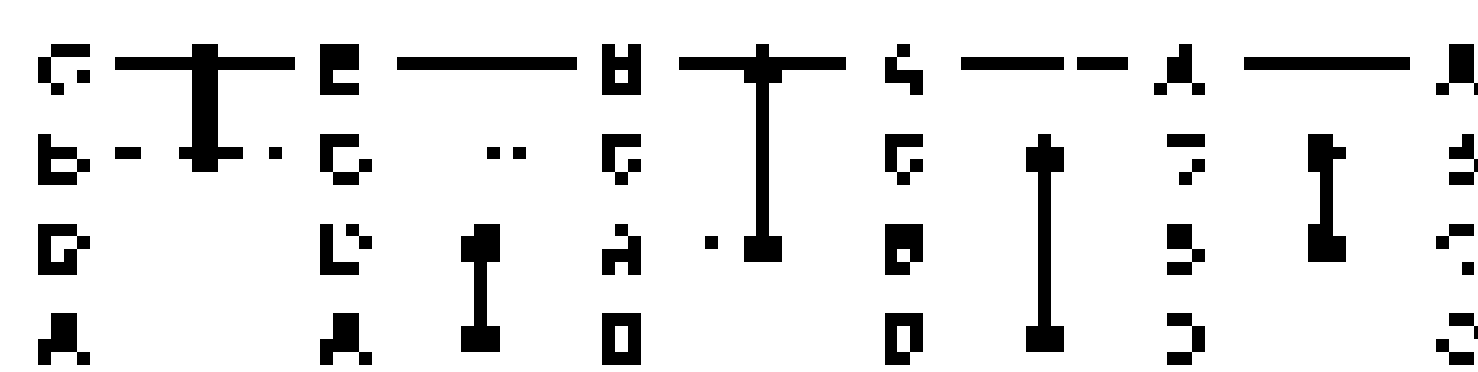


РИСУНОК 11.4. СЕТЬ СОРТИРОВКИ

Каждый элемент сети сортировки может быть элементарным из трех параллельных шагов. Компаратор 0-1 и компаратор 2-3 могут работать одновременно на первом шаге, после чего компаратор 1-2 и компаратор 0-3 могут одновременно работать на втором шаге, а компаратор 2-3 завершает сортировку на третьем шаге. Для такой заданной сети требуется сгруппировать компараторы в последовательности параллельных каскадов (этапы) такой, каждый из которых состоит из компараторов, которые могут работать одновременно тем. выражением (11.7). Что бы параллельные вычисления были эффективными, нашей задачей становится разработка сетей с минимально возможным числом параллельных каскадов.

Программа 11.2 непосредственно соответствует эти сведения для каждого N , в то же время как можно ознакомиться с прямой восходящей структурой, изображенной на рис. 11.5. Чтобы построить сеть слияния размера N мы воспользуемся двумя копиями сети размером $N/2$; одна из них предназначена для линий с четными номерами, другая — для линий с нечетными номерами. Поскольку соответствующие два набора компрессоров не пересекаются, мы можем разместить их таким образом, чтобы эти сети чередовались. После этого мы разместим компрессоры между линиями 1 и 2, 1 и 4 и т.д. Чередование четных и нечетных линий гарантирует выполнение частотного т. программы 11.3. Довольство того, что эти сети выполняются слияние данных образом, аналогично доказательству лемм 11.1 и 11.2, для которых применялся принцип гусей и елки. На рис. 11.6 показан пример выполнения этого типа

Программа 11.3. вышел из вышедшей реализации программы Батлера без операции тасования, спотыкнувшись о сетям, представленным на рис. 11.3. Эта программа представляла собой компактную и эффективную обертку (не использующую дополнительного пространства оперативной памяти) функции снятия, которую, к сожалению, легче понять, если считать ее чередуемыми представленной сетью, а то же время прямо доказательство того, что она правильно решает задачу снятия, само по себе очень интересно. Мы проведем исследование одного такого доказательства в конце этого раздела.

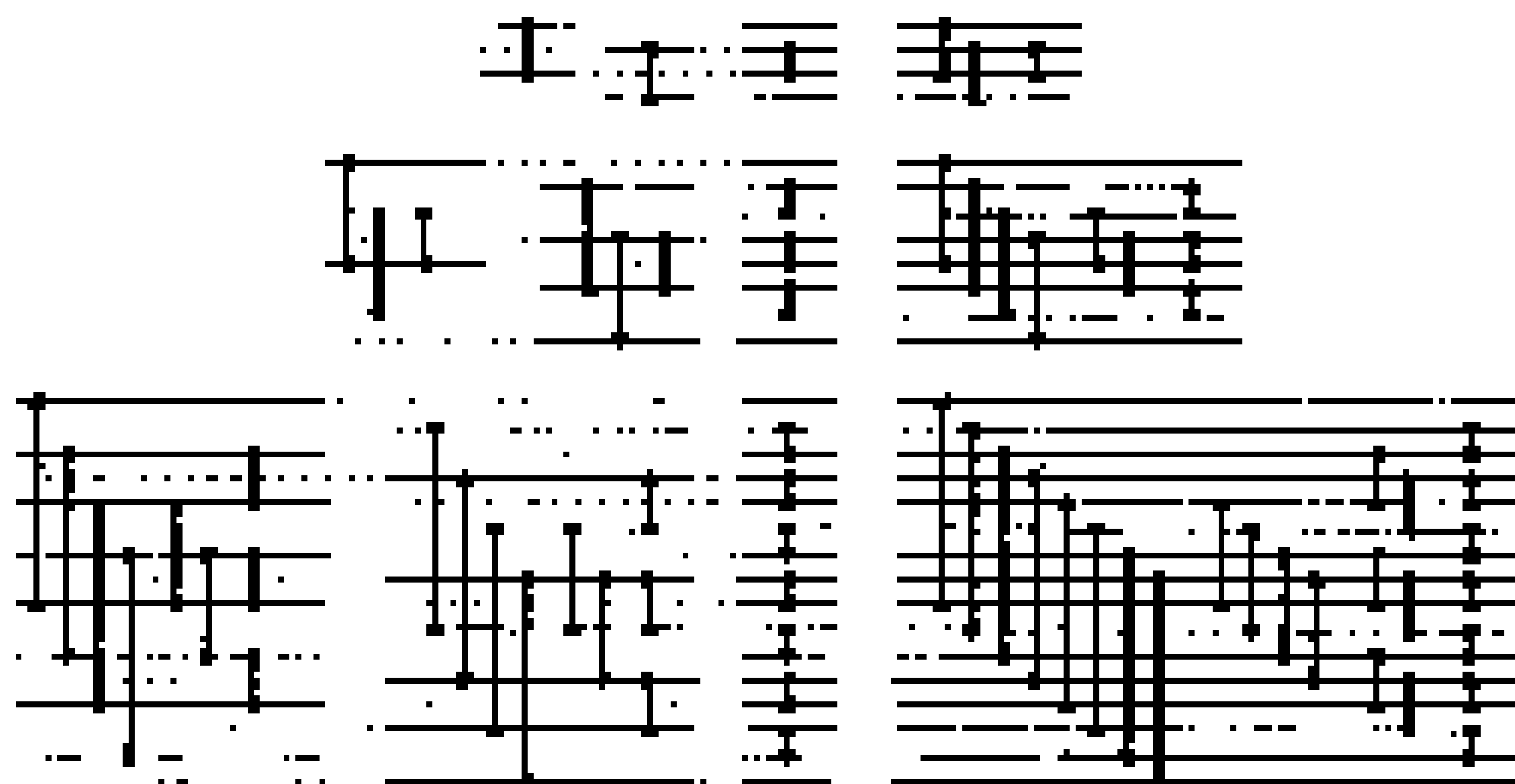


Рисунок 115. НРЧЕТНО-ЧЕТНОЕ СЛЫШАНИЕ ВЗРАСТА

[illegible]

Рисунок 11.7 служит иллюстрацией связи нечетно-четной сортировки Бэтчера (описанной на основе сетей слияния, представленной на рис. 11.5) с использованием стандартной конструкции рекурсивной сортировки слиянием. Эта конструкция дважды рекурсивна: один раз для сетей слияния, другой раз — для сетей сортировки. И хотя они не оптимальны — мы вскоре рассмотрим оптимальные сети — тем не менее, они достаточно эффективны.

Программа 11.3. Нечетно-четное слияние Бэтчера (нерекурсивная версия)

Данная реализация нечетно-четного слияния (которая гарантирует, что размер файла N нечетен: $N \bmod 2 = 1$) является $\Theta(N \lg N)$ и является быстрой, так как она принадлежит процедуре слияния, в отличие от выбора, в которой слияние осуществляется рекурсивной версией (см. программу 11.2 и рис. 11.5). Она работает для любого $\lg N$ прохода, представленного единообразными и нехарактерными командами слияния-объема.

```
template<class T>
void merge(int* a, int l, int m, int r)
{
    int k = -1; // указатель, что
                // k/2 == 1/2
    for (int i = m; i < r; i += 2)
        for (int j = l; j < m; j += 2)
            swap(a[j+1], a[i+1+k]);
}
```

Лемма 11.3 Сети нечетно-четной сортировки Бэтчера имеют глубину приблизительно $\frac{N}{4} (\lg N)^2 / 4$ и имеют размер $\frac{N}{2} (\lg N)^2 / 2$ и являются оптимальными.

Сеть слияния требует выполнения около $\lg N$ дополнительных шагов, а сети сортировки требуют выполнения $1 + 2 + \dots + \lg N$ или примерно $\frac{N}{2} (\lg N)^2 / 2$ дополнительных шагов. Расчет компьютерной сложности численно в качестве выражения (см. упражнение 11.24).

Непосредственно функция слияния и программа 11.3 в рамках стандартной сортировки слиянием, представленной программой 8.3, позволяют получить компактный именный неадаптивный метод сортировки, который требует $\frac{N}{4} (\lg N)^2$ операций слияния. С другой стороны, мы можем заменить рекурсию на сортировку слиянием и не посредственно реализовать предложенную версию в полном ее объеме, как показано в программе 11.4. Как и в случае программы 11.3, эту программу легче понять, если рассмотреть ее как чередующиеся проходы сетей, показанных на рис. 11.7.

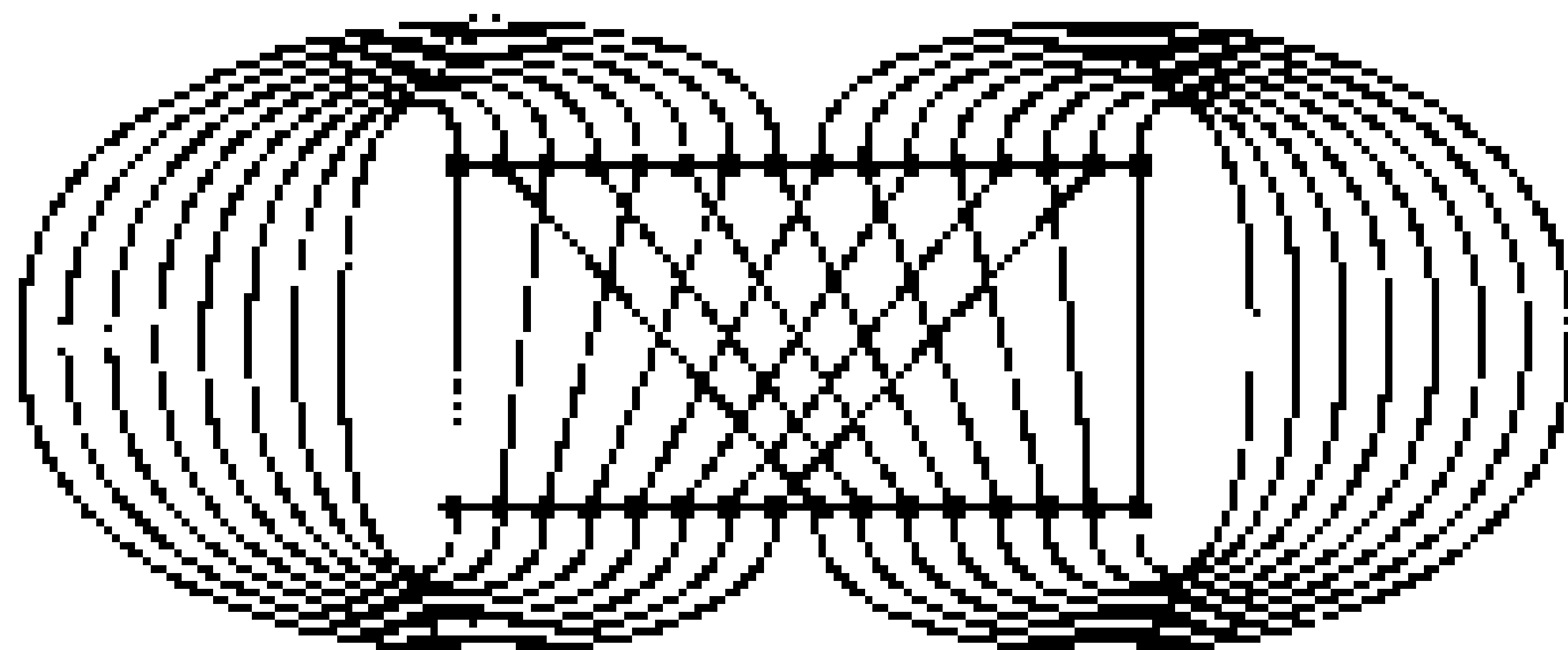


РИСУНОК 11.6 ПРИМЕР НЕЧЕТНО-ЧЕТНОГО СЛИЯНИЯ БЭТЧЕРА.

Для удобства все операции выполняются, для каждого элемента слияния Бэтчера применительно к каждому элементу выполняется 25 операций слияния-объема, выполняемых на этапе дупликации. Мы также должны отметить, что для каждого элемента слияния Бэтчера требуется 25 операций слияния-объема.

РИСУНОК 11.10. МАШИНА ИДЕАЛЬНОГО ТАСОВАНИЯ

Иллюстрация на этапе
распределения элементов
по ячейкам памяти. В ячейки
случайно распределяются
элементы исходного массива
(предполагая, что элементы
даны). Подлежащий сорту
массив делится на две половины
параллельно ключевым элементам.



репрезентовать эти, подобную показанной в нижней части рис. 11.8. Такая машина на каждом шаге выполняет предусмотренные рассматриваемым алгоритмом операции сравнения обмена на некоторой паре смежных папцессоров, после чего осуществляет полное перемешивание данных. Программирование работы этой машины сводится к определению, какие пары процессоров должны выполнить операции сравнения-обмена на каждом шаге.

На рис. 11.11 показаны динамические характеристики как возникающего метода, так и версии нечетко-четкого алгоритма Ботчера с полным перемешиванием.

Тасование — это важная абстракция, описывающая движение данных в алгоритме, построенном по принципу "разделяй и властвуй", и он применяется в различных задачах, отличных от сортировки. Например, квадратная матрица размером 2^k -на- 2^k хранится в развернутом по строкам виде, затем с помощью тасовок транспонируется эту матрицу (приводит матрицу к развернутому по столбцам виду). В яслях более важных примеров входят быстрое преобразование Фурье и полиномиальное преобразование (см. часть В). Мы можем решить каждую из этих задач, используя тот же самый логический механизм идеального тасования, подобный показанной на рис. 11.10, но с гораздо более мощными процессорами. Мы можем даже обойтись без них с использованием универсальных процессоров, способных выполнять прямо и обратное тасование (некоторые из алгоритмов для этого были даже разработаны для практического применения); мы вернемся к обсуждению параллельных машин данного типа в разделе 11.5.

Упражнения

- 11.16. Дать примеры сетей сортировки четырех (см. упражнение 11.6), пяти и шести элементов. Использовать минимально возможное число компараторов.
- 11.17. Написать программу, способную подсчитать число сравнений шагов, необходимых для выполнения любой заданной сортирующей программы. Совет: Воспользоваться возникающей стратегией решения задачи. Сначала входные данные для назначения к каждому из элементов для каждого компаратора выполните следующие действия: отметить обе входные линии как входные для компаратора $i + 1$, если четная одна из входных линий есть i , и отметить другую линию во входных i .
- 11.18. Сравнить время выполнения программы 11.4 с аналогичным показателем для программы 11.3 для случайно упорядоченных ключей при $N = 10^4, 10^5, 10^6$ и 10^7 .
- 11.19. Нарисовать сеть Ботчера, ориентированную на самый тип 10-а-1-1.

• 11.20. Доказать существование взаимосвязи между размером обратным тасовщиком и тасовщиком, представленным на рис. 11.3

• 11.21. Из положения в тексте следует, что на рис. 11.3 в ячейках были представлены 11 слоев сортировки 21 элемента. Нарисуйте схему, которая содержит минимальное число компараторов.

11.22. Найдите число компараторов в нечетно-четной сети сортировки Бэтчера для $2 \leq N \leq 32$. Как сеть, когда N не делится на 2, строится на базе схемы N ячеек сети, построенной для нечетной степени 2, превосходящей N .

• 11.23. Для $N = 2^k$ вывести точное выражение для определяющего числа компараторов, используемых в нечетно-четной сети сортировки Бэтчера. Совет: Проверьте свой ответ на данных рис. 11.3, которые показывают, что в сети имеется 1, 2, 4, 25 и 65 компараторов для N равно 2, соответственно, 4, 8, 16 и 32.

• 11.24. Постройте сеть для сортировки 21 элемента, выбрав для этой цели подходящий рекурсивный стиль. Когда сеть размером N представляется собой комбинацию сетей размером $N/2$ и $N/3$, то сортировка будет сложной. (Вспомогательные результаты упражнения 11.19 и в частности теорема 11.4.)

11.25. Воспользуйтесь рекурсивными соотношениями для расчета числа компараторов в сети сортировки. Проверьте, как указано в упражнении 11.24 для $2 \leq N \leq 32$. Сравните полученные результаты с результатами, полученными в упражнении 11.22.

• 11.26. Найдите 16-ячейковую сеть сортировки, которая не требует меньше числа компараторов, чем сеть Бэтчера.

11.27. Изобразить сеть слияния, соответствующую рис. 11.8, для быстрой сортировки. Используя схему, она связана с упражнением 11.14.

11.28. Нарисовать сеть сортировки, соответствующую структуре Шелла с приведенными Пратом (см. раздел 6.6) для $N = 32$.

11.29. Нарисуйте таблицу, содержащую число компараторов в сети, описанных в упражнении 11.28, в числе компараторов в сети Бэтчера для $N = 16, 32, 64, 128$ и 256.

11.30. Разработать сеть сортировки, эквивалентную быстрой сортировке 3-х-связной или 4-связной бинарной файлом на N элементов.

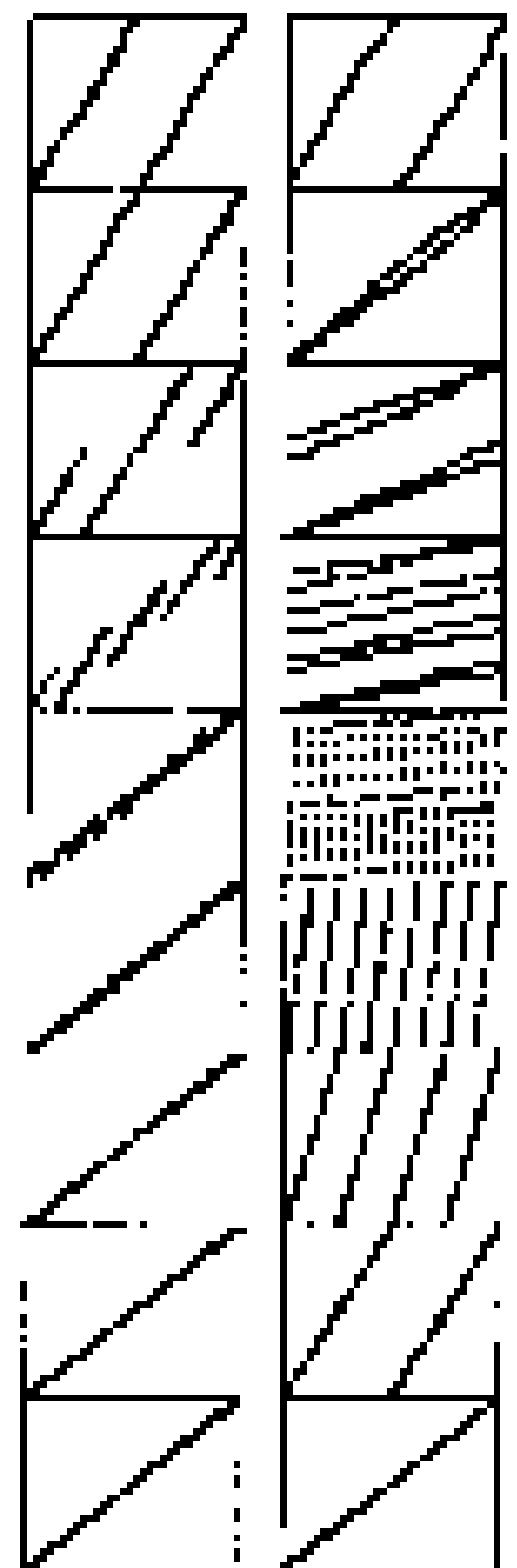


РИСУНОК 11.11.
ДИНАМИЧЕСКИЕ
ХАРАКТЕРИСТИКИ
НЕЧЕТНО-ЧЕТНОГО
СЛИЯНИЯ.

Вспомогательная схема
для нечетно-четного слияния
(рис. 11.11) показывает
различные состояния
массива, возникающие
в процессе сортировки.
Слева показан исходный
массив, состоящий из
элементов, которые
будут сортированы.
Справа показан массив
после первого шага
слияния, когда элементы
уже частично упорядочены.
Центральная часть
показывает промежуточные
состояния массива во
время выполнения
алгоритма.

- 11.31. Воспользуйтесь сетью из упорядоченных 11 30 для разработки схемы подобной алгоритму Прата, на базе чисел, кратных 3 и 4. Вычертите полученную сеть для $N = 32$ и решите упражнение 11.29 исключительно к этой сети.
- 11.32. Вычертите версию четной четной сети сортировки Батчера для $N = 16$, в которой каждой шестичисловой выполняются между каскадами перемешивных компараторов, соединяющих смежные линии (четвертая концевая каскадная сеть этой сети должны быть каскады из сети шестичисловой четной четной рис. 11.8).
- 11.33. Написать программу для машины, изображенной на рис. 11.10, выполняющей следующие сравнения: каждая инструкция есть последовательность из 13 бит, при этом 4-й бит, $1 \leq i \leq 13$, покрывает биты от разряда i до разряда i и процессор $i - 1$ должен выполнить операцию сравнения байтов. Программа представляет собой последовательность инструкций, а машина выполняет каскадное сравнение между каждыми двумя инструкциями.
- 11.34. Написать программу сортировки для машины, изображенной на рис. 11.10, выполняющей сравнения, сформированные в упражнениях 11.31.

11.3. Внешняя сортировка

Мы переходим к рассмотрению другого аспекта задачи абстрактной сортировки, которая заключается, когда сортируемый файл настолько велик, что не помещается целиком в оперативной памяти компьютера. Для описания такой ситуации мы используем термины машины сортировки гофрированной бумаги. Существует множество различных типов устройств внешней сортировки, которые накладывают различные ограничения на элементарные операции, применяемые при реализации таких видов сортировки. Кроме того, будет полезно изучить методы сортировки, использующие для промежуточных базисов операции: операция считывания (зада) данных из первичного запоминающего устройства в оперативную память и операция записи (отда) данных из оперативной памяти на внешнее запоминающее устройство. Мы считаем, что стоимость этих двух операций настолько выше стоимости простейших примитивных элементарных операций, что последние в дальнейшем мы можем полностью игнорировать. Например, в условиях задачи абстрактной задачи мы можем позволить себе не учитывать стоимость сортировки в оперативной памяти. При огромных размерах оперативной памяти и неэффективных методах сортировки подобный подход может оказаться невыгодным, но в общем случае на практике при необходимости сортировки можно учесть эти затраты при расчете общей стоимости внешней сортировки.

Многообразие типов простых запоминающих устройств и их стоимость часто разработку методов внешней сортировки в зависимости от состояния технологии на текущий момент. Эти методы могут оказаться почти сложными, многие параметры оказывают влияние на их рабочие характеристики; конечно может случиться так, что искусные методы могут оказаться неоптимальными и непрактичными только в силу тех или иных технических ограничений. По этой причине в данном разделе мы не будем заниматься проблемами разработки конкретных реализаций, а сосредоточимся лишь на изучении общих принципов внешней сортировки.

Довольно часто жесткие требования, предъявляемые к методам доступа к данным в зависимости от типа внешней системы, оказываются более сложным фактором, чем

малой стоимости операций чтения данных. Например, для большинства типов устройств операция чтения и запись из внешнего запоминающего устройства в оперативную память в общем случае значительно эффективнее выполняется в виде крупных блоков данных. Именно этого, как shown устройству, обладающего громадными возможностями, главным образом требуется в таких расчетах, что максимальная производительность еще достигается в тех случаях, когда доступ к таким блокам данных осуществляется последовательно. Например, мы не можем прочитать элемент данных, хранящийся в конце магнитной ленты без того, чтобы не проскочить все элементы, хранящиеся в начале ленты — на практике доступ к элементам данных на магнитной ленте ограничивается доступом к тем элементам, которые расположены в некоторой окрестности элемента доступа к которым осуществляется чаще, чем к остальным. Некоторые из современных технологий предлагают тем же свойствам. В связи с этим, в данной главе мы рассмотрим некоторые из методов, которые позволяют эффективно считывать и записывать крупные блоки данных, откуда непосредственно следует, что алгоритмы сортировки регулярного вида доступа к данным могут быть достигнуты на машинах с устройствами тех типов, которые представляют для нас интерес.

Когда мы выполняем процесс считывания или записи некоторого числа данных файлов, мы имеем, что эти файлы находятся на различных внешних запоминающих устройствах. На старых вычислительных машинах, в которых данные хранятся на магнитных лентах, это предположение было необходимым требованием. При работе с магнитными дисками становится возможной реализация алгоритма, который, согласно замыслу, использует существование внешнего устройства, но в общем случае работы с несколькими устройствами остается более эффективной.

Первым шагом при создании высокоэффективной программы сортировки файлов сверхбольшого размера является создание копии файла. Вторым шагом может быть представление исходного файла в обратном порядке. Как трудно, с которыми приходится сталкиваться при решении этих задач, возникает и при реализации внешней сортировки. (В процессе сортировки иногда приходится выполнять одну из этих операций.) Целью построения абстрактной модели является то, чтобы отделить проблемы построения программной реализации от проблемы разработки алгоритма.

Рассмотренные алгоритмы сортировки представляют собой некоторую последовательность проходов по всем данным, и мы обычно говорим о сложности того или иного метода внешней сортировки путем подсчета числа таких проходов. В общем случае нам требуется значительно меньшее число проходов — десять или, возможно, меньше. В силу этого обстоит дело уменьшение этого числа проходов хотя бы на один существенно улучшает рабочие характеристики алгоритма. Основное наше предположение заключается в том, что мы предполагаем, что основную долю времени выполнения конкретного метода внешней сортировки составляют операции прохода и подпада данных. Следовательно, мы можем игнорировать время выполнения внешней сортировки, учитывая только существование ее проходов во время, необходимое для считывания и записи всего файла.

Конечно говоря, абстрактная модель сортировки, которой мы в дальнейшем будем пользоваться, построена на предположении, что сортируемый файл состоит из

чтобы полностью понять, как в оперативной памяти, к чему он определяет значения двух других параметров времени выполнения сортировки (число проходов по данным) и количество используемых внешних устройств. Мы полагаем, что нам легче рассмотреть их по отдельности.

- M записей на внешнем устройстве, эфире, который уже должен был выполняться
- пространства оперативной памяти, достаточное для размещения M записей
- $2P$ внешних устройств, которыми мы сможем попытаться во время сортировки.

Мы рассмотрим только 3 шагового устройства, из которых находится файл данных данных. В метки 1, 2, ..., $2P - 1$ тем самым обозначены внешние устройства. Если сортировка выполняется в том, чтобы возразить запись на устройство 1 в асимметричном виде. Как мы увидим далее, существует некоторая зависимость между P и скоростью работы $P = O(1)$ (то есть) сортировки — мы вынуждены в том, чтобы получить эту зависимость в численном выражении с тем, чтобы иметь возможность фрагментировать алгоритмические стратегии.

Существует несколько причин, в силу которых эта идеализированная модель может не соответствовать действительности. Тем не менее, как и всякая хорошая абстрактная модель, она отображает наиболее важные аспекты реальной ситуации, и это обеспечивает понимание только ранжи. Задача, которую можно производить, несомненно, является достаточно сложной задачей, которая применяется непосредственно в практических ситуациях.

Большая часть методов внешней сортировки основывается на следующих принципах общей организации. Мы начинаем с первого прохода по сортируемому файлу, в процессе которого производится его разбиение на блоки, размер которых примерно соответствует пространству оперативной памяти, после чего выполняется сортировка этих блоков. Затем осуществляется чтение отсортированных блоков, при необходимости с этой целью выполняются несколько проходов по файлу, при этом с каждым проходом степень упорядоченности возрастает, пока весь файл не окажется отсортированным. Такая процедура называется сортировкой-случаем (sort-merge), она в основном применяется на практике с тем пор, когда компьютеры получили широкое распространение в коммерческих учреждениях с пятидесятых годов прошлого столетия.

Простейшая стратегия сортировки-случаем, описанная названными аббревиатурами, заключается в том, что (basic merge sort), показана на рис. 1.1.2. Этот метод состоит из прохода, осуществляющего начальный разделение (initial distribution), за которым следует несколько проходов многоэтапного слияния (multiway merge pass).

На начальном проходе мы осуществляем распределение входных данных (то есть) во внешнем устройстве P , $P = 1, \dots, 2P - 1$ в виде отсортированных блоков данных, каждый из которых содержит M записей (с исключением, возможно, исключительного блока, который может не состоять, если M не кратно M). Такое распределение не трудно выполнить — мы считываем первые M записей с устройства ввода, сортируем их в отсортированном порядке (то есть) блок данных на устройстве P , затем считываем следующие M записей с внешнего устройства, сортируем их и записываем в отсортированный блок на устройстве $P + 1$ и т.д. Если, исключив устройство $2P - 1$, у нас все еще остаются несортированные данные (т.е., если $M > PM$), мы продолжаем на устрой-

[illegible]

РЭСУНОК 7.12

[illegible]

ство $P + 1$ второй опосредованный блок и т.д. до тех пор, пока весь шифр не будет извлечен. Но жерновная процедура распределения количества опосредованных блоков, размещаемых на каждом устройстве, только выполняется M/M , округляясь до ближайшего целого числа в сторону уменьшения или увеличения. Если M есть кратное M , то размеры всех блоков одинаковы и равны M/M (один и тот же, то же блок, за исключением последнего, разны M/M). Для небольших M числа блоков может оказаться меньше P , ввиду чего одна или более членов набора устройств могут оказаться пустыми.

На первом многопутевом проходе, осуществляемом заново, мы рассматриваем устройство в интервале от M до $2M - 1$ как входные, а устройство в интервале от 0 до $M - 1$ как выходные. Мы осуществляем M -кратное сдвигание блока данных размером M , помещенных на входные устройства, в результате которого получим M групп данных размером M , которые затем помещаем на выходные устройства с соблюдением условия максимального возможного блочного. Прежде всего, мы проводим сдвигание первых блоков с каждого входного устройства и помещаем результат этого сдвига на устройство 0 , затем мы помещаем результат сдвига второй блок на каждом входном устройстве на устройство 1 и т.д. По достижении устройства $M - 1$ мы далее помещаем второй блок данных на устройство 0 , затем третий блок на устройство 1 и т.д. Мы продолжаем этот процесс до тех пор, пока все входные данные не будут начерчены. По завершении процедуры распределения данных шифроразмещенных блоков на каждом устройстве равно $M'_i(2M)$, которое до

Ближайшего к началу числа n в сторону увеличения или уменьшения. Если N есть крайнее PM , то все близкие к нему размер PM (в противном случае исключительный блок имеет маленький размер). Если N не больше PM , то получится один отсчитываемый блок (по отношению O) и на этом содержимое закончится.

В протянутом случае мы построили стат. процесс и выполняем вложенный процесс: итерационного сближения рассматриваемых устройств $Q_1, 1, \dots, P-1$ как входные устройства, а устройства $P, P-1, \dots, M-1$ как выходные. Мы выполняем P -итерацию сближения с целью получить на отсортированных блоках размера $P \times M$, полученных из входных устройств, блоки размера $P^2 \times M$ с заданным их размещением на входных устройствах. Циклическая заданная $P \times M$ завершении второго прохода. Результат помещается на устройство P , если M не больше $P^2 \times M$.

Продолжая таким образом, перемещаясь между устройствами от 0 до P , с одной стороны, и между устройствами от $P-1$ до $2P-1$, с другой, мы увеличим размер блока в P раз при помощи P путей го слития, пока в конечном итоге не получим один блок на устройство 0 или на устройство P . Завершающее слияние на каждом проходе может не быть P -путным слиянием, т. е. полным слиянием этого типа, но если это так, то это хорошо объяснено.

Рисунок 11.13 служит иллюстрацией того, что происходит, если число криволинейных только количественно и относительно размеры проходов. Оценка стоимости стрижки мы получаем, выполняя операции умножения, указанные в предоставленной на рисунке таблице, суммируя результаты (без учета нижней строки) и для суммы по числу стрижки каждого графа. Эти вычисления выражены каждая через число проходов по данным.

Чтобы выполнить Р-путевое планирование, мы можем воспользоваться очередью по приоритетам размером R . Мы хотим постоянно иметь на выходе записанный элемент из числа тех, которые еще не поступили из входа, из которого на R отсчитываем блок, подходящий к выводу, с последующей записью каждого элемента на выходе следующим элементом к тому же блоку. Чтобы завершить эту процедуру, мы должны иметь индекс устройства с помощью операции `device()`, считывающей значение ключа существующей записи, которая должна быть связана с указанным устройством (и порождающей служебную метку, которая применяется по значению того ключа, относящегося к достижению конца блока). Само значение представляет собой простой диск, который сдвигает середину записи с устройства, внешнего минимальный ключ, а

0	1	2	3	4	5
15+1					
			5+1	5+1	5+1
2+3	2+3	1+0			
			1+3	1+6	
17+5					

РИСУНОК 11.13. РАСПРЕДЕЛЕНИЕ ПРОХОДОВ В 3-ПУТЕВОМ СБАЛАНСИРОВАННОМ СЛИЯНИИ.

В примере анализа рассматривается распределение параметров сбалансированной сортировки с шагом 3-х раз, разность значений δ 1,5 раз превосходит максимальную длину очереди, минимальная длина очереди равна 1, максимальная 4, 5 и 6 раз величина разности δ , 1 и 2 являются минимальными, 3 и 4 являются стандартными, 5 и 6 являются средними, разность 7 является максимальной. Тогда на минимальном уровне очереди, соответствующем разности 0, 1 и 2, и самым меньшим разностям на разности 3, 4 и 5, самое большее приходится по три прохода. На следующем уровне очереди, соответствующем разности 6, 7 и 8, самое большее приходится по два прохода. На разности 9, 10 и 11, самое большее приходится по одному проходу. Таким образом, в 3-путевом сбалансированном слиянии, с шагом 3-х раз, разность значений δ 1,5 раз превосходит максимальную длину очереди, минимальная длина очереди равна 1, максимальная 4, 5 и 6 раз величина разности δ , 1 и 2 являются минимальными, 3 и 4 являются стандартными, 5 и 6 являются средними, разность 7 является максимальной. Тогда на минимальном уровне очереди, соответствующем разности 0, 1 и 2, и самым меньшим разностям на разности 3, 4 и 5, самое большее приходится по три прохода. На следующем уровне очереди, соответствующем разности 6, 7 и 8, самое большее приходится по два прохода. На разности 9, 10 и 11, самое большее приходится по одному проходу. Таким образом, в 3-путевом сбалансированном слиянии, с шагом 3-х раз, разность значений δ 1,5 раз превосходит максимальную длину очереди, минимальная длина очереди равна 1, максимальная 4, 5 и 6 раз величина разности δ , 1 и 2 являются минимальными, 3 и 4 являются стандартными, 5 и 6 являются средними, разность 7 является максимальной.

переходит эту запись на ahead, после чего помещает эту запись в очередь по приоритету на следующую запись с той же устройством, продолжая такую процедуру. Часть действий до тех пор, пока текущая метка не станет наименьшей в очереди по приоритету. Мы можем воспользоваться реализацией сортирующей очереди, чтобы сделать прототип алгоритма для очереди по приоритету пропорциональным $\log R$, но обычно R так мало, что эта запись кажется не важной по сравнению с операцией записи на постоянное устройство. А нашей абстрактной моделью игнорируем запись на постоянное устройство по приоритету и предполагаем, что в нашей реализации имеется эффективный метод последовательного доступа к данным по произвольному устройству. Благодаря чему мы можем избежать любой выкладки путем подбора числа проходов по данным. На практике мы можем воспользоваться элементарной манипуляцией очереди по приоритету и реализовать все функции, связанные с созданием программы, обеспечивающих максимально возможную производительность нашего устройства.

Лемма 11.4. Для метки 2^k требуется k проходов и ограниченное количество, достаточное для реализации M записей сортирующей очереди, в случае которой анализ R -элементов обобщенного списка, требует $O(\log R)$ проходов.

Один проход нужен для распределения. Если $M = 2^k$, то блок получает размер $M/2$ после первого слияния, $M/4$ после второго, $M/8$ после третьего и т.д. Сортировка заканчивается после того, как будут выполнены $k = \log_2 (M/2)$ проходов. В противном случае, если имеет место условие $M/2^{k+1} < M \leq M/2^k$, эффект неполных и пустых блоков приводит к появлению разницы в размерах блоков блока k концу процесса, тем не менее, он ограничен после выполнения $k = \lceil \log_2 (M/2) \rceil$ проходов.

Например, если мы хотим отсортировать 1 миллиард записей, имея для этой цели шесть внешних устройств и ограниченное количество, позволяющее разместить 1 миллиард записей, мы можем решить эту задачу через трехступенчатую сортировку-слияние, которую мы в общем случае рассматриваем по лемме — одна проход для начального распределения и $\lceil \log_2 (1000) \rceil = 7$ проходов слияния. После выполнения прохода начального распределения мы получим отсортированные блоки данных, хранящиеся по 1 миллиарду записей, 9 миллиардов записей в каждом блоке после первого слияния, 9 миллиардов записей в каждом блоке после второго слияния, 17 миллиардов записей в каждом блоке после третьего слияния и т.д. Можно подсчитать, что на сортировку файла уходит в 10 раз больше времени, чем на то, чтобы просто получить его копию.

Наиболее важное решение, которое приходится принимать на практике в процессе сортировки связано с тем, какой размер R , т.е. гравитация слияния. А используя небольшую абстрактной модели последовательный метод доступа к данным не является предположением unreasonable, откуда следует, что R должно быть равно по крайней мере доступным для нас внешним устройствам. Такая модель реализуется для многих внешних запоминающих устройств. Если всего лишь небольшое число внешних устройств может быть использовано для сортировки, необходимо использовать методы доступа, отличные от последовательных. В таких случаях мы все еще имеем возможность использовать многопутевое слияние, но при этом должны учесть влиятельную особенность, заключающуюся в том, что внешние

показателя P приводит к уменьшению числа переходов, но и та же мера увеличивает количество сформированных связей между соседними узлами.

УДОЗНАЮЩАЯ

- 11.35. Как выглядит, как кодируется и как декодируется строка "K E Y S сортируются по возрастанию 3 путевых сбалансированных значения в строке примера, представленного на рис. 11.12.
- 11.36. Как отразится на числе проходов многопутевого сбалансированного файла, если число уровней уменьшится для каждого принципа увеличения удвоения?
- 11.37. Как отразится на числе проходов многопутевого сбалансированного файла, если число уровней увеличится в 10 раз для каждого принципа увеличения удвоения?
- 11.38. Разработать интерфейс для хранения этого и любого, который предусматривает независимую передачу данных в внешние устройства, работающие независимо (для детализации ключей существующих в данной системе). Использовать этот интерфейс для реализации P -путевого слияния, при этом P есть некоторая произвольная константа, но P должно быть не меньше 2. Записи в каждой строке должны размещаться на различных выходящих устройствах. Сравнение файлов выполняется поочередно, применяя P ключей, необходимых для копирования файла на выходящее устройство, один за другим.
- 11.39. Использовать интерфейс на упражнении 11.38 при написании программы на языке парадигмы обратный для файла максимально допустимого в данной системе размера.
- 11.40. Как вы будете выполнять идеальное сбалансированное кодирование на выходящем устройстве?
- 11.41. Разработать статическую модель многопутевого слияния, которая охватывает алгоритмы, способные взаимодействовать с одним файлом на другом на одном и том же устройстве с гарантированной стоимостью переключения, которая равна нулю или близка к нулю.
- 11.42. Разработать модель к внешней сортировке, который основан на разложении, сбалансированном попарному и быстрой сортировке или в переставочной сортировке MFD (ссылка на старший индекс), сбалансированном с его сравнением с многопутевой сортировкой. Вы можете перейти на более темные уровни абстракции, но вы должны быть при описании сортировки сбалансированной в этом смысле, однако вы должны оставаться в теме. Файлы будут взаимодействовать в соответствии с тем, как они взаимодействуют с устройством и выходящим устройством.
- 11.43. Как вы будете сортировать сбалансированное значение устройства, если любые другие внешние устройства являются для абстракции (можно рассмотреть только операционной системы)?
- 11.44. Как вы будете сортировать содержимое внешнего устройства, если это невозможно для всего лишь еще одно устройство (а также сбалансированная запись)?

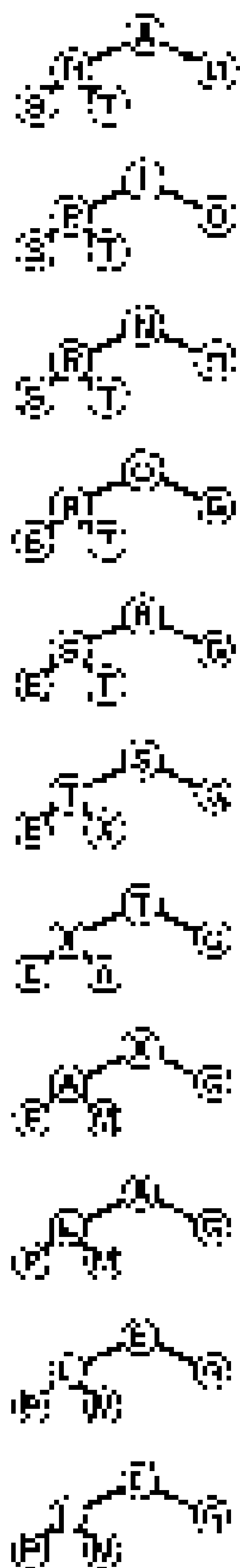
11.4. Различные реализации сортировки-слияния

Общая стратегия этой программы, описанной в разделе 11.12, доказала свою эффективность на практике. В этом разделе мы рассмотрим два усовершенствования этой стратегии, позволяющие снизить объем затрат. Первое из них, метод майора и миноранта (majority-minority method), гарантирует получение того же эффекта, что и объем просмотра используемой оперативной памяти, следующий метод, метод двойного сканирования (double-sweep method), обеспечивает тот же эффект, что и выделение чисел используемых нами устройств.

В разделе 11.1 мы описали применение очереди по приоритетам для P -узлов системы, но при этом заявили о том, что P настолько мало, что усовершенствование алгоритма с целью повышения быстродействия функционально неуместны. Тем не менее, во время начального распределения мы можем воспользоваться быстрой очередью по приоритетам, чтобы получить сгруппированные узлы файлов, размеры которых не позволяют разделить их с операционной системой. Если узлы находятся в Q , чтобы извлечь (например, прочесть) данные из файла через очередь по приоритетам (обычно реализуем, как P -розы, записывая в очередь по приоритетам канальную систему), необходимо занести с соответствующим элементом из Q и одним дополнительным узлом. Если этот новый элемент меньше, чем только что помещенный на выход, то поскольку он может и не стать частью текущего сгруппированного блока, мы маркируем его как принадлежащий следующему блоку и считаем, что он больше всех остальных элементов текущего блока. Когда маркированный элемент поднимается в первую очередь по приоритетам, мы начинаем новый блок. На рис. 11.14 этот метод показан в действии.

[illegible]

Если мы воспользуемся приведенной схемой, то получим следующие блоки данных, мы заменим запятой на интервалную память, и этот случай нужно отметить на из памяти еще за собой до тех пор, пока следующее дерево не опустеет. После этого мы заменяем интервалную память очередным пакетом данных и продолжаем этот процесс снова и снова. В среднем алгоритмическое дерево занимает по памяти относительно этого про-



РHCYI+OK 11.14. 061607 C
3AM5JIEHPEM

[illegible]

своих только по поводу пространственной оперативной памяти. В противном случае этому, подобно с замещением, сохраняет эту же структуру в оперативной памяти. Так что не следует забывать о том, что ее эффективность в два раза выше. Полное доказательство этой леммы требует полного и разностороннего анализа (см. раздел 11.4.2), хотя это утверждение нетрудно проверить экспериментальным путем (см. упражнение 11.47).

Что касается файлов с произвольной организацией, то практический результат применения выборки с замещением, по существу, состоит в уменьшении числа проходов к 1: вместо того, чтобы начинать с отсортированных блоков данных, размер которых примерно равен объему пространства оперативной памяти, мы для начала можем использовать отрезки в два раза большего объема пространства оперативной памяти. Для $P = 2$ такая стратегия эквивалентна одному проходу циклически. Для значений P больше 2 полученный эффект несколько скромнее. Тем не менее, мы знаем, что для произвольных файлов с произвольной организацией требуется довольно редко, в каком-либо направлении в той или иной степени, то и ускорения использования метода выборки с замещением можно получить отрезки другого размера. Например, если мы одному из файлов не пользуемся в файле более M блоков, принадлежащих ему по умолчанию, то этот файл может быть полностью отсчитан за один проход выборки с замещением, и при этом никакое влияние не понижается. Эта возможность служит наиболее веским аргументом в пользу практического применения выборки с замещением.

Большая недостатком обобщенной многопроходной сортировки является тот факт, что примерно только половина внешних устройств активно используется во время выполнения процедур слияния: P входных устройств и устройство, используемое для накопления выхода. Альтернативой этому остается выделение $(2P - 1)$ -ступенчатых сортировок с переключением данных на устройства II, с последующим распределением данных на другие машины с ленты в конце каждого прохода слияния. Но этот подход не превосходит первый по эффективности, поскольку он указывает количество проходов, что обусловлено необходимостью распределения данных. Сбалансированные многопроходные машины, по-видимому, требуют либо сложность в виде лентопрозрачных устройств, либо выполнения множества итераций копирования. Разработаны несколько контрольных алгоритмов, которые обеспечивают занятость всех внешних устройств на всех этапах устройства, на которых производится слияние отсортированных блоков данных небольшого размера. Предлагаемый из этих методов получил название многофазного слияния (разумное название).

Осуществляющийся здесь многофазный процесс заключается в том, чтобы разделить отсортированные блоки, полученные в результате выполнения выборки с замещением на доступные лентопрозрачные устройства с определенной степенью неравномерности (оставшаяся одно устройство пустым) с последующим применением стратегии "Зависимый от организации" (зависимый от организации): поскольку спланированные ленты соединяются по длине, одна из них будет использоваться раньше остальных, после чего она может быть использована как входная. То есть, мы меняем длину выходную ленту (длина на ней уменьшается несколько отсортированных блоков) и пустую к этому моменту входную ленту, продолжая этот процесс до тех пор, пока остается только один блок. На рис. 11.15 показан соответствующий пример.

Например, если мы хотим отсортировать 1 миллиард записей, используя один и тот же блок 3 устройства и пространство оперативной памяти, достаточное для размещения 1 миллиона записей, мы можем сделать это посредством двухпутевой многофазной сортировки с $\log_2 500 \approx 9$ проходами. Добывая из входного распределения, мы получаем несколько больших затрат (одна прохода, чем в случае балансирующей сортировки, которое использует в два раза больше чипов устройств). То есть, мы можем рассматривать многофазную сортировку как процедуру, позволяющую выполнять ту же работу, не сбалансировав ресурсы. Для желанного количества устройств многофазное сортирование обеспечивает большую эффективность, чем балансирующая сортировка, о чем свидетельствует рис. 11.17.

Как мы уже отмечали в начале раздела 11.1, тот факт, что мы сосредоточили свое внимание на алгоритме абстрактной сортировки с последовательным доступом к внешним устройствам, позволил выделить проблемы, связанные с разработкой алгоритмов, от решения их реализованных на практике. Для разработки практических приложений нам необходимо преобразить основные предположения и поработать с тем, чтобы они были соблюдены. Например, мы типично от абстрактной реализации функций ввода-вывода, которые передают данные между процессором, внешними устройствами и системными программами средствами. В общем случае в современных системах используются хорошо оптимизированные так называемые промежуточные ресурсы.

Принимая эту крайнюю точку зрения, заметим, что многие из современных вычислительных систем испытывают необходимость в промежуточных ресурсах (то есть в буферах) — более абстрактная точка зрения к внешним устройствам, такая, которая использовалась до сих пор. В виртуальной памяти мы имеем возможность обратиться к круглым блокам данных, содержащим заданное количество записей, помещая их систему в очередь с приоритетной доставкой из очереди в очередь с внешними устройствами и оперативную память, когда нам это нужно: при этом возникает проблема, что доступ к данным в очереди требует не только обращения от процессора к данным, находящимся в оперативной памяти, однако эта проблема не является главной: пока программа обращается к внешней памяти, расположенной в близлежащей области по отношению к чипам, к которым она постоянно обращалась, то потребность в передаче данных с внешних устройств в оперативную память полностью исчезает в результате абстракции внешней памяти — более удачно сформулировано. Например, программы, которые обеспечивают последовательный доступ к дан-

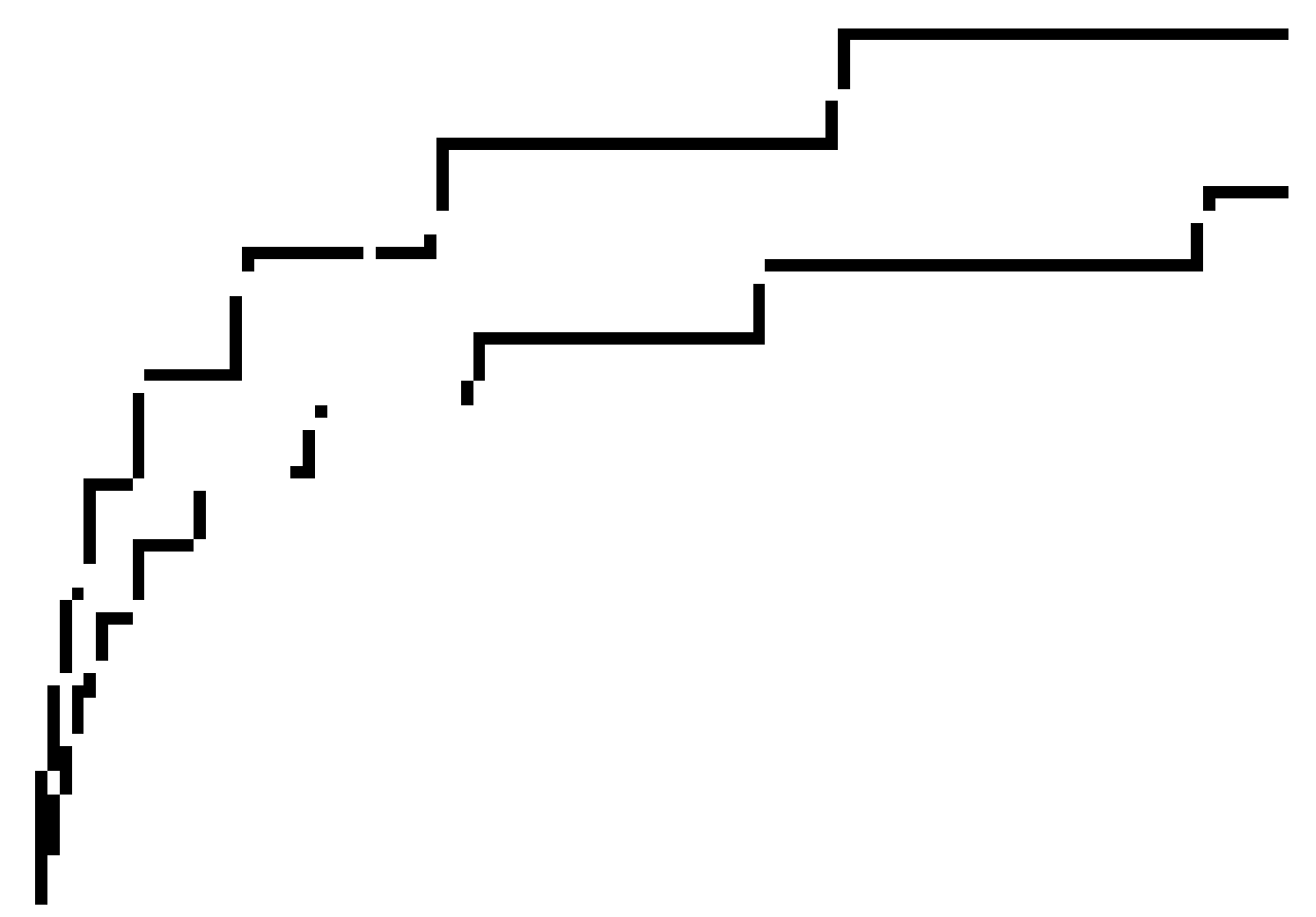


РИСУНОК 11.17. СРАВНЕНИЕ ЗАТРАТ НА БАЛАНСИРОВАННОЕ И МНОГОФАЗНОЕ СОРТИРОВАНИЕ

Устойчивых, выполняющих расчет многофазной сортировки на нескольких устройствах, мы можем получить больше информации об эффективности процедуры многофазной сортировки, если рассмотрим случаи с $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100$. В этих случаях процедура сортировки требует затрат, которые можно сравнить с затратами на сортировку сбалансированной сортировки. В таблице 11.1, для $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100$ показаны значения затрат на сортировку сбалансированной сортировки и многофазной сортировки. В таблице 11.1, для $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100$ показаны значения затрат на сортировку сбалансированной сортировки и многофазной сортировки.

ным, попадают в эту категорию.) Но если данные, к которым производится доступ, разбросаны в разных местах крупного файла, то система виртуальной памяти начнет испытывать трудности (slowly), затрачивая все свое время на доступ к данным во внешней памяти, результаты которой будут катастрофическими.

Не следует игнорировать виртуальную память как возможную альтернативу организации хранения данных в крупных файлах. Мы могли бы изучить программную реализацию виртуальной памяти непосредственно, или это еще проще, могли бы воспользоваться методами внутренней сортировки, такими как быстрая сортировка или сортировка слиянием. Эти методы внутренней сортировки обеспечивают более эффективное хранение в наиболее рабочей зоне виртуальной памяти. Такие методы как внутренняя сортировка или сортировка слиянием сортировка в условиях, когда адрес данных находится в разных концах оперативной памяти, во всей видимости, не позволят нас ее перевернуть.

С другой стороны, использование виртуальной памяти может привести к недостаточным производительным затратам системы, так что работа на объектные данные может оказаться (подобно тем, которые рассматривались выше) может оказаться неэффективным способом получения информации от высокопроизводительных устройств. Одна из основных характеристик научных вычислений заключается в том, что они разрабатываются с расчетом, чтобы максимизировать полезное число операций на чистой компьютерной системе, работающей с максимальной эффективностью и при этом из частей при этом из пространства. Когда в качестве подопытных частей рассматривать процессоры, мы приходим к идее параллельных вычислений, которые рассматриваются в разделе 11.5.

Упражнения

- в 11.45. Показать, какие отрезки порождаются методом поиска с заменением с помощью 10-разрядных размером 4, помеченными A K L M N O P Q R S T U V W X Y Z.
- о 11.46. Как применяется метод поиска с заменением на файле, который был порожден порождающим процессом применения метода выбора с замещением к заданному файлу?
- 11.47. Экспериментально определить среднее число отрезков, порожденных параллельным применением метода выбора с замещением в отрезках по приоритетам размером 10000 к файлам с произвольной протяженностью при $k = 10^2, 10^3, 10^4$ и 10^5 .
- 11.48. Каким будет число отрезков в каждой группе при использовании метода поиска с замещением для генерации отрезков заданного распределения для файла, содержащего N элементов, с использованием отрезков по приоритетам размером M , при $M = 10^2$?
- о 11.49. Показать, как выполняется сортировка элементов E A S Y Q U E S T I O N W I T H E R L E N T Y O F A E Y S в результате применения многофазного сдвига в стиле примера, показанного на диаграмме рис. 11.15.
- с 11.50. В примере многофазного сдвига на рис. 11.15 мы поместили два фиктивных отрезка на заданную ленту с 7 отрезками. Предложите другие способы распределения фиктивных отрезков на лентах и выберите среди них такой, который обеспечивает минимальное по стоимости сдвигание.

11.51. Нарисуйте таблицу, соответствующую рис. 11.11, с целью определить максимальное число отрезков, которые могут быть сдвинуты с помощью метода многофазного сдвига с пятью приходами (с данными 16 исполняемыми устройствами).

11.52. Нарисуйте таблицу, соответствующую рис. 11.16, с целью определить максимальное число отрезков, которые могут быть сдвинуты с помощью метода многофазного сдвига с теми же данными, что и пять приходов по всей совокупности данных (с использованием шести устройств).

- o 11.53. Напишите программу, принимающую число приходов, выполняемых многофазным сдвигом, и эффективное количество приходов, выполняемых методом многофазного сдвига для заданного числа устройств и заданного числа начальных блоков. Используйте эту программу для расчётов таблицы этих данных для каждого метода для $P = 3, 4, 5, 16$ и 100 и $N = 10^2, 10^3, 10^4$ и 10^5 .
- 11.54. Напишите программу, последовательно выполняющую начальные отрезки устройств в условиях P путевых многофазного сдвига. Взяв ряд, когда число отрезков есть эффективное число Фибоначчи, отрезки должны быть выполнены устройствами в соответствии с требованиями алгоритма: новая запись заключается в том, чтобы отложить удобный способ распрямления отрезков по очереди на единицу времени.
- 11.55. Разработать алгоритм с замечанием, сопоставляющим интерфейс, определённым в упражнении 11.38.
- 11.56. Чтобы получить программную реализацию сортировки сдвига, используйте решения, указанные 11.38 и 11.55. Используйте полученную программу для сортировки файла максимального размера в нашей системе размеров, использующейся многофазным сдвигом. Если возможно, определите, как отразится на времени выполнения программы увеличение числа устройств.
- 11.57. Как лучше всего справиться с файлами небольших размеров в рамках реализации быстрой сортировки применительно к файлу сверхбольшого размера в среде виртуальной памяти?
- 11.58. Если на вашем компьютере функционирует подходящая система виртуальной памяти, выполните эмпирическое сравнение быстрой сортировки, параллельной сортировки MSD, последовательной сортировки LSD и параллельной сортировки для сверхбольшого файла. Выберите файл максимально возможный в вашей системе размеров.
- 11.59. Разработать программную реализацию рекурсивной многопутевой сортировки сдвига, в основе которой лежит лемма 8.1.2.5, которую можно использовать для сортировки сверхбольшого файла в среде виртуальной памяти (см. упражнение 8.11).
- 11.60. Если на вашем компьютере функционирует подходящая система виртуальной памяти, эмпирически определите значения, при котором достигаются минимальные время выполнения программной реализации на упражнении 11.59. Напишите файл максимально возможный в вашей системе размеров.

11.5. Параллельная процедура сортировки-слияния

Как сделать так, чтобы несколько независимых процессоров работали совместно, решая одну задачу сортировки? Мы рассмотрим процедуру внешнего упорядочивания устройств или сами попытаемся самостоятельно разобраться в этом вопросе. Мы рассмотрим функции, применяемые к различным подсистемам операционных систем. Проблема параллельности не является в последнее время такой уж актуальной. Разработка эффективных параллельных алгоритмов сортировки, предлагаемых разнообразных машин параллельных вычислений. Во всех этих случаях проблема сортировки возникает как эффективное средство организации и выполнения работы.

Мы уже рассматривали вопрос параллельной обработки данных низкого уровня в процессе изучения сетей сортировки в разделе 11.3, когда обсуждали возможность одновременного выполнения нескольких операций сдвигания-обмена. Сейчас мы рассмотрим модель параллельных вычислений среднего уровня с использованием нескольких универсальных процедур (а не только для сортировки), позволяющих доступ к данным в тем же объеме. Мы хотим сказать, что интерпретация многих практических приложений, зато можем проводить исследования стартовых и конечных случаев.

Абстрактная модель, которую мы используем для представления параллельной обработки данных, базируется на предположении, что сортируемые файлы расположены между независимыми процессорами. Мы предполагаем, что имеются

- N записей, подлежащих сортировке;
- P процессоров, способных хранить (N/P) записей.

Мы принимаем эти процессоры именами $0, 1, \dots, P-1$ и предполагаем, что файл, который служит входом, размещен в памяти локальных процессоров (то есть, в каждом процессоре содержится N/P записей). Цель сортировки заключается в упорядочивании записей таким образом, чтобы N/P наименьших записей находились в памяти процессора 0 , N/P следующих наименьших записей находились в памяти процессора 1 и так далее, в упорядоченном порядке. Как мы увидим далее, существует зависимость между P и длиной времени выполнения — мы хотим получить количественную оценку этой зависимости в результате сравнения различных стратегий.

Представленная модель — одна из многих возможных моделей параллельной обработки данных и для нас характерны именно ее упрощения в смысле практического понимания, какое место имеет в представлении данной модели процесс сортировки (раздел 11.3). И в самом деле, рассматриваемая модель не учитывает одну из наиболее важных проблем, которая весьма актуальна для параллельной обработки данных: взаимодействия, которое заключается во обмен данными между процессорами.

Мы предполагаем, что стоимость такого обмена данных гораздо выше, чем обращение к локальной памяти, и что наибольшая эффективность достигается в тех случаях, когда такой обмен осуществляется по возможности реже. Большие блоки данных. В этом смысле каждый процессор рассматривает остальных процессоров как внешние

аппаратного устройства. К то же время, эту модель с помощью уровня абстракции можно рассмотреть как несуществующую в плане практического применения, такую же чрезмерно упрощённую; её также можно считать моделью абстрактной и в теоретической точке зрения, поскольку ей не хватает полноты определения. Поэтому, несмотря на то, что она представляет собой структуру, в рамках которой возможна разработка полезных алгоритмов.

И в самом деле, рассматриваемая задача (с учетом специальных граничных условий) представляет собой пример нелинейных абстрактных задач, ибо мы для ее решения можем попытаться ввести операторы, которые изучались в разделе 1.2, ввести соответствующие канонические абстрактные континуальности, которые должны все приложить для работы с блоками чипов.

Определение 11.2 Алгоритм называется *полным* на языке L , если алгоритмически разрешима задача для любого M и n из множества M и n из множества M . Если алгоритмически разрешима задача для любого M и n из множества M и n из множества M , то алгоритм называется *полным* на языке L .

Эту операцию можно реализовать, произведя следующие два простых действия, затем идя по ходу пера вдоль дор-ва погонья и в обратном направлении. Получаются в результате следы.

Քաղաք 11.7, Հին քաղաքի համապատասխան ուղղափառացի զննման արձանագրության 13. ուղղության համաձայն ՀՀ/ԲՀ խառնուրդի լայնությունը ՀՀ-ի սահմանից հեռավորագրված քաղաքային տարածքում պահպանվում է իրականում: Քաղաքի հինգ շենքերից մեկից՝ արդարացիորեն ընդհատված է իրականում:

Чтобы установить этот факт, вынужденно приняв нулевую гипотезу, требуется сравнить процентную разницу между двумя группами (11,4%) с тем же процентом в контрольной группе, подобно представленному на рис. 11.3, поскольку требуется вносить поправки на различия.

будем применять метод, описанный в п. 11.7, но вместо стержня (или лопатки). Но прежде чем использовать этот метод на конкретной машине, необходимо выбрать значения некоторых параметров метода. Если интерес к этому методу вообще не связан с задачей, то выбор характеристик

Лемма 11.8. Пусть $\mathcal{A}$ — алгебра операторов на L^2 -пространстве с нормированным ортонормальным базисом $\{e_k\}_{k \in \mathbb{Z}}$. Если $\mathcal{A}$ удовлетворяет условиям (11.1)–(11.3), то для любого $\varepsilon > 0$ найдется $\delta > 0$ такое, что

Под марокканскими жемчужинами (марокканский жемчуг) в данном контексте мы понимаем некую форму созвучности результатов квантификаторов. Если в 11.8 не было прямых сопоставлений между 11.3 и 11.7

Чтобы реализовать компьютер значения из двух взаимодейств. мы должны сделать так, чтобы оба типа и произв. обобщ. к-ции хранились в тех же ячейках памяти (или в соседних ячейках памяти). при этом на одном из них действует следующее соотношение значений элементов, а на другом — та же самая обратная зависимость. Если процессор блокнот происходит именно по сравнению с распределением конкретного процессора, мы можем получить общее представление, необходимое для сортировки, функции сложения. процессор одного блока $\approx (Pr)^2/2$. Таким образом представляется возможным большее число последовательных. например, представляется, что

переданы многократных блоков данных транслируются параллельно, без затрат, что доводит до очень высокой частоты в реальных компьютерных параллельных действиях. Несмотря на все это, она представляет собой отправную точку для понимания того, что можно ожидать от практической реализации.

Если стоимость передачи блока данных определяется быстрой доставкой отдельного процессора (еще одна ключевая идея, к которой можно только приближаться на реальных машинах), то мы должны принять в расчет время, затрачиваемое на типичную сортировку. Каждый процессор выполняет $(N/P) \lg(N/P)$ операций (параллельно), чтобы выполнить начальную сортировку N/P блоков, и примерно $P^2 \lg P / 2$ этапов сжатия типа $(N/P) \rightarrow (N/P)$. Если стоимость сжатия — α , а стоимость сжатия на одну запись составляет β , общее время выполнения приблизительно равно

$$\alpha(N/P) \lg(N/P) + \beta(N/P) P^2 \lg P / 2.$$

Для сверхмощных N и малых P этот показатель — лучшее из того, что можно рассчитывать в условиях идеальной параллельной сортировки, основанной на сжатии, поскольку в этом случае стоимость составляет $\alpha(N \lg N/P)$, которую можно считать оптимальной, любая сортировка требует $N \lg N$ операций в самом лучшем случае, что можно проследить в этом случае — сжать P из них одновременно. В случае больших значений P преобладает второе слагаемое и стоимость приближается к значению $\beta N(P \lg P) / 2$, которое можно считать субоптимальным, хотя все еще конкурентоспособным. Например, первое слагаемое составляет $256\beta N/P$ стоимости сортировки 1 элемента для элементов на 64 процессорах, а во втором слагаемом первое слагаемое составляет $320\beta N/P$.

Когда значение P достаточно велико, то некоторые машины при передаче данных могут возникнуть узкие места. Если такое случится, примененное математическое уравнение, представленное на рис. 11.6, может быть рассмотрено как средство управления поддержкой подобного рода. Многие из этих причинны в некоторых машинах являются встроеными узкими местами архитектуры, которые позволяют эффективно реализовать операции тасования.

Этот пример показывает, что в некоторых случаях можно обнаружить эффективную работу процессоров при решении задач сортировки файлов сверхбольшого размера. Чтобы знать, как это сделать наилучшим образом неизбежно требуется проанализировать множество различных вариантов для данного типа параллельной машины и рассмотреть поведение используемой машины на малых при различных значениях ее параметров. Кроме того, может потребоваться совершенно другой подход к проблеме параллельной обработки данных. Тем не менее, необходимо отметить, что увеличение числа процессоров приводит к увеличению стоимости обмена данными между ними, является оптимальным для параллельных вычислений, в этом случае представляется собой эффективное средство управления только для сортировки, что имеет место как на низком уровне, так и на высоком уровне, и чем мы имеем возможность убедиться в разделе 11.2, так и на высоком уровне, и чем мы имеем возможность убедиться в следующем разделе.

Метод сортировки, описанный в этом разделе и в других местах данной главы, основан на определенных условиях, которые мы изучали на протяжении глав 5–10, поскольку они предусматривают кэширование и другие ограничения, хотя

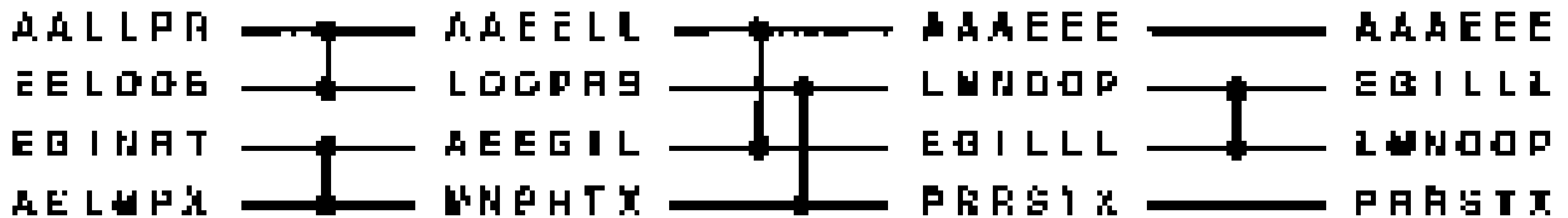


РИСУНОК 11.18. ПРИМЕР ПОБЛУЧНОЙ СОРТИРОВКИ

Рисунок рисует процесс параллельной сортировки данных, который можно использовать в качестве ориентира для рис. 11.19 для сортировки данных. Конкретные значения данных в каждой строке исключены: их можно поменять местами и считать любые данные, а значения переменных: `initialData` и `sortedData` — на их месте любых. Этот код является лишь иллюстрацией алгоритма.

Мы не рассматриваем в данном программном коде. В главах 6—10 простых параллельных, рассматривая различные способы работы с данными, были достаточно, чтобы можно было сравнить большое число различных методов решения одной и той же базовой задачи. В параллельном коде этому в данной главе посвящаются на формулировании различных шагов, но можно возможность рассмотреть лишь некоторые случаи работы с данными. Эти примеры служат в первую очередь тем фактом, что различные ограничения, накладываемые внешней средой, предоставляют возможности для получения разных решений, а наиболее важная часть этого процесса состоит в нахождении подходящих абстрактных формулировок конкретных задач.

Сортировка имеет исключительное значение во многих практических приложениях, и разработка эффективных методов сортировки является одной из главных задач, на решение которых должна быть ориентирована архитектура таких компьютеров и полные этапы программирования. Пытаясь истинно, что новые достижения в области параллельных систем, можно получить продолжение и усовершенствование существующих средств, которые мы обсуждали в главах 6—10, в силу того факта, что появившиеся новые революционные изобретения абстрактное мышление, которое проводилось при изучении сложнейшего ранее материала, может оказаться необходимым, если нам придется разрабатывать быструю действующую процедуру сортировки для сетей машин.

Упражнения

- 11.61. Показать, используя приведенный код и данные рисунка 11.18, как выполняются этапы 11.7.
- 11.62. Реализовать последовательную версию параллельной сортировки с нечетным числом элементов Батчера: (a) воспользовавшись стандартной сортировкой массивов (программы 8.4 и 8.7) для сортировки базовых данных, (b) используя стандартную обмену элементов (программа 8.2) для реализации алгоритмов сортировки и (c) воспользовавшись подходящими методами сортировки Батчера (программа 11.3) для сортировки отсортированных блоков.
- 11.63. Дать оценку времени выполнения программы, описанной в упражнении 11.62, как функции от N и M для больших N .

• 11.64. Выполнить упражнения 11.62 и 11.63, но при этом в обоих случаях записать не только четное значение Битчера программы 11.53 на программу 8.2.

11.65. Найти значения R для которых $(\mathcal{A} \cup \mathcal{B})_R \neq \mathcal{A} \cup \mathcal{B}$ для $\mathcal{A} = 10^3$, 10^4 , 10^5 и 10^6 .

11.66. Найти приближенные выражения для $\mathcal{A} \cup \mathcal{B} \neq \mathcal{A} + \mathcal{B}$ для аппроксимации чисел с помощью элементов данных, используемых в рассмотренной приближенной структуре Битчера для $\mathcal{A} = 1, 4, 16, 64$ и 128 .

11.67. Сколько параллельных шагов потребуется для сортировки 10^5 данных, которые распределены на 1000 дисков, чистоты для этой задачи 100 пикосекунд?

Ссылки на источники к части 3

Следующие материалы для ссылки = типовой разделчик, который не является частью манускрипта Кэрол (Karr), посвященный вопросам сортировки и поиска. Практически по каждой теме, которые мы затрагивали до сих пор, в этой книге можно найти достаточное количество В качестве, все рассмотренные, касающиеся работы, характеристики, различные алгоритмы, которые обсуждались в данной книге, в указанной манускрипте объяснены и описаны, как они используются в алгоритмическом анализе.

Значительная литература по вопросам сортировки. Опубликованная Карлом Р. Ривестом (Rivest) в 1973 г. аннотированная содержит список ссылок. Благодаря которым можно ознакомиться с обзорными работами в области работы и завершения работы множества эмпирических методов, часть из которых была рассмотрена здесь. Кроме того, есть ссылки, опубликованные автором Бобом Карлом (Bob Carr), касающиеся работы, найденные в книге Боба Янга (Bob Yang) и Говна (Gown). Обзор состояния наших знаний о сортировке Пелла можно найти в статье Селливереста (Sellyverest) за 1996 г.

Что касается быстрой сортировки, то наиболее важной работой является пионерская статья Хоара (Hoare), опубликованная в 1962 г. ее рассмотрение можно найти в списке ссылок, которые обсуждались в главе 7. Более подробные детали, касающиеся математического анализа и практических применений многих модифицированных алгоритмов, можно уже найти более поздних работ, которые были широко применены на практике, можно найти в статье Селливереста, опубликованной в 1978 г. Беллами (Bellamy) и Мак-Ниром (McNair) дали его современную трактовку. Мак-Нир и Беллами описывают работу в главе 7 и трехуровневой процедурой быстрой сортировки в главе 10 основаны на книге на этой статье и статье Беллами и Мак-Ниром, опубликованной в печати в 1978 г. Первый алгоритм, называемый «быстрой сортировкой» (или быстрой сортировкой для сортировки сортировки с объяснения) был опубликован в статье Хиллсбруншта (Hillsbrunsh) и Хиллс (Hill) в 1939 г.

Структура данных описывается в книге Виллемса (Willem) = тем, где, в каком она была реализована и использована Брауном (Brown), поддерживает все операции над очередями с приоритетами, описанные в алгоритме. (Включая сортировку данных, описанные Фредом (Fred), Селливерестом (Sellyverest) и Тарьяном (Taryan), являются усовершенствованными базовыми понятиями и представляются членами практической теории.

В книге, опубликованной в 1993 г., Мак-Ниром, Бобом (Bob) и Мак-Ниром представляют последние данные реализации алгоритма сортировки.

Список литературы

1. Ахла-Янес and Г. Н. Гормел, *Handbook of Algorithms and Data Structures*, second edition, Addison Wesley, Reading, MA, 1984.
2. L. Bentley and M. J. Orlin, "Engineering a sort routine," *Software—Practice and Experience* 23, 1 (January, 1993).
3. L. Bentley and R. Sedgwick, "Sorting and searching strings," *Eighth Symposium on Discrete Algorithms*, New Orleans, January, 1990.
4. R. Driscoll, "Implementation and analysis of binomial queue algorithms," *SIAM Journal of Computing* 7, 4 (August, 1978).
5. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan, "The pairing heap: a new form of self-adjusting heap," *Algorithms* 1, 1 (1986).
6. Hildebrandt and H. Isbits, "Radix exchange — an internal sorting method for digital computers," *Journal of the ACM* 6, 2 (1953).
7. A. R. Hoare, "Quicksort," *Communications of the ACM* 5, 1 (1962).
8. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, second edition, Addison Wesley, Reading, MA, 1997.
9. M. McIlroy, K. Bowler and M. D. McIlroy, "The new radix sort," *Computing Systems* 6, 1 (1993).
10. L. Rivest and D. E. Knuth, "Biography of Computing Sorting," *Computing Science*, 13 6 June, 1972.
11. Sedgwick, "Implementing quicksort programs," *Communications of the ACM* 21, 10 (October 1978).
12. Sedgwick, "Analysis of heap sort and related algorithms," *Fourth European Symposium on Algorithms*, Barcelona, September, 1995.
13. Villanar, "A data structure for non-priority queue," *Communications of the ACM* 21, 4 (April, 1978).

Поиск

Базисные задачи:

12	Таблица смежности и матрица бинарного поиска
13	Сбалансированные деревья
14	Хеширование
15	Порядковый поиск
16	Внешний поиск

Таблицы символов и деревья бинарного поиска

Получение конкретного фрагмента или фрагментов информации из бинарных типов ранее охваченных данных — основополагающая операция, называемая поиском, характерная для многих вычислительных задач. Как и в случае с алгоритмами сортировки, алгоритмы и структуры в § 6 не являются конкретными приоритетами, они описаны в главе 9, мы работаем с данными, разделенными на записи, или элементы, каждый из которых имеет ключ, используемый при поиске. Цель поиска — отыскание элементов с ключами, которые соответствуют заданному ключу поиска. Конечно, необходимость поиска данных подразумевает доступ к информации внутри элемента (а не просто к ключу) с целью ее обработки.

Поиск используется повсеместно и связан с выполнением множества различных операций. Например, в банке требуется отслеживать информацию о счетах всех клиентов и выполнять поиск в этих записях для предоставления данных и выполнения банковских операций. Но алгоритмы быстрого отыскания информации могут на каждом этапе и вычислять новые свободные места, отказы и графики билетов или предоставления каких либо изменений в список пассажиров. Еще один пример — средство поиска в системном интерфейсе программы, которое отыскивает в сети все документы, содержащие заданное ключевое слово. Требования, предъявляемые к этим приложениям, в чем-то совпадают (и для банка, и для системы поиска требуются надежность и надежность), а в чем-то различны. Банковские данные имеют длительный срок хранения по сравнению

в значениях остальных упомянутых приложений). Так же можно во всем этом процессе обнаружить эффективные алгоритмы поиска.

Определение 12.1 Таблица символов — это структура данных, состоящая из элементов, которые используются для быстрого определения позиции каждого элемента и возврата элемента с заданным значением.

Иногда таблицы символов называют также словарями (dictionary), по аналогии с проверкой вписанном словом представленные определения слов путем переключения на соответствующие алфавитным порядкам. Так, в словаре английского (или любого другого) языка "ключи" — это слова, а "значения" — значения, которые им соответствуют, которые содержат определения, правила применения к другим символам. Словарики используются для хранения информации в словарях, обычно используются на языковом распознавании знаний. Телефонные книги, энциклопедии и другие справочники, в частности, организованы таким же образом, и некоторые из рассмотренных методов поиска (например, алгоритм Бинарного поиска в разделах 2.6 и 12.4), также используются на них, но здесь упорядочены.

Таблицы символов и компьютерная структура относятся к тому, что они являются частью более сложными, чем словарь или телефонная книга. Поэтому большинство рассмотренных методов создают структуры данных, которые не только позволяют использовать эффективные алгоритмы поиска, но и поддерживают эффективные реализации операций хранения данных элементов, удаление или изменение элементов, объединение двух таблиц, слияние одной в другую и т.п. В этой главе будут даны рассмотрены многие из алгоритмов, связанных с операциями, которые используются применительно к операциям по приоритетам в главе 9. Разработка динамических структур данных для поиска — одна из областей и наиболее активно изучаемых проблем в компьютерной науке; она будет рассмотрена в главе 12. Это направление в этой главе, как и в главах 14—16. Как будет показано, для решения задачи реализации таблиц символов разработаны (и продолжают разрабатываться) множество оригинальных алгоритмов.

Изучение компьютерных наук и компьютерных технологий приводит к другим приложениям таблиц символов, кроме упомянутых основных, поскольку эти таблицы — фундаментальное средство при организации программного обеспечения и компьютерных систем. Таблица символов служит базисом для программы языка — это символические имена, используемые в программе, а элементы содержат информацию, описывающую упомянутые объекты. Начиная с уровня развития языков программирования, любая таблица символов позволяет программистам переходить от использования числовых адресов в машинных кодах к символическим именам языка программирования, и наоборот, связывая конкретные приложения с номерами тысячелетия. Когда символические имена имеют определенное значение в рамках конкретной компьютерной системы, алгоритмы поиска могут и будут играть важную роль при компьютерной работе.

Таблицы символов часто встречаются также в абстрактных языках описания, а иногда и на аппаратном уровне. Для объяснения этого понятия иногда используется термин "символьная память". Мы будем использовать значения программным языкам, но которые на рассмотренных методах применяются также и к аппаратной реализации.

Как и при изучении методов сортировки в главе 9, в этой главе изучение методов поиска начинается с рассмотрения ряда элементарных методов, пригодных для большинства таблиц и в некоторых особых ситуациях, а которые представляют собой не технологии, неспециализированные завершённые методы. Затем, в оставшейся части главы основное внимание уделяется более продвинутому набору методов поиска — B⁺ и B-деревьям, сегментационной и широкой используемой структуре данных, допускающей произвольные запросы быстрого поиска.

В разделе 12.5 в качестве иллюстрации эффективности математического анализа при разработке эффективных алгоритмов рассмотрены два алгоритма поиска. Для полноты изложения в этой главе материалы и потери части информации, приведённой в главе 7.5, в ряде случаев для ознакомления с некоторыми доказательствами будет приводиться ссылка на эту главу. Позже в этой главе мы обратимся также к алгоритму быстрого поиска в деревьях, который рассмотрен в разделе 12.4 и 12.5.

12.1 Абстрактный тип данных таблицы символов

Как и при рассмотрении операций приоритета, алгоритм поиска можно рассмотреть как принципально к интерфейсу, охватывающему множество общих операций, которые могут быть вызваны из конкретных реализаций, что позволяет легко и просто заменить одну реализацию другой. Интерес представляет следующий список:

- Добавка нового элемента.
- Поиск элемента (или элементов) с заданным ключом.
- Проверка существования элемента.
- Выход из таблицы по заданию элемента в таблице символов.
- Сортировка таблиц по ключам (преобразование всех элементов в порядок их упорядочивания).
- Преобразование двух таблиц в символ.

Подобная множеству других структур данных, к этому набору может потребоваться добавление стандартных операций создания, проверки, освобождения элементов, уничтожения и уничтожения. Кроме того, может потребоваться рассмотрение различных других практических применений, например, методы поиска. Например, часто операции поиска и вставки реализованы весьма неуклюжо, поскольку во многих реализациях поиск сложный даже безуспешный, тем не менее, предоставляет точную информацию, необходимую для дальнейшего поиска. Поэтому для этого сформирован следующий алгоритм.

В общем случае термин "алгоритм поиска" используется в значении "реализация абстрактного типа данных таблицы символов", хотя, строго говоря, поскольку мы рассматриваем только операции, связанные с построением основывающейся структуры данных таблицы символов и добавлением к поиску, реализацию алгоритма абстрактного типа данных. Ключевым моментом является то, что можно использовать многокомпонентные приложения, если доступны в качестве высокоуровневой абстрактной таблицы данных среды программирования. Стандартная библиотека C содержит функции, реализующие алгоритм Б-дерева быстрого поиска, описанного в разделе 12.4, а библиотека стандартных таблиц C++ предоставляет базовую поддержку таблиц, описанных в "Алгоритмах

ными кинематографом. Как обычно, трудно добиться, чтобы реализация общего назначения удовлетворяла требованиям производительности, предъявляемым к специализированным приложениям. В процессе изучения многих прикладных методов, разработанных для реализации объектных языков символов, мы обнаружили контакт, который способен понять характеристики готовых реализаций и принять решение о необходимости реализации, предназначенной для конкретного приложения.

Как и в случае с эвентировкой, мы рассмотрим методы без представления типов для работы с данными элементов. Стоит же подробно, как в разделе 6.5, будет использоваться реализация, использующие интерфейс, который определяет `Item` и обеспечивает абстрактные операции с данными. Мы рассмотрим методы как с использованием символов, так и с использованием кода, использующие в качестве аннотаций ключи или идентификаторы данных. Чтобы реализовать ذلك, выполняемые приписки элементов `key` и `value`, понятия `Item` (элемент), `isElement` (—?), `isKey` (—?), расширяясь до элементов, содержащих ключи типа `Key`. Поскольку требуется несколько базовых элементов, чем было необходимо для взаимодействия с объектами эвентировки, будем считать, что они определены в пакете абстрактных типов данных, реализующих интерфейс `Class` — как показано в программе 12.1. Функция `key()` будет применяться для извлечения ключа из элемента, с порождающим объектом `object` — для проверки равенства двух ключей. В этой главе и главе 13 также перетронуется `object` для сопоставления значений двух ключей, что помогает при локализации, эвентировке, поиске. Описанные в главах 14 и 15, основанные на извлечении идентификаторов ключей за счет использования базовых операций с кодами, которые не применяются в главе 10. Кроме того, предполагается, что элементы могут генерировать значения `value` (`val()`), и что ключи имеют доступ к функции `val()`, которая может проверить, является ли элемент нулевым. Нулевые элементы используются для кодирования `isElement` значения в том случае, когда ни один элемент в таблице символов не имеет искомого ключа. В некоторых реализациях предполагается, что нулевые элементы имеют служебный код.

Программа 12.1 Пример реализации АД элементов

Это определение класса элементов, представляющих собой небольшие значения, состоящие из целочисленных ключей и связанных с ними информации. Оно основано на описании таблицы, реализующей таблицу символов. В реализации таблицы символов — ключевые значения, в которых хранение — и используется для хранения ключей, а функции `key()` и `val()` — системы символов для получения значений ключей и проверки соответствия элементу нулевым.

В определении типа элемента есть также методы: функция `isKey` (ссылающаяся на `key()`), `isVal` (ссылающаяся на `val()`) и `isItem` (ссылающаяся на `key()`), которые будут использоваться драйверами. Это позволяет создавать и тестировать различные реализации таблиц символов, используя различные стратегии хранения.

```
#include <stdio.h>
#include <stdlib.h>
static int maxKey = 1000;
typedef int Key;
class Item {
public:
private:
```



```

key: keyval:
float info:
main:
  main()
  { keyval = getKey(); }
key: key()
  { return keyval; }
int null()
  { return keyval == nullKey; }
void read()
  { keyval = 1000 * read()/RAND_MAX;
    info = 1.0 * read()/RAND_MAX; }
int main(int argc, char ** argv)
  { return (argc == 0 ? 0 : 1); }
void show(float x, float y)
  { printf("keyval: %d\n", keyval); }
}
extern void read();
extern void show();

```

Чтобы пользоваться при разработке интерфейса и реализации для чисел с плавающей точкой, строк и более сложных элементов, описанных в разделах 6.8 и 6.9, нужно только убедиться, что в Key правильно определены `key()`, `null()`, `read()` и `show()`, а также элементы `rand`, `key` и `info`, сделать их функциями-членами, которые соответствуют их логическому соответствию образу.

Программа 12.2 — интерфейс, который определяет базовые операции таблиц символов (за исключением операции `delete`). Этот интерфейс будет использоваться и клиентскими программами и все реализации должны в этой и нескольких следующих главах. Абстрактный тип данных первого класса не понимается в том смысле, как это понималось в разделе 4.8 (см. примечание 12.5), поскольку в большинстве программ используется только одна таблица, а добавление конструкторов копирования, перегруженных операций присваивания и деструкторов, хотя и возможно, часто в большинстве реализаций. Но все же они отличаются от обычных характеристик алгоритмов. И программа 12.2 может быть также определена версией интерфейса для манипулирования экранными элементами подобно программе 9.3 (см. выражение 12.7), но это может усложнить программу и типичный интерфейс, когда достаточно манипулировать элементом по своему ключу. Интерфейс не знает, какой элемент является элементом, который должен быть удален. В большинстве реализаций используется интерпретация "удалить элемент с ключом, равным данному элементу", при этом подразумевается предварительный поиск. В других реализациях, которые предоставляют экраны и могут типизировать проверку идентичности элемента, необходимость поиска перед удалением исключается, и поэтому для них допустимы более быстрые алгоритмы. Кроме того, при рассмотрении алгоритмов для операций абстрактных, предполагающих наличие приложений, которые образуют несколько таблиц символов, можно использовать реализации АТД первого класса таблиц символов, которые ведут к минимальному времени работы и выходу памяти (см. раздел 12.5).

Программа 12.2 АТД таблицы синонимов

В этом интерфейсе определены операции для таблицы синонимов: инициализация, поиск значения по ключу, поиск элемента с заданным ключом, добавление нового элемента, удаление элемента, выбор к-го наименьшего значения и т. д. Обработка элементов в таблице не описана (я оставил ее открытой для вас).

```
template <class Tkey, class Tval>
class AT
{
public:
    AT():
        // Инициализация из конструктора
    {
        init();
    }
    void init();
    Tkey search(Tval);
    void insert(Tval);
    Tkey select(int);
    void show();
};
```

В некоторых алгоритмах не представляется возможным какое-либо определение порядка ключей, поэтому для различных ключей в них используется только оператор «равенства» (`==`), однако во многих различных таблицах синонимов используется разная организация ключей, применяемая в алгоритмах для структурирования данных и управления данными. Кроме того, обязательным элементом `keyof` и `valof` (содержащим явные ссылки на порядок ключей). Функция `keyof` объявляется в пакете и имеет функцию, которая отправляет все элементы в выходной поток по порядку без необходимости переконфигурации. Реализация `valof` может быть объявлена с целью получения функции, которая покажет элементы в порядке их адреса, возможно, поместив в элемент не только процедуру, возвращающую элемент, но и другие данные. Используемые функции `keyof` для таблиц синонимов и `valof` (или `keyof` и `valof`) являются обязательными для реализации алгоритмов, в которых оператор `==` используется, нет необходимости, чтобы ключи были связаны друг с другом, поэтому даже алгоритмы, требующие поддержки операции `keyof` и `valof`.

Случай возможности существования элементов с дублированными ключами при реализации реализации таблицы синонимов должен рассматриваться особо. Некоторые приложения не требуют существования дублированных ключей, поэтому ключи могут использоваться в качестве дескрипторов. Примером такой ситуации может служить использование номеров картриджей эластичного хранения в качестве ключей для различных файлов. Напротив, в других приложениях может потребоваться наличие нескольких элементов с одинаковыми ключами, например, список по ключевому слову в базе данных документов, как правило, будет приводить к нескольким ссылкам на документ в указанном поле.

Обработка элементов с дублированными ключами может выполняться одним из нескольких способов. Одним из подходов — построение на том, чтобы каждая в первую очередь структура данных содержала только элементы с различными ключами и объединение для каждого ключа ссылки на список элементов приложения, содержащих дублированные ключи. То есть, в базовых структурах данных используются списки, которые содержат ключ и список, и представляют элементы с одинаковыми ключами.

Программа 12.3 — пример консольной программы, иллюстрирующей умение языка составлении для различных таблиц циклов. Программа воспринимает таблицу значений для поиска различных значений и последовательности ключей (регулярной или произвольной или связанной со стандартной вход), и затем выводит их в отсортированном порядке.

```
#include <csapp.h>
#include <stdlib.h>
#include "Item.h"
#include "HT.h"

int main(int argc, char *argv[])
{
    int N, maxN = MAX(argc-1), m = MAX(maxN/2, 1);
    HT*HTm, *Keys;
    for (N = 0; N < maxN; N++)
    {
        Item v;
        if (N % 5 == 0) else if (!N % 10) break;
        if (!setkey(v.key(1), 2, 1)) continue;
        setkey(v);
    }
}
```


1

Как обычно, создается иллюзия в виду, что различные реализации операций на таблицах символов обладают различными характеристиками практической полезности, которые имеют отношение к конкретному набору операций. В одном положении операции *insert* может выполняться сравнительно редко и недорого, для построения таблиц *insert* при огромном количестве попыток новых операций *insert*, в другом, в сравнительно небольших таблицах, может выполняться огромное количество операций *insert* и *delete*, порождаемых операциями *insert*. Не все реализации будут поддерживать все операции, и некоторые из них могут обеспечивать эффективную реализацию операций *insert* за счет других; при этом явное предположение, что менее эффективные функции выполняются реже. Каждая из базовых операций и интерпретация таблицы символов порождает важные предположения, поэтому для обеспечения эффективного использования различных комбинаций операций требуется множество базовых вариантов реализации. По этой и некоторым другим причинам для основных значений были выбраны реализации базовых функций *insert*, *delete*, *find* и *link*, и прилагается некоторый плановый стандарт не функций *insert*, *delete*, *find* и *link*, когда в этом возникает необходимость. Широко известное количество алгоритмов, требующих рассмотрения, обусловлено различным характером практической применимости различных комбинаций базовых операций, а также ограниченными возможностями для значений *link*-*find*, *insert*-*delete* и другими факторами.

В этой главе мы встречаемся с рядами чисел, в которых среднее между соседними членами равно их сумме. Такие ряды называются рядами Фибоначчи. Мы рассмотрим некоторые свойства этих рядов, а также их приложения в различных областях математики и физики. В частности, мы увидим, что ряды Фибоначчи имеют тесную связь с золотым сечением, которое является одним из самых красивых и гармоничных соотношений в природе.

Изучены и многие другие операции с табличным символом. Примерами могут служить поиск по клеткам (Поиск Желтой), при котором поиск может начинаться с точки, в которой закреплена предыдущая точка, может и прерываться, если нужно получить только приблизительные результаты, поворачивая в каждый интервал, и поиск аннотированного списка (если определено понятие расстояния), при котором получается поиск клеток, для жидких и твердых.

Упражнения

- 12.2 Создать реализацию класса `Item` (классическую программу 12.1), поддерживающую обработку различных таблиц значений элементов, состоящих из комбинации из целочисленных ключей в строке (1), а также поддерживающего буфер для строк, как это делается в программе 6.11.

- 12.3 Используя программу 12.2 АТД таблицы символов, создать реализацию АТД списка и очереди.
- 12.4 Используя АТД таблицы символов, предложенный программой интерфейса 12.2, создать реализацию АТД очереди по приоритету, который поддерживает операции удаления как максимального, так и минимального элементов.
- 12.5 Используя АТД таблицы символов, предложенный программой интерфейса 12.2, создать реализацию сортировки массива, совместимой с реализацией, описанной в главах 6—10.
- 12.6 Добавьте в программу 12.2 объявление деструктора, конструктора копирования и перегруженной операции присваивания, чтобы преобразовать ее в АТД германского языка (см. рисунки 4.3 и 9.5).
- 12.7 Определите интерфейс АТД таблицы символов, который позволит клиентскому программисту удалять конкретные элементы с использованием дескрипторов и именит, клонировать, рисунки 4.8 и 9.3).
- 12.8 Приложите интерфейс типа элемент и реализации элементов с двумя полями: 16-битным десятичным числом и строкой C, которая охватывает информацию, связанную с этим числом.
- 12.9 Укажите среднее количество отдельных элементов, которые программа-клиент (программа 12.3) будет извлекать среди N прикладных данных, меньше 1000, для $N = 10, 10^2, 10^3, 10^4$ и 10^5 . Определите свой ответ эмпирически, если иначе не будет обонан методов.

12.2 Поиск с использованием индексации по ключам

Предположим, что значения ключей — отдельные неотрицательные числа. В этом случае простейший алгоритм поиска основывается на следующем элементарном факте: поиск элемента по ключу, как описано в реализации, приведенной в программе 12.4. Каждый простой оператор `push` увеличивает все табличные значения `pushed`, затем выставляются (`insert`) значения со значениями ключа `k` предыдущих `push` в массив `at[k]`, и выполняется поиск (`search`) элемента со значением ключа `k`, пытаясь его в `at[k]`. Для удаления (`remove`) элемента со значением ключа `k` значение `pushed` уменьшается в `at[k]`. Реализация очереди выбора (`select`), сортировки (`sort`) и удаления (`remove`) в программе 12.4 используют линейный просмотр массива с пропуском пустых элементов. Реализация преобразования элемента решает задачи обработки элементов с дублированными ключами и проверки таких условий, как указанные операции с тем же ключом, отсутствующего в таблице.

Эта реализация служит отправной точкой для всех реализаций таблиц символов, которые рассматриваются в этой главе и главах 13—15. Как таковая она может использоваться для различных клиентов и с различными типами элементов. Компилятор будет проверять, выполняются ли интерфейсы, реализованные клиентом, и все же ограничения.

Операция индексации, на которой основывается поиск с использованием индексации по ключам, сочетается с базисной операцией и является сортировкой с подсчетом или дублированием элементов, которая была рассмотрена в разделе 6.10. Когда это требуется,

нельзя выбрать метод поиска с использованием индексации по ключам, поскольку отношения между n и m могут оказаться неэффективными.

Если элементы вообще отсутствуют (иначе говоря только ключи), можно использовать таблицу бит. В этом случае таблица символов является таблицей существования (existence table), поскольку i и j могут быть не только считаны, но и проиндексированы i и набором ключей таблицы. Например, на 32-разрядном компьютере этот метод можно было бы использовать для быстрого нахождения того, используется ли уже конкретный 4-матричный номер телефонного компьютера, используя таблицу из 312 слов.

Лемма 12.1 Если задана матрица M размерности m по строкам и n по столбцам, то для любых таблиц символов S и T можно реализовать по M индексацию по ключам S и T с помощью M элементов S и T для быстрого поиска. Если S и T являются таблицами символов, то для быстрого поиска S и T можно использовать M элементов S и T для быстрого поиска. Если S и T являются таблицами символов, то для быстрого поиска S и T можно использовать M элементов S и T для быстрого поиска.

Эта лемма является частным случаем леммы 12.1. Обратите внимание, что на ключи накладываются условия $M \leq n$.

Программа 12.4 не обрабатывает дублирующиеся ключи и в ней предполагается, что значения ключей лежат в пределах между 0 и $M-1$. Для хранения данных элементов с дублирующимися ключами важно было бы использовать список списков или один из рассмотренных в разделе 12.1. и перед использованием ключей в качестве индексов можно было бы использовать их преобразование (см. упражнение 12.13). Но мы сейчас рассмотрим более сложный случай случая ключей до $M-1$, в котором рассматриваются всевозможные приемы для реализации таблиц символов для общих ключей используется этот же подход. Так как структура преобразования ключей из достаточно широкого диапазона в узкий с использованием действия для элементов с дублирующимися ключами. Пока будем предполагать, что старым элементом со значением i и j , равным i и j , встраивается элемент, можно использовать i и j в упражнении 12.11. для же считать это успешным шагом (см. упражнение 12.10).

Программа 12.4 Таблица символов, основанная на индексированном по ключам массиве

В этой программе предполагается, что значения ключей — положительные целые числа, меньшие заданного значения M , и для использования M элементов массива. Предполагается, что конструктор `Item` создает элементы по заданным ключам, равным заданному ему значению, чтобы конструктор `ST` мог создать значение M в нулевом элементе. При этом, прежде всего, необходимо создать массив памяти, который требуется, когда переопределяется значение ключа и его значение необходимо в конструктору `ST`, когда значение M было по сравнению со значением M .

```

type Item = Item; class Key;
class ST
{
private:
    Item *table;
    int n;
public:

```


Индексирование по ключам массива удобно для многих приложений, но оно не универсально, для многих же ситуаций в узкой области действительности можно считать, что эти и несколько следующих глав помогут разработать варианты для случаев, когда диапазон возможных значений ключей столь широк, что невозможно использовать индексированную таблицу с одной потенциальной таблицей для каждого ключа.

Упражнения

12.10 Реализуйте АТД таблицы символов первого класса (см. упражнение 12.1), используя динамически распределенные массивы. Индексирование по ключам.

а. 12.11 Измените реализацию, представленную в программе 12.1, чтобы обеспечить "энергичную" реализацию функции `search` (путем отбрасывания ключевых нулевых значений).

б. 12.12 Измените реализацию, описанную в упражнении 12.10, чтобы обеспечить "энергичную" реализацию функции `search` (см. упражнение 12.11).

12.13 Разработайте версию программы 12.4, в которой используется функция `h(key)`, преобразующая ключ в потенциальное целое число манитора M (так, чтобы никакие два ключа не отображались на один и тот же целая число). (Это усложнение становится полезным различием, когда ключи относятся к узкому диапазону (не обязательно начинающемуся с 0) и в других простых случаях.)

12.14 Модифицируйте версию программы 12.4 для случая, когда манитор представляет собой таблицу, позволяющую положительным целым и отрицательным M (для каждой либо стрелкой информации). В реализации попытайтесь динамически распределить массив, состоящий приблизительно из $M/\log_2 M$ строк, где $\log_2 M$ — значение $\log_2$ в двоичной системе в десятичной компьютерной системе.

12.15 Измените реализацию, описанную в упражнении 12.14, для экспоненциально более быстрого представления стрелок и стандартной величины отдаленных целых чисел в произвольной лоскутной системе M (используемых для целых чисел, меньших M для близости к области памяти, доступной программе в компьютерной системе, в выражении в битах (см. программу 12.3)).

12.3 Последовательный поиск

В общем случае, когда значения ключей относятся к широкому большому диапазону, можно было использовать в качестве индексов, один из простых способов реализации таблиц поиска — последовательное сканирование элементов массива в установленном порядке. Когда требуется сделать новые элементы, мы выведем его в массив, переменная большинство из них являются, как эти элементы для подтверждения вставки, когда необходимо выполнить поиск, чтобы проверить наличие последовательности. Поскольку массив упорядочен, при запуске ключа, значение которого больше или меньше, можно сразу вступить в процесс поиска. После того, как будет обнаружено соответствие, реализация операций `search` и `add` не представляет сложности. Программа 12.5 описывает реализацию таблицы символов, которая выполняется из этого подхода.

Программа 12.5 Таблица символов (упорядоченная) с использованием массива

Таблицу символов (таблицу 12.4), в свою очередь реализовав с помощью массива, но для этого необходимо чтобы элементы были небольшими (целыми числами). Получаемая упорядоченности массивов соответствуют тем, что при вставке нового элемента добавление элементов (начиная с конца списка) должно происходить. Как это сделать для сортировки элементов? В этом случае функция select может быть использована для выбора элементов с указанным значением, но значением null (или для выбора элементов с большим значением). Реализация функции select и sort таковы, а реализация функции insert (вставка) и функция remove (удаление) см. приложение 12.10.

```
template <class Item, class Key>
class ST
{
public:
    Item nullItem, *at;
    int N;
public:
    ST(int maxN)
    { at = new Item[maxN+1]; N = 0; }
    int count()
    { return N; }
    void insert(Item x)
    { int i = N++; Key v = x.key();
      while (i > 0 && v < at[i-1].key())
          ( at[i] = at[i-1], i--, );
      at[i] = x; }
    Item search(Key v)
    {
        for (int i = 0; i < N; i++)
            if ((at[i].key() < v)) break;
        if (v == at[i].key()) return at[i];
        return nullItem;
    }
    Item select(int k)
    { return at[k]; }
    void show(const char *s)
    { int i = 0;
      while (i < N) at[i++].show(s); }
};
```

В программе 12.5 можно было бы использовать массивы для хранения элементов и реализовать итерации для подсчета использования элементов для каждого элемента проверки, но лучше всего использовать массив, так как, если мы знаем, что элемент таблицы не содержит какого-либо ключа. В частности, следующую часть массива можно было бы использовать в качестве служебной, а затем перед поиском записать ее часть ключа искомого элемента. В таком случае поиск будет быстрее, так как элемент, содержащий искомого ключа, а то, что элемент не ключ и больше, можно использовать, например, нулевой или другой служебный.

Другой частью списка с элементами является упорядоченность элементов. В массиве по порядку не является обязательным. При вставке новый элемент вставляется в конец массива; во время поиска массив просматривается последовательно.

Этот типично характеризуется тем, что операция insert выполняется быстрее, а операция delete — медленнее. Это требует значительно большей работы (для выполнения каждой из них требуется реализация одного из методов, описанных в главах 7–10). Удаление (insertion) элемента с заданным ключом можно выполнить, отыскав его, а затем переместив последний элемент списка в позицию удаленного элемента и увеличив для всех элементов на 1, удалив тем самым элемент с заданным ключом реализацией путем перемещения этой операции. Если доступен дескриптор, предоставляющий список элементов в массиве, то не требуется и операция перемещения элементов из массива в массив.

Еще одна простая реализация таблицы символов — несортированная списокная. В этом случае можно также хранить список в упорядоченном виде с целью упрощения реализации операций delete либо избежать его использования для ускорения операции insert. Программа 12.6 демонстрирует второй подход. Как обычно, преимущество при реализации списка состоит из-за того, что массив не обязательно сразу определяет максимальный размер таблицы, а недостаток — в необходимости хранения дополнительного блока памяти (пол ссылки) к некомпактности эффективной реализации операции delete.

Программа 12.6 Таблица символов (несортированная) с использованием списка

В этой реализации операции insert, delete, search и lookup используются списочный список, в котором каждый элемент имеет указатель на следующий элемент. Функция insert помещает новый элемент в конец списка и выполняется за постоянную времени. Функция delete удаляет заданный элемент (реализация функции lookup для просмотра списка).

Поскольку список не упорядочен, реализация операций insert и delete осуществляется

```
#include <stdlib.h>
template <class T, const class Key>
class ST
{
private:
    T** m; // массив
    struct node
    {
        T* item; node* next;
        node(T* a, node* n)
            : item(a), next(n) {}
    };
    typedef node* link;
    int N;
    link head;
    link search(link t, Key k)
    { if (t == 0) return nullitem;
      if (t->item.key() == k) return t->item;
      return search(t->next, k);
    }
public:
    ST(int count)
    { head = 0; N = 0; }
    int count()
    { return N; }
    void insert(Key k)
    { return insert(head, k); }
    void insert(T* a)
    { head = new node(a, head); N++; }
};
```


Подходы к негалактичному упорядоченному массиву и к упорядоченному списку оставлены для самостоятельной реализации (см. упражнения 12.20 и 12.21). Все четыре упомянутых подхода (с использованием массивов и списков, упорядоченных и неупорядоченных) могут использоваться в приложениях сами вместе друг с другом, причем не только в рамках выполнения и требуемого объема памяти. В итоге в нескольких следующих главах исследуются множество различных методов к решению задачи реализации таблиц и списков.

Содержание элементов в упорядоченном виде — полнотрапезная вещь, что и с общим успехом и репликация таблиц: структура ключей используется для определенной структуры данных в целях ускорения поиска. Структура может обеспечивать быстрые расширения и другие другие операции. Но при этом следует учитывать затраты на поддержку структуры, которые могут привести к замедлению других операций. Мы рассмотрим с кивками примеры реализации этого. Например, в приложениях, где функция `get` требуется часто, нужно было бы выбрать упорядоченное представление (с использованием массива или списка), поскольку такая структура таблиц не требует дополнительной функции для тривиальной, в отличие от неупорядоченной таблицы, реализации функции `get`. В приложениях, где часто требуется частые тригонометрические операции `set`, можно было бы использовать неупорядоченное представление упорядоченного массива, поскольку при такой структуре таблиц затрачиваемое на выполнение упомянутой операции время постоянно. И наконец, время выполнения `set` не зависит от размера списка линейно: зависит от количества элементов, даже если список упорядочен.

Чтобы подробно проанализировать производительность таких произвольных ключей, начнем с рассмотрения затрат на работу с ними: поиск ключей и особенно расширение списка данных в неупорядоченном виде. Первый часто пытаются реализовать для массива и для списка. Однако при этом. Мы исследуем затраты как при увеличении, так и при уменьшении размера и других случаях. Строго говоря, в реализации с использованием упорядоченного массива (см. упражнение 12.3) для каждого неупорядоченного элемента используются две операции сравнения (`==` и `<`). При увеличении списка `add` каждая новая пара будет считаться одной операцией сравнения, поскольку сначала проверяется на равенство, а можно выполнить неупорядоченную операцию.

Лемма 12.2 При увеличении массива попытке к таблице с n элементами для каждого элемента требуется $n/2$ операций сравнения (в среднем).

См. лемму 2.1. Доказательство применимо к массивам или спискам, упорядоченным для неупорядоченных.

Лемма 12.3 При увеличении списка n элементов n операция `add` требует n операций сравнения, операция `set` требует n операций сравнения для выполнения операции `set` для каждого элемента (в среднем).

Для утверждения операции для произвольной и неупорядоченной как массивом, так и списком, в чем легко убедиться, описывающиеся в реализации (см. упражнения 12.20 и упражнение 12.21).

Лемма 12.4. Для массива, обитурованного леммой 12.3, время для выполнения операции поиска в таблице символов, содержащей N различных элементов, требуется вычисление подлинности $N/2$ операций суммирования (в среднем).

См. лемму 12.2. И также эти утверждения справедливы для представлений с использованием как массивов, так и связанных списков. В чем возможно убедиться, заменив массив на реализацию (см. главу 12) и утверждение 12.21).

Построение упорядоченной таблицы путем последовательной вставки, по существу, эквивалентно выполнению алгоритма сортировки вставками, который был описан в разделе 6.2. Однако время, необходимое для построения таблицы, скрытого квадратичной зависимостью от количества элементов, поэтому вряд ли стоит использовать этот метод для построения большой таблицы. Однако при выполнении огромного количества запросов к небольшой таблице проверка упорядоченности элементов почти не спрашивается, поскольку в соответствии с леммами 12.3 и 12.4 этот процесс может в два раза уменьшить время, затрачиваемое на обитуровывание элементов при поиске. Эти элементы с дублированными ключами не должны храниться в таблице, дополнительные затраты на поддержание упорядоченности таблицы не столь велики, как может казаться, поскольку вставка выполняется только после обитуровывания элементов при поиске и, следовательно, время, затрачиваемое на вставку, пропорционально времени, затрачиваемому на поиск. С другой стороны, если элементы с дублированными ключами могут храниться в таблице, при поиске элементов неупорядоченной таблицы время выполнения операции может быть уменьшено в разы. Использование неупорядоченной таблицы рекомендуется для приложений, в которых выполняется огромный количество операций при очень большом числе запросов.

Помимо учета этих различий, приходится, как обычно, идти на компромисс: для реализации с использованием связанных списков таблица занимает значительный объем памяти для поиска, и то время, как для реализации с использованием массивов необходимо заранее знать максимальный размер таблицы или же предусмотреть увеличение таблицы с течением времени (см. также 14.5). Кроме того, как упоминалось в разделе 12.3, использование связанных списков обходится дороже, позволяющей эффективно решать другие операции типа *join* и *union*.

Эти результаты во взаимосвязи с другими алгоритмами, описанными ранее в этой главе и главах 13 и 14, обобщены в табл. 12.1. В разделе 12.4 рассматривается структура, при которой табличность требует поиска от количества элементов увеличивается до $O(N^2)$, в связи с чем он широко используется при работе со связанными таблицами (клетки таблицы взаимодействуют чрезвычайно редко).

Таблица 12.1 Затраты на вставку и поиск в таблицах символов

Значения в этой таблице представляют приблизительные значения для элемента базиса вложения как функции от количества элементов в таблице N и размера таблицы M (нормы Θ и Ω означают N), для таблиц, в которых норма элемента может зависеть независимо от наличия в таблице элементов с дублированными ключами. Эти значения являются значениями максимума (показаны стрелки) время выполнения, состоит из операций базисного, а время выполнения других операций зависит от времени на элемент так как размерная таблица, более сложные методы поиска требуют дополнительного времени от количества элементов или их количества. Время выполнения базисного элемента для поиска операций. Также не $O(N^2)$ с таблицей операции требуют времени затрат на сортировку элементов — минимально

записи, которую выполняет операция delete для неупорядоченного набора элементов — является лишь теоретической и не имеет практического значения (см. раздел 7.8). Значения, полученные звездочкой, относятся к крайне нежелательным худшим случаям.

	худший случай			средний случай		
	вставка	удаление	поиск	вставка	удаление	поиск
неупорядоченный список	1	1	1	1	1	1
упорядоченный список	N	N	1	$N/2$	$N/2$	$N/2$
неупорядоченный список	N	N	N	$N/2$	$N/2$	$N/2$
упорядоченный список	1	N	$N/2$	1	$N/2$	N
неупорядоченный список	1	N	$N/2$	1	$N/2$	N
двухсвязный список	N	N	1	$N/2$	$N/2$	$N/2$
двухсвязный список	N	N	N	$N/2$	$N/2$	$N/2$
двухсвязный список	$N/2$	$N/2$	$N/2$	$N/2$	$N/2$	$N/2$
двухсвязный список	N	N	N	$N/2$	$N/2$	$N/2$
двухсвязный список	1	N	$N/2$	1	1	1

В разделах 12.3–12.9 рассматриваются формы баз данных, которые обеспечивают определенную гибкость в том, что может храниться в базе данных и что может быть сделано с данными $\lg N$, но только в среднем. В главе 13 будут рассмотрены динамические формы типа «дерево», в которых количество элементов, хранящихся в базе данных, увеличивается в зависимости от количества элементов, либо существенно увеличивает простоту, либо. В главе 14 рассматриваются полные структуры, которые в среднем обеспечивают почти постоянное время поиска и вставки, но не обеспечивают эффективную поддержку операций sort и ряда других операций. В главе 15 будут рассматриваться методы горизонтального поиска, которые отличаются тем, что являются горизонтальными, описанными в главе 16. В главе 16 рассмотрены методы, применимые к файлам, которые хранятся внешне.

Упражнения

- 12.16 Добавьте операцию insert в приведенную разностную таблицу символов с использованием упорядоченного массива (программа 12.3).
- 12.17 Создайте функцию `searchIndex` для приведенных разностной таблиц символов с использованием списка (программа 12.4) и массива (программа 12.5). Функция должна выполнять в таблице поиск элемента с заданным значением, а затем вернуть элемент, если таковой в таблице отсутствует.
- 12.18 Создайте операцию `search` для приведенной разностной таблицы символов с использованием списка (программа 12.6).
- 12.19 Укажите количество операций сравнения, необходимых для помещения ключей `E A S Y Q U E S T I O N` в упорядоченную таблицу с использованием `ArrayList`, которая изначально соответствует одному из четырех типовых худших случаев упорядоченный или неупорядоченный массив или список. Примите, что для каждого ключа k в таблице `table`, `table[k]` — это значение `k` в таблице, вычитаемое из `k`, как в выражении (2.5).

- 12.20 Реализуйте операцию `insert`, описанную в главе 12 для интерфейса таблицы символов программы 12.2, используя для представления таблицы символов неупорядоченный массив. Характеристики производительности программы должны соответствовать табл. 12.1.
- 12.21 Реализуйте операцию `insert`, описанную в главе 12 для интерфейса таблицы символов программы 12.3, используя для представления таблицы символов упорядоченный массив. Характеристики производительности программы должны соответствовать табл. 12.1.
- 12.22 Напишите программу, реализующую таблицу символов с использованием списка (программа 12.6), чтобы они поддерживали дескрипторы элементов (см. упражнение 12.7): любые встраиваемые конструкции копирования и перегрузки операций присваивания (см. упражнение 12.8), любые операции `delete` и `new` для программ-драйвер, тестирующих реализацию интерфейса и реализацию ADL, а также любая таблица символов.
- 12.23 Создайте программу-драйвер проверки производительности, в которой функция `insert` используется для заполнения таблицы символов, а функция `search` используется для поиска элементов; эти операции должны выполняться несколько раз применительно к произвольным последовательностям элементов различной длины, от малой до большой. Программа должна измерять время, затрачиваемое на каждое выполнение, и выводить средние значения в виде текста или графика.
- 12.24 Создайте программу-драйвер проверки производительности, в которой бы функция `insert` использовалась для заполнения таблицы символов, а функция `search` использовалась, чтобы при каждом вызове элемента в таблице в среднем производилось $\log_2$ сканирований и примеров столько же применялось эти операции должны повторяться несколько раз применительно к произвольной последовательности ключей различной длины, от малой до большой. Программа должна измерять время, затрачиваемое на каждое выполнение, и выводить средние значения в виде текста или графика.
- 12.25 Создайте программу-драйвер, в которой функции из программы 12.2 для интерфейса таблицы символов используются применительно к реальному и к крайнему случаю, который может возникнуть в реальных приложениях. Простые функции не имеют $\log_2$ упорядоченных файлов, файлов, упорядоченных в обратном порядке, файлов с элементами как числами и файлами, которые состоят только из двух различных значений.
- 12.26 Какую реализацию таблицы символов описанную бы вы использовали для приложения, в котором в произвольном порядке выполняется 10^4 операций `insert`, 10^4 операций `search` и 10^4 операций `delete`? Обсудите ответ.
- 12.27 В действительности это упорядоченные элементы из главы 2 являются. Дайте ответ на вопрос упражнения 12.26 для пяти других принципов сортировки операций и их результатов.
- 12.28 Алгоритм самоорганизующегося поиска — это алгоритм, который изменяет порядок элементов так, чтобы чаще запрашиваемые элементы, скорее всего, находились в начале поиска. Напишите реализацию операции `insert` для упражнения 12.23, чтобы при каждом вызове при поиске был вычислен следующий элемент поменял найденный элемент в начале списка, перемещая из одну позицию вправо все элементы, расположенные после найденного элемента и освобождая место. Эта процедура называется алгоритмическим методом поиска.

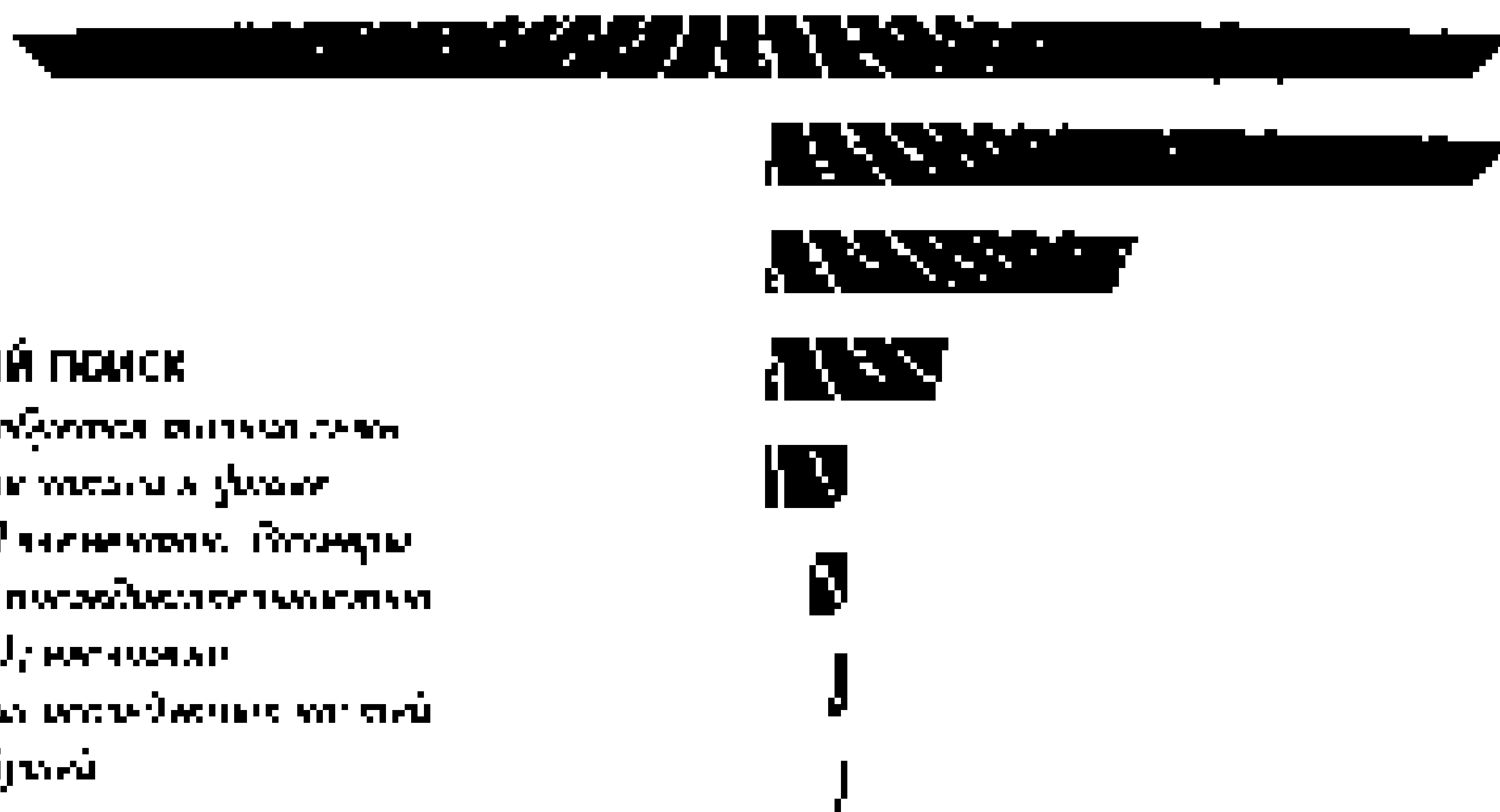


РИСУНОК 12.2 БИНАРНЫЙ ПОИСК

Для бинарного поиска требуется отсортированный массив для выполнения поиска и функция, которая возвращает до 2^{32} элементов. Вспомогательные функции позволяют получить элементы массива, а также, если требуется, извлечь элемент из массива. Вспомогательные функции позволяют получить элемент из массива.

Почему же рекурсивная функция в рисунке 12.2 завершается рекурсивным вызовом?

На рис. 12.1 показана подфункция, выполняемая в ходе бинарного поиска в небольшой таблице: на рис. 12.2 приведен больший пример. В ходе каждой итерации обрабатывается несколько элементов таблицы, поэтому количество требуемых итераций мало.

Лемма 12.5 Для бинарного поиска требуется не больше $\log_2 N$ и сравнений для поиска элемента в массиве из N элементов.

См. лемму 2.3. Интересно отметить, что максимальное количество сравнений, необходимое для бинарного поиска в таблице, размер которой равен N , в точности равен количеству бит в двоичном представлении числа N . Поскольку операция сравнения целого числа можно преобразовать в двоичное представление N и двоичное представление числа $\lfloor N/2 \rfloor$ (см. рис. 2.6).

Поддержка таблицы в отсортированном виде, как это делается при сортировке вставками, приводит к тому, что время выполнения становится квадратичной функцией от количества обрабатываемых элементов, но это количество может оказаться очень большим и тогда даже можно пренебречь тем количеством элементов, которое очень велико. В типичной ситуации, когда все элементы (или большая их часть) доступны для поиска, можно построить (или использовать) таблицу с помощью конструктора, который принимает массив в качестве аргумента и использует один из стандартных методов сортировки, описанных в главе 6 и последующих главах. Для поддержки таблицы во время выполнения поиска этого объектного таблицы может быть несколько различных способов. Например, можно поддерживать порядок во время выполнения, как это делается в программе `lib` (см. также упражнение 12.21), либо объединить статические элементы в массив, выполнить сортировку и обращаться к существующей таблице (как описано в упражнении 12.1). Вполне объективно может быть связано с таблицей элемент, ключ которого меньше или равен какому-либо из элементов таблицы, поэтому может потребоваться перемещение любого элемента с целью приближения к нему. Уменьшенная вероятность ошибки сортировки объектного таблицы — наилучший результат использования бинарного поиска. С другой стороны, существует от-

реинициализации приложений, в которых эти классы таблицы может быть другое отпар-
тирование и в этом случае, в зависимости от структуры данных, одобрения/защиты, а также
различий. Как программа 12.7, Сигаретный лонг пытается, наиболее предпочтитель-
ным методом

Если первые элементы требуются выводить динамически, может показаться, что для этого нужна сложная структура. Однако сам по себе связный список не позволяет оценить эффективность реализации. Поскольку эффективность бинарного поиска зависит от возможности быстро перейти к середине списка, то работа программы через `LinkedList` — это относительно сложная задача. Для суммирования эффективности бинарного поиска и сложности связного списка требуется более сложная структура данных, которая обманом поддается в выборе.

Если в таблице присутствуют дублированные строки, бланковый шаблон можно расширить до подзаголовка, применив на таблице сжатие для подсчета количества элементов с данным значением или вернуть их в виде группы. Поскольку элементов, значения которых совпадают с значениями, образуют в таблице непрерывный блок (плоскую таблицу), в программе 12.2 успешно можно использовать то же меню «Элементы». Если пользователю требуется доступ ко всем элементам подобного ряда в программе, можно добавить код для заполнения параметров в объектах подзаголовка от точки завершения блока, в таком случае выводится два независимых, ограниченных элемента, значения которых равны текущему значению. В этом случае время выполнения шаблона составит приблизительно 1/3 от времени выполнения шаблона. Аналогичный шаблон используется для решения более общей задачи нахождения значения, которое состоит из заданного ряда элементов, значения которых попадают в заданный интервал. Подобные шаблоны можно использовать для создания в таблице элементов, значения которых находятся в заданном диапазоне.

Приложение 12.7 Билетный поиск (в таблице отобразятся отклонения на будущее)

[illegible]

Массив должен быть в сортированном. Эта функция может использоваться в качестве замены функции `sort` из программы 12.5, которая обеспечивает динамическую сортировку во время работы. Массив `data` может быть передан в виде строки таблицы символов, который получен бы массив в качестве аргумента, а затем этот массив преобразован в строку. Этот вариант может использоваться в функции `sort` для более гибкого применения одной из стандартных процедур сортировки.

END PAGE.

```

Item searchR(int l, int r, Key v)
{ if (l > r) return nullItem;
  int m = (l+r)/2;
  if (v == st[m].key()) return st[m];
  if (l == r) return nullItem;
  if (v < st[m].key())
    return searchR(l, m-1, v);
  else return searchR(m+1, r, v);
}

public:
Item search(Key v)
{ return search(0, N-1, v); }

```


Последовательность сравнений, выполняемая алгоритмом бинарного поиска, представлена: конкретная последовательность используется в зависимости от значения искомого элемента x и значения N . Сравнения могут быть описаны в виде структуры бинарного дерева, подобной приведенной на рис. 12.3. Это дерево аналогично использованному в главе 4 для описания разбора پوش-файлов во время сортировки шкатулки (рис. 3.3). В бинарном списке не существует один путь к дереву, тогда как при сортировке списка — все пути. Это дерево является статическим и неизменным: в разделе 12.5 будут рассмотрены алгоритмы, в которых для поиска искомого элемента используется динамическая, или построенная структура бинарного дерева.

На верхней диаграмме показан поиск в файле, состоящем из 15 элементов, проиндексированных от 0 до 14. Мы проиндексировали средний элемент (индекс 7), затем (рекурсивно) проиндексировали левую половину, если искомого элемента меньше него, и правую — если он больше. Каждый поиск соответствует пути, проложенному по дереву сверху вниз; например, поиск элемента, который находится между элементами 10 и 11, использовал бы просмотр 7 элементов (0, 11, 9, 13, 10, 14) в файле с размером, заданном на 1 элемент: средняя часть 2, последовательность не стоит симметрично, что невозможно заметить на нижней диаграмме, нарисованной для 12 элементов.

Одно из возможных усовершенствований бинарного поиска — более точное представление о разрыве между левым полем в текущем наборе данных (вместо среднего значения его со средним элементом на каждом шаге). Эта техника имитирует способ поиска имени в телефонном справочнике или слов — в словаре: если название написано буквами, находящимися в начале шифра, поиск закончится в начале книги, но если она начинается с буквы, находящейся в конце шифра, поиск закончится в конце книги. Для реализации данного метода, приведенного формулировки поиска (интерпретация текста), приведенная 12.7 подразумевает изменение следующей формулы оператора:

$$x = (1 + x) / 2$$

заменяется на оператор

$$x = 1 + (x - 1) * \text{key} / ((x - 1) * \text{a}[x] * \text{key}() + \text{a}[x] * \text{key}())$$

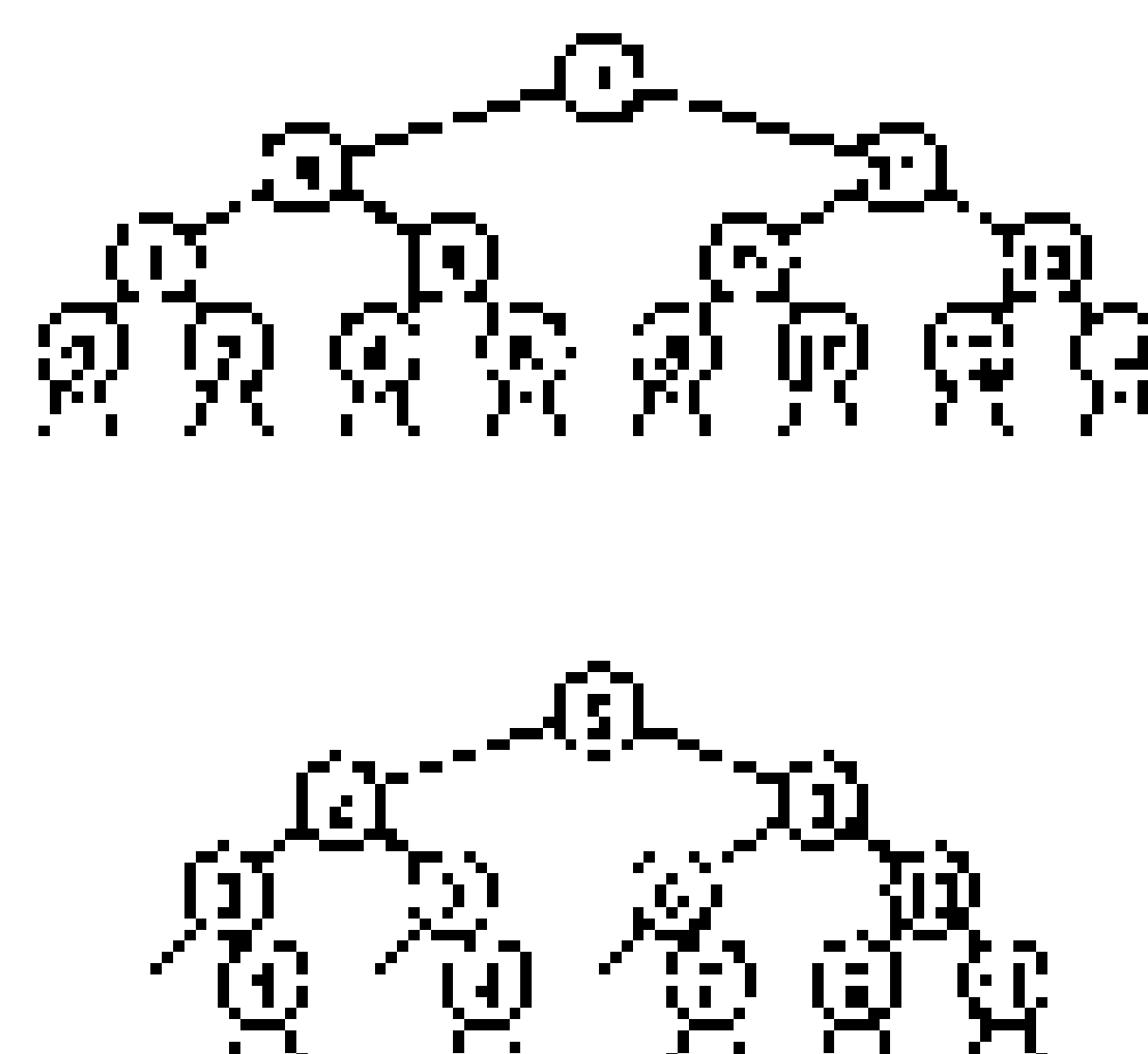


РИСУНОК 12.3
ПОСЛЕДОВАТЕЛЬНОСТЬ
СРАВНЕНИЙ ПРИ БИНАРНОМ
ПОИСКЕ

На этой диаграмме бинарное дерево "разбито на части" поочередно по индексам от 0 до 14, чтобы показать путь, проложенный при поиске элемента.

Для иллюстрации этого метода поиска на рисунке, в файле файла с индексацией элементов (0-14) в файле. Это дерево может быть использовано для поиска, например, статических данных, статических данных, статических данных и статических данных (рис. 3.3 и 8.3), а также для поиска элементов и для поиска элементов.

Для объяснения и численных значений, что выражение $(1 + 1) / 2$ равнозначит выражению $1 + \frac{1}{2} (0 - 1)$, мы начинаем с левого интервала, добавляя к левой граничной точке половину размера интервала. Исполнователе интерпретатора пока не дано в языке в этой формуле коэффициента $\frac{1}{2}$ ожидаемой позиции курсора — а именно $(x - \hat{x}) / (\hat{k} - \hat{x})$, где $\hat{k}$ и $\hat{x}$ означают значения $a[l].key()$ и $a[r].key()$, соответственно. При этом предполагается, что значения ключей аппроксимируются числами и равномерно распределены.

Можно показать, что при интерполяционном поиске в файлах с пропорциональными ключами или *хешами* (то есть с произвольной прямой) используется время $O(\log N)$ операций. Докажем это утверждение с помощью леммы из главы этой книги. Эта функция растет очень медленно и на практике ее можно считать постоянной: если N равно 1 миллиарду, то $O(\log N) < 5$. Таким образом, любой элемент можно найти, выполнив только несколько обращений (в среднем — это существенно быстрее по сравнению с бинарным поиском, для ключей, которые распределены не линейно пропорционально). Прогнозируемость интерполяционного поиска еще выше. Действительно, граничный случай является веткой поиска в непостоянных индексациях по ключам, описанный в разделе 12.2.

Однако интерполяционный поиск в значительной степени оптимизируется из предположения, что ключи равномерно распределены во всем интервале. Более того равномерно — в простом случае, что обычно и имеет место на практике, ветка окажется неэффективной. Кроме того, для его реализации требуются дополнительные вычисления. Для небольших значений N значение обычного бинарного поиска ($O(\log N)$) достаточно близко к значению интерполяционного поиска ($O(\log N)$), в котором придется использовать последний метод. С другой стороны, интерполированный поиск определенно заслуживает внимания при работе с большими файлами, в приложениях, в которых сравнения выполняются относительно часто, и при использовании платного метода, сопряженного с большими затратами на доступ.

Упражнения

- 12.34 Реализуйте рекурсивную функцию бинарного поиска (см. программу 12.7).
- 12.35 Нарисуйте деревья, которые соответствуют рис. 12.3 для $M = 17$ и $N = 24$.
- 12.36 Нарисуйте для N , для которых бинарный поиск и таблица символов размером N занимают 10, 100 и 1000 раз больше места, чем хеш-таблица. Предложите значения эмпирически и проведите их экспериментально.
- 12.37 Предположите, что попытки и численность таблицы символов размера N решаются методом сортировки вставками, но для вычисления отсортированных элементов бинарный поиск. Предположите, что поиск выполняется в 1000 раз чаще, чем попытки. Определите и проценты числа промахов, затрачиваемых на попытку, для $N = 10^3, 10^4, 10^5$ и 10^6 .
- 12.38 Разработайте реализацию таблицы символов, в которой используется бинарный поиск и "тепловая" таблица, а также поддерживаются операции *вставка*, *удаление*, *поиск* и *лист*, проводимый в следующей структуре. Храните большой массив

размещенный массив для основной таблицы символов и регулярный массив для массивов установленных элементов. При вызове функции `search` отсортируйте массивы символов и элементов (или наоборот, в зависимости от реализации), объедините их с основной таблицей, а затем выполните бинарный поиск.

12.39 Напишите функцию `search` в зависимости от упражнения 12.38.

12.40 Ответьте на вопрос упражнения 12.37 для реализации из упражнения 12.38.

• 12.41 Реализуйте функцию `search` для бинарного поиска (упражнение 12.37), которая работает со всеми элементами в таблице символов с учетом размещенного.

12.42 Создайте программу, которая при заданном значении N должна производить массив N макроструктур, произвольных от 0 до $N-1$, в формате `struct { int i; char *p; }`, где i — инструкция в списке означает "сравнить элемент поиска со значением i в таблице; затем сообщить о результате при поиске, если они равны, вывести i по инструкции, если он меньше, i по инструкции, если он больше" (случае 0 эмуляруется на случай провала при поиске). Необходимо, чтобы все макроструктуры обладали тем свойством, что для любого поиска должно выполняться столько же операций сравнения, как и при бинарном поиске в этом же наборе данных.

• 12.43 Расширьте макрос, созданный в упражнении 12.42, чтобы производился вывод машинного кода, выполняющий бинарный поиск в таблице размером N при наименьшем возможном количестве машинных инструкций на одно сравнение.

12.44 Предположите, что $a[i] = 10^i$ для $1 \leq i \leq N$. Сколько позиций в таблице исследуется при бинарном поиске по прежнему неудачного поиска $2k+1$?

• 12.45 Найдите значение N , для которого интервальный поиск в таблице символов размером N выполняется в 1, 2 и 10 раз быстрее бинарного поиска, при условии произвольности ключей. Продолжите значения N и проверьте их экспериментально.

12.5 Деревья бинарного поиска

Для решения проблемы минимизации затрат на установку в качестве основы используются деревья символов, и поэтому они являются естественной структурой.

Лежащая в основе структура данных позволяет разобьять алгоритмы с высокой средней пропускной способностью операций ввода-вывода. Такие данные существуют в виде массивов, но между ними и компьютерными науками существует одна из наиболее фундаментальных.

В главе 5 уже рассматривались деревья, тем не менее, только в них не понимались полностью. Они более известны, чем деревья (или как они называются), — это каждый узел указывает только на один другой узел, поэтому они являются деревьями. Определяется множество бинарного дерева — множество узлов, для которых $1 \leq i \leq n$ и i — это узел, указывающий на другой дополнительный узел или на один из узлов системы узлов, которые не имеют детей. Узлы в данной системе называются также *внутренними узлами* (или *узлами*). Для вычисления только каждый внутренний узел имеет только одного или нескольких детей, и все эти внутренние узлы называются *внутренними узлами*. Для поиска каждого внутреннего узла требуется только один шаг поиска и указывают на конкретный поиск.

Лемма 12.1 *Дерево бинарного поиска (BST) — это бинарное дерево, в котором из отмеченных узлов каждого узла есть левый, причем левый в любом узле больше (или равен) ключу в его материальном дереве бинарного поиска, а правый узел меньше (или равен) ключу от его материального дерева бинарного поиска.*

В программе 12.5 BST деревья используются для реализации операций `search`, `insert`, `delete` и `find`. В первой части реализации узлы в BST-дереве представляются как структуры элемента (с ключом), левую и правую ссылки. Кроме того, для обеспечения эффективности реализации операции `find` программа поддерживает поле, содержащее количество узлов в дереве. Левая ссылка указывает на BST-дерево в экземпляре с меньшим типом (размером) ключа, а правая — на BST-дерево с элементами с большим (или равным) ключом.

Программа 12.5 Таблица символов на базе дерева бинарного поиска

В этой реализации функции `search` и `insert` используют приватные рекурсивные функции `searchR` и `insertR`, которые непосредственно отражают рекурсивное распределение BST-дерева. Обратите внимание на использование аргумента ссылки в функции `insertR` (см. стр. 160). Ссылка `head` указывает на корень дерева.

```
template <class Item, class Key>
class ST
{
private:
    struct node
    {
        Item item; node *l, *r;
        node(Item x)
            { lkey = x; l = 0; r = 0; }
    };
    typedef node *link;
    link head;
    Item nullItem;
    Item searchR(link h, Key v)
    {
        if (h == 0) return nullItem;
        Key t = h->item.key();
        if (v == t) return h->value;
        if (v < t) return searchR(h->l, v);
        else return searchR(h->r, v);
    }
    void searchR(link& h, Item x)
    {
        if (h == 0) { h = new node(x); return; }
        if (h->key() < h->item.key())
            insertR(h->l, x);
        else insertR(h->r, x);
    }
public:
    ST(int maxN)
        { head = 0; }
    Item search(Key v)
        { return searchR(head, v); }
    void insert(Item x)
        { insertR(head, x); }
};
```


последняя до тех пор, пока пусто, и устанавливается равная ссылке на новый узел, содержащий элемент; если новая ссылка меньше предыдущей — корень, а иначе — добавляется в левое поддерево. В противном случае элемент устанавливается в правое поддерево. То есть, каждый новый элемент добавляется только к уже легшему рекурсивному вызову, когда устанавливается новая ссылка. В разделе 12.9 мы увидим, что будут рассмотрены более сложные двудольные структуры, которые естественным образом представляются с помощью той же рекурсивной схемы, на которой часто строится основной алгоритм.

На рис. 12.5 и 12.6 показаны примеры создания BST-деревьев с использованием последовательности данных в первоначальное пустое дерево. Визуальное представление связано с тем, что узлы и ссылки на них в новой части дерева, а не старая структура дерева никак не взаимодействуют. Поскольку каждый узел имеет две ссылки, дерево растет скорее в ширину, нежели в высоту.

При использовании BST-деревьев реализованная функция sort требует considerable объема дополнительной работы. Построение BST-дерева сводится к сортировке элементов, поскольку при сопоставлении элементов расширения BST дерево представляется собой бинарный поиск. На приведенных выше рисунках можно наблюдать наглядное описание алгоритма (если не обратить внимания на их расположение по высоте и связи). Проявляя робость только со значениями, но строгий поперечный обзор, мы определяем, присутствуют ли элементы в том узле, что демонстрирует рекурсивную реализацию функции `showR` в программе 12.9. Для отображения элементов в BST-дереве в порядке их ключей мы отображаем элементы в левом поддереве и левее их ключей (рекурсивно), затем корень, и далее элементы в правом поддереве в порядке их ключей (рекурсивно).

Программа 12.9 Сортировка с помощью BST-дерева

Понимая рекурсивный способ BST-дерева, можно использовать в порядке следования их ключей. В этой реализации функция `showR` принимает значения `low` и `high` для вывода элементов в порядке следования их ключей.

```
private:
    void showR(int low, int, int high, int cur)
    {
        if (hi == 0) return;
        showR(hi-1, cur);
        h->data.show(cur);
        showR(h->r, cur);
    }
public:
    void show(const int low)
    { showR(low, 0, 0); }
```

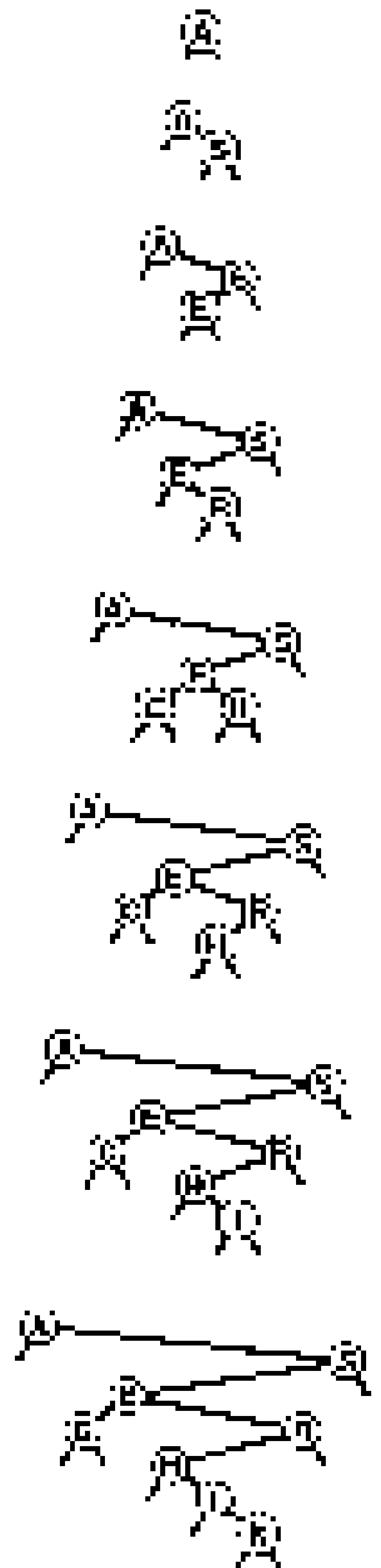


РИСУНОК 12.5 СОЗДАНИЕ ДЕРЕВА БИНАРНОГО ПОИСКА

Для создания бинарного дерева бинарного поиска из последовательности данных A D E B C F G и формирования нового BST-дерева. Если для каждого элемента установить его позицию при сортировке, то можно создать дерево.


```

if (v < p->data.key()) p->l = new
node(N);
else p->r = new node(x);
}

```

Функция BST-дерева в программе 12.4 не проверяет явное наличие элементов с дублированными ключами. Если веточке какого-либо узла, к которому относятся какой-либо ключ, уже вставленным в дерево, узел помещается справа от присутствующего в дереве узла. Одним из побочных эффектов подобного поведения является то, что узлы с дублированными ключами не отображаются в дереве последовательно (см. рис. 12.7). Однако, на самом деле, группировка поиска с точкой, в которой функции `key()` возвращают первое совпадение, потерю не несет, так как связь 0. Как упоминается в разделе 9.1, существует несколько других возможностей обработки элементов с одинаковыми ключами.

Дерево бинарного поиска является быстрой сортировкой. Узлы в корневом дереве соответствуют разделенным элементам при быстрой сортировке (ни какие-либо элементы слева от него не могут быть больше, и никакие элементы справа не могут быть меньше него). В разделе 12.6 будет показано, как это соотношение связано с инвариантами свойств дерева.

Упражнения

- 12.46 Нарисуйте BST-дерево, образующееся при вставке элементов с ключами E A S Y Q U L T I O N в первоначально пустое дерево.
- 12.47 Нарисуйте BST-дерево, образующееся при вставке элементов с ключами E A S Y Q U E S T I O N в первоначально пустое дерево.
- 12.48 Укажите количество сравнений, необходимых для нахождения ключей E A S Y Q U L E S T I O N в первоначально пустом дереве, считая использование BST-дерева. Считайте, что операции `key()` выполняются для каждого ключа, чтобы он мог выполняться операция `less` для каждого элемента при поиске, как имеет место в программе 12.3.
- 12.49 Вставка ключей в порядке A S E R T I N G C в первоначально пустое дерево также даст важную часть дерева, показанного на рис. 12.6. Приведите десять других возможных порядков этих ключей, которые обеспечат тот же результат.

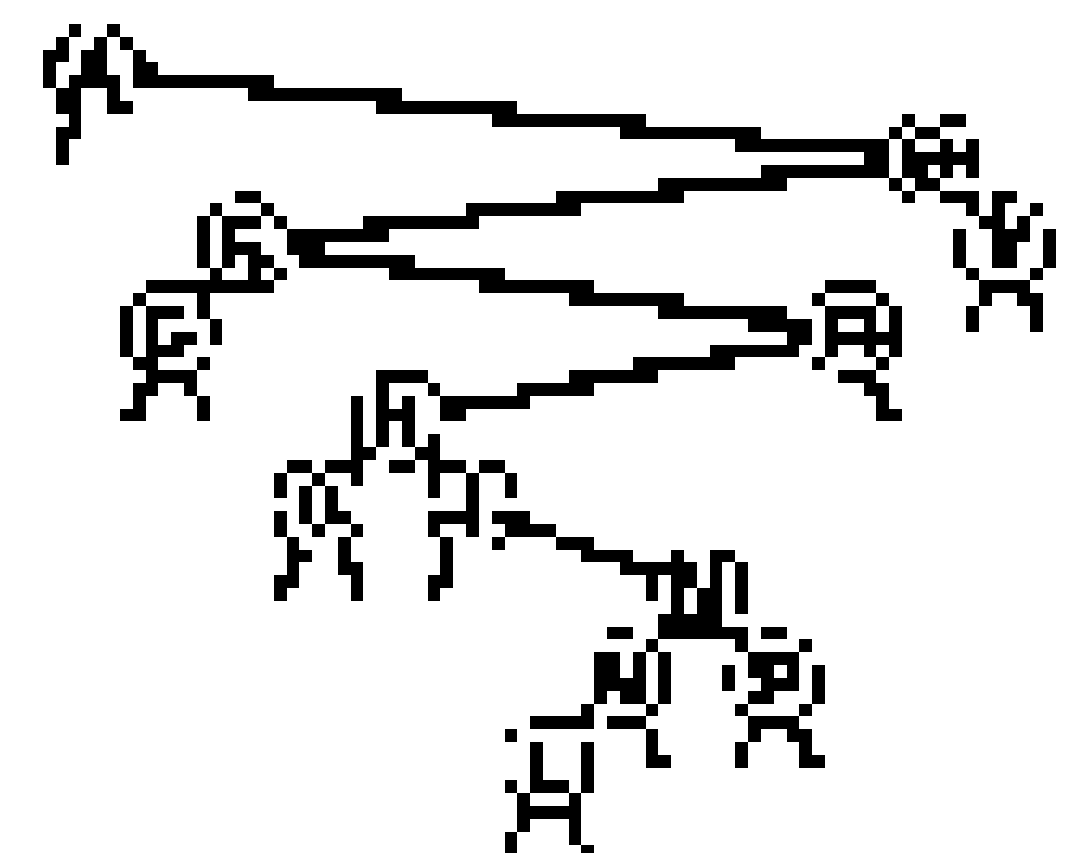
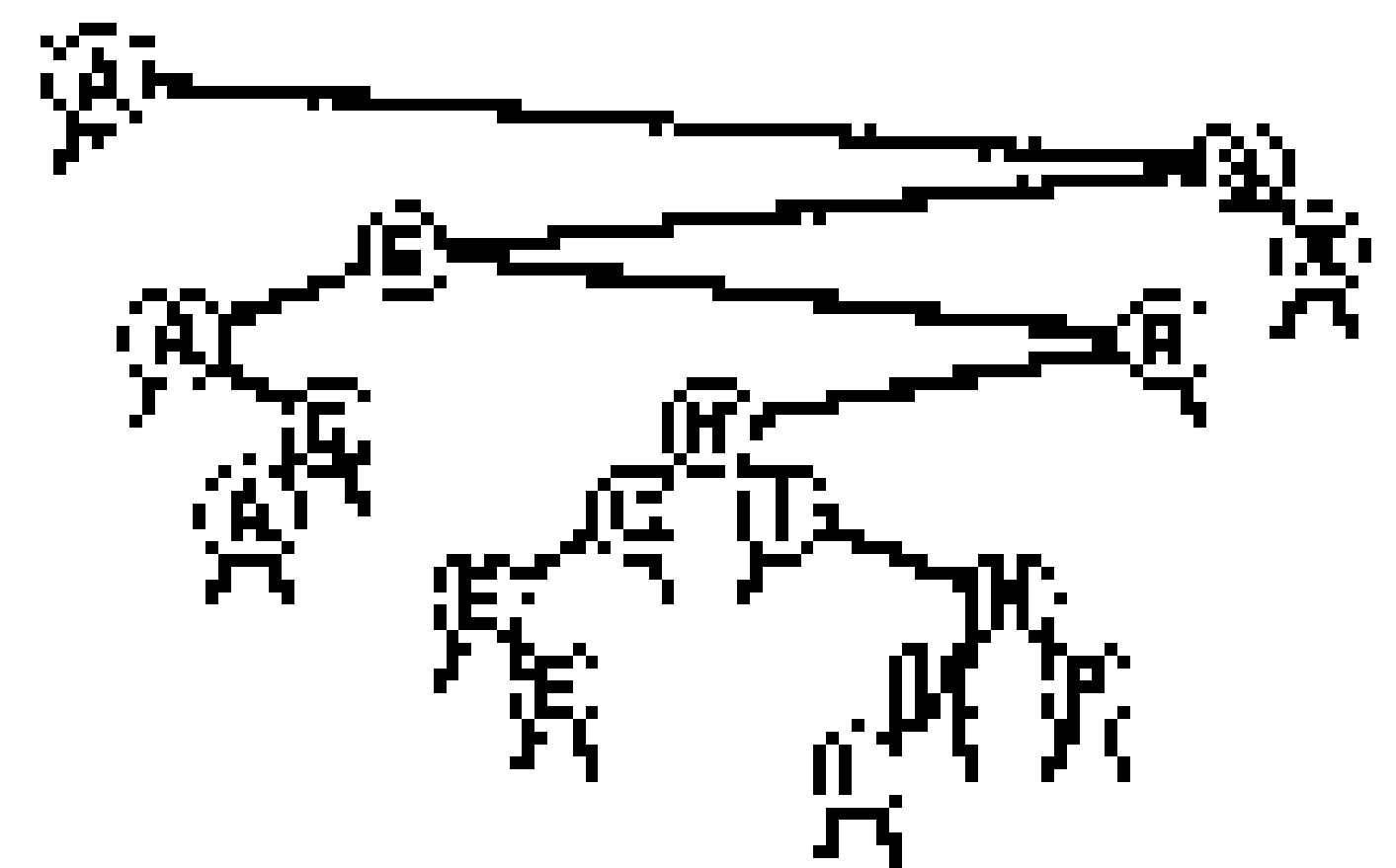


РИСУНОК 12.7 ДУБЛИРОВАННЫЕ КЛЮЧИ В ДЕРЕВЬЯХ БИНАРНОГО ПОИСКА

Ключи BST-дерева образуют список, дублирующийся каждый раз, когда ключ встречается повторно. Если ключ встречается повторно, то он вставляется в дерево, как элемент, который больше, чем элемент, с которым он встречается повторно. Если ключ встречается повторно, то он вставляется в дерево, как элемент, который больше, чем элемент, с которым он встречается повторно. Если ключ встречается повторно, то он вставляется в дерево, как элемент, который больше, чем элемент, с которым он встречается повторно. Если ключ встречается повторно, то он вставляется в дерево, как элемент, который больше, чем элемент, с которым он встречается повторно.

12.50 Реализуйте функцию `searchInsert` для BBT-деревьев (программа 12.4). Функция должна вернуть в тайкисе индекс или элемент, с таким же значением, как и `value` в `value`, а также вернуть `0` или `1`, если `value` больше или меньше `value`.

- 12.51 Создайте функцию, которая возвращает количество элементов в BBT-дереве, имеющих ключ, равный `value`.

12.52 Предположите, что вы хотите реализовать поиск, поскольку часто требуется выполнить обращение к элементу поиска в бинарном дереве. Давая ли ключи элементам в дереве в форме `key` (элементы `key` и `value` являются элементами дерева)? Обсудите ответ.

- 12.53 Упростите код поиска и вставки в реализацию BBT-дерева программы 12.4, используя следующие функции: `find`, содержащий элемент с заданным значением `value`, который возвращает элемент и путь к нему, указывая на корень дерева; и `insert`, содержащий элемент с заданным значением `value`, который возвращает элемент `value` и путь к нему, указывая на корень дерева. (См. также 1.1).

12.54 Измените реализацию BBT-дерева в программе 12.5 для хранения элементов в бинарном дереве. Введите ключи в список, размещенный в узлах дерева. Измените интерфейс, чтобы вернуть элемент `value` (необходимо вернуть элемент `value` и элемент `key`).

12.55 В неструктурированной процедуре поиска в программе 12.10 для определения того, какую часть узла необходимо заменить новым узлом, используется избыточное количество ресурсов. Подумайте, как можно изменить эту процедуру, чтобы избежать избыточного указателя на узлы.

12.6 Характеристики производительности деревьев бинарного поиска

Время выполнения алгоритма обработки BBT-дерева зависит от формы дерева. В лучшем случае дерево может быть полностью сбалансированным и содержать примерно $\log_2 N$ узлов между корнем и каждым из листовых узлов, но в худшем случае в каждом из узлов дерева может содержаться N узлов.

Можно также сказать, что в среднем время поиска также будет связано с количеством узлов в дереве, поскольку первый непустой элемент становится корнем дерева. Если N ключей должны быть найдены в произвольном порядке, то этот элемент должен быть ключом (в среднем), что даст безлокальную зависимость времени поиска (при использовании аналогичных рассуждений применительно к последовательности). Действительно, возможен случай, когда BBT-дерево превращается в список, что эквивалентно тому же алгоритму, что и бинарный поиск (см. упражнение 12.58). Однако если бы был найден алгоритм для данного алгоритма, то можно было бы доказать, что время выполнения для всех видов поиска. В действительно произвольной ситуации дерево может быть любой формы, и поэтому полностью сбалансированное дерево является исключительно редким случаем. Действительно, без ссылок каждый не удастся обработать дерево полностью сбалансированному после каждой вставки. Однако полностью несбалансированные деревья также встречаются редко при пре-

нета пути в таком дереве равно просто $2 \cdot N$ (см. лемму 12.7), обуславливая линейный рост. В любом BST-дереве среднее количество сравнений, необходимых для выявления истинной или ложной границы, приблизительно на 1 больше среднего количества сравнений, необходимых для выявления ложной или истинной границы.

В соответствии с леммой 12.6 следует ожидать, что затраты на поиск для BST-деревьев должны быть приблизительно на 30% выше затрат для бинарного поиска в отсортированных массивах. Но в соответствии с леммой 12.7 дополнительные затраты не возникают, поскольку левый или правый поддеревья почти при тех же затратах — почти такая же сложность при использовании бинарного поиска не доступна. На рис. 12.8 показано BST-дерево, полученное в результате деления массива произвольных перестановок. Хотя оно содержит несколько длинных и несколько коротких путей, его можно считать хорошо сбалансированным: для выполнения того же поиска требуется около 12 сравнений, в среднем количество сравнений, необходимых для обнаружения произвольного элемента при поиске равно 7.86, что сравнимо с 3.74 для случая бинарного поиска.

Леммы 12.6 и 12.7 определяют производительность для лучшего случая при условии, что левые или правые поддеревья сбалансированы. Если это не так, производительность алгоритма имеет тенденцию к снижению.

Лемма 12.8 В худшем случае для поиска в дереве бинарного поиска с N ключами может потребоваться N сравнений.

На рис. 12.9 и 12.10 показаны два примера худшего случая BST-деревьев. Для этих деревьев поиск с использованием бинарного дерева ничем не лучше последовательного поиска с использованием отсортированных списков.

Таким образом, высокая производительность алгоритма реализации бинарного поиска с использованием BST-деревьев требует, чтобы ключи в достаточной степени были подобны произвольным ключам, а дерево не содержало длинных путей. Кроме того, худший случай не стал уж редким встречается на практике — он возникает при записке ключей в первоначально пустое дерево по порядку или в обратном порядке с применением стандартного алгоритма последовательности операций, которую мы определенно можем предпринять, не получив никакого явного предупреждения не делать этого. В главе 13 исследуются технологии предотвращения этого худшего случая и крайне маловероятный и полностью исключенный, предупреждения всех деревьев и подобных деревьев для лучшего случая, длины всех путей в которых гарантированно ограничиваются логарифмической сложностью.

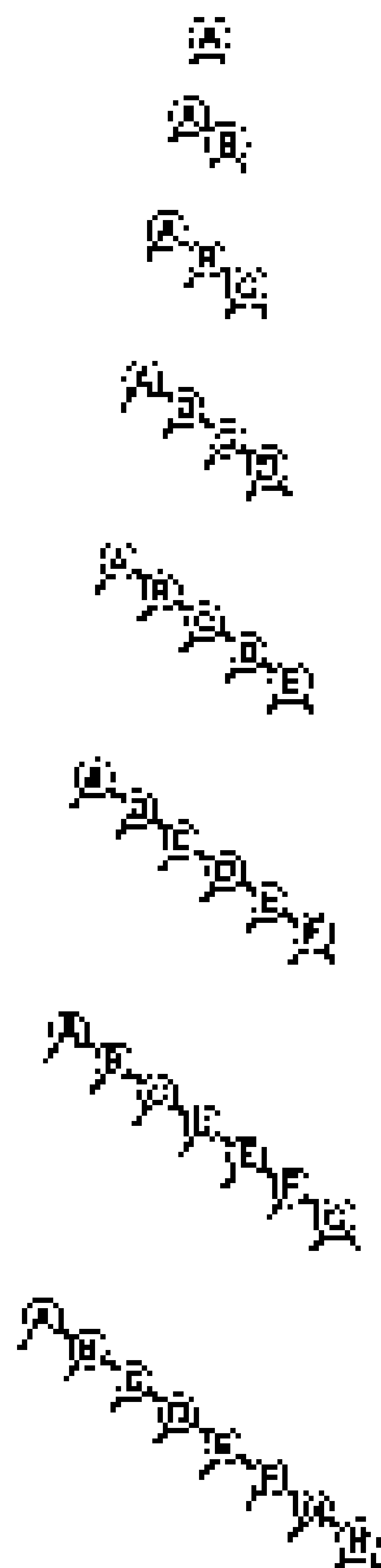


РИСУНОК 12.9 ХУДШИЙ СЛУЧАЙ ДЕРЕВА БИНАРНОГО ПОИСКА
Если ключи в BST-дереве добавляются в порядке возрастания, или в порядке убывания (или, что эквивалентно, в порядке убывания или в порядке возрастания), то дерево превращается в отсортированный список, и бинарный поиск превращается в последовательный поиск.

Нельзя из других рассмотренных рассмотренных таблиц символов может использоваться для анализа или поиска в таблице. В таблице 2^{2^N} содержится 2^N различных ключей, в этом смысле каждая из них — время выполнения каждой из методов, описанных в разделе 12.1.2, определяется асимптотической сложностью. Более того, из анализа следует, что среднее расстояние до узла в бинарном дереве пропорционально логарифму количества узлов в дереве, что позволяет эффективно выполнять следующие наборы операций: поиск, удаление и вставка операций с АТД. Таблица символов, но и будет вскоре показана.

Упражнения

12.56 Составьте рекурсивную программу, которая вычисляет максимальное количество сравнений, требуемых для поиска узла в данном БСТ-дереве (дерево дерева).

12.57 Составьте рекурсивную программу, которая вычисляет максимальное количество сравнений, требуемых для нахождения пути поиска в данном БСТ-дереве (дерево внутреннего пути дерева, ведущего на N).

12.58 Приведите все возможные варианты узловых ключей $E, A, N, Y, Q, U, E, S, T, I, O, N$ и нарисуйте для каждого пустое БСТ-дерево, чтобы убедиться при этом, что дерево было эффективно бинарно по отношению к символу последовательности сравнений. Выполняйте поиск любого узла в БСТ-дереве и при бинарном поиске.

12.59 Составьте программу, которая построит набор узлов и бинаризирует пустое БСТ-дерево так, чтобы данное дерево было эффективно бинарно по отношению, как описывалось в упражнениях 12.58.

12.60 Нарисуйте все различающиеся по структуре БСТ-деревья, удовлетворяющие тем же условиям, что и в предыдущем примере, для $2 \leq N \leq 5$.

12.61 Определите вероятность того, что каждый из узловых ключей упражнения 12.60 является внутренним узлом в каком-либо из различных деревьев в перечисленном наборе деревьев.

12.62 Сколько бинарных деревьев, удовлетворяющих N условиям, имеют высоту N ? (каково существует различных способов вставки N различных узлов в первоначально пустое дерево, позволяющих образованное БСТ-дерево иметь высоту N ?)

12.63 Докажите с использованием метода индукции, что различие между длиной внешнего и внутреннего путей в любом бинарном дереве составляет $2N$ раз, где N —

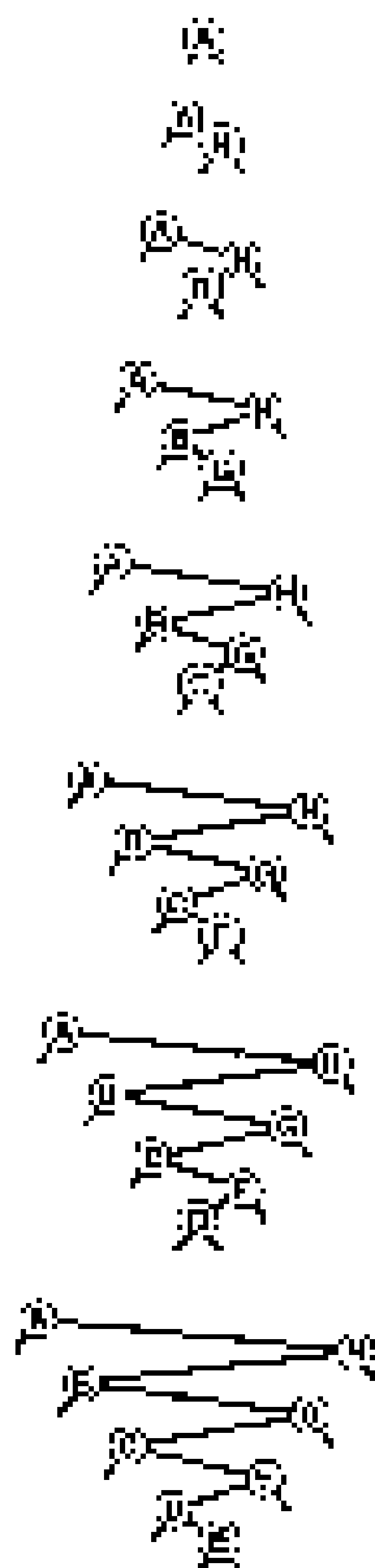


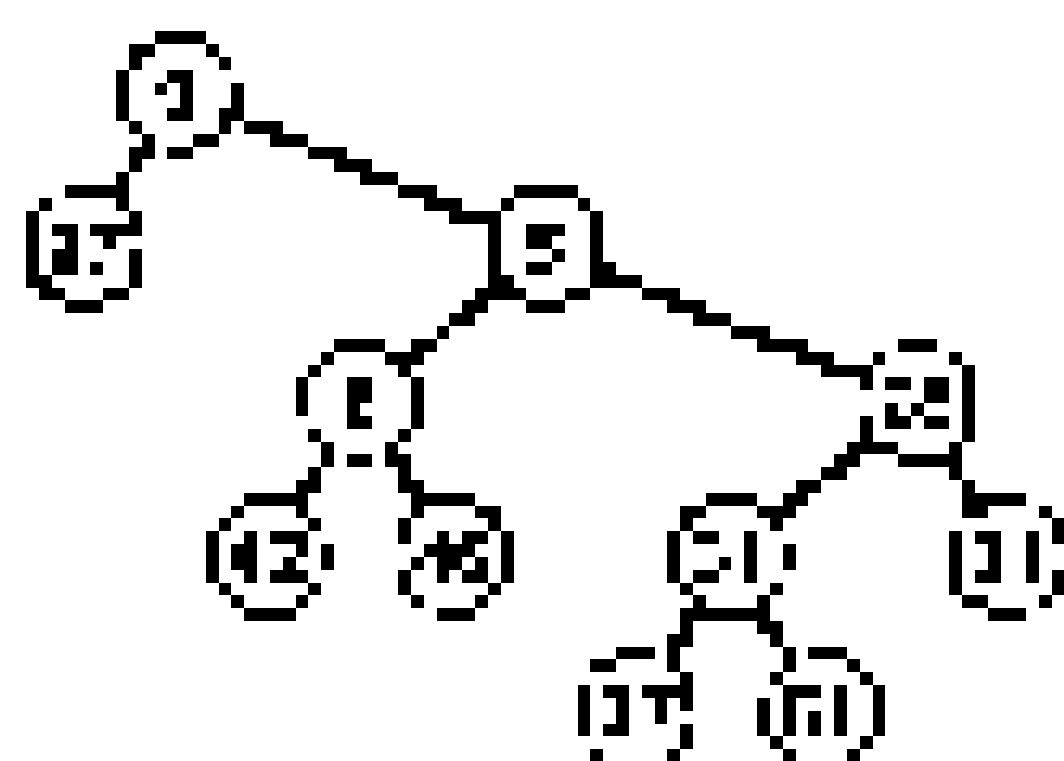
РИСУНОК 12.12 ЕЩЕ ОДИН ХУДШИЙ СЛУЧАЙ ДЕРЕВА БИНАРНОГО ПОИСКА

Может быть другим способом построения дерева, удовлетворяющего тем же условиям, что и в предыдущем БСТ-дереве. Тем не менее, дерево, показанное здесь, эффективно показывает, что для любого набора узлов, удовлетворяющего условиям, существует дерево, удовлетворяющее условиям, которое имеет высоту $2N$.

те менее, они всегда готовы для использования гаджетомеханикой языка, не требуя дополнительного времени или распределения. Другая важная особенность этого языка заключается в том, что он допускает добавление дополнительных массивов (содержащих дополнительные данные) к каждой точке информации без какого-либо изменения кода манипулирования данными. Такая перестройка также обеспечивает индекс элементов, что предоставляет способ немедленного доступа ко всей информации, связанной с этим элементом, путем использования индекса для доступа к соответствующему массиву.

Таким образом, реализация ВЭП-сервисов как «раз-
ска» упрощения поиска в больших массивах элементов
назад весьма полезна, поскольку исключает доста-
точно высокие затраты на копирование элементов во
внутреннее представление АТД и перегрузку стан-
ции с расходом времени и концентрации пользователя.
Использование массивов не подходит, когда
объем памяти требует достаточно большой, а таблица
символов увеличивается и увеличивается в значитель-
ных пределах. Поэтому так актуально, если та-
рханс трудно представить максимальный размер таб-
лицы символов. Если невозможно то по предель-
ному, неиспользуемые слова могут принадлежать по-
пулярному речевому набору и должны иметься в со-
блюдении.

Важное преимущество концепции индексирования — обеспечение быстрой работы слов в строке текста (см. рис. 12.11). В программе 12.11 показан пример такой реализации. Она считывает текстовую строку из внешнего файла. Затем, просматривая каждую позицию в текстовой строке с целью определения конца строки от данной позиции до конца строки, она вставляет в указчик в таблице символов, используемых указателем на строки. Подобное применение указателя строки отличается от описанного выше хранения элементов, как в упражнении 12.2, поскольку указчик распределяется на память под область хранения не требуется. Необходимо знать только длину, но мы поддерживаем только «каждый тип количества символов, которое требуется для строки 101+1: содержать 10000». Нет необходимости хранить длину, но если изменить операцию, что одна является префиксом строки, можно



0 call me instead some ...
5 be instead some year ...
8 instead some years ...
16 two years ago never ...
21 some ago never mind ...
26 ago never mind how ...
31 never mind how long ...
37 mind how long previous ...
42 how long previously ...
46 long previously instead ...
61 previously having lit ...

ЧИСЛОК Д.Н. ИНДЕКС
ТЕКСТОВОЙ СТРОКИ

[illegible]

любой символ для выяснения того, присутствует ли данный символ в тексте, и простых подстрок функции `search`.

Программа 12.11 Пример индексирования текстовых строк

В этой программе предполагается, что `Item`.hxx определяет представление для `Item` для хранения с р(к-значениях, перегруженные операторы `operator<` и `operator>` позволяют `string`, перегруженный оператор `operator==`, которая использует `string` и оператор `or` преобразования `Item` в `string` (`Item to string`). Главная программа открывает текстовый файл из указанного файла и использует таблицу символов для построения индексов из строк. Определенные начальные символы в таблице с р(к. Затем она выполняет запрошенные строки из стандартного ввода и выводит результаты, в которых выводится строка для каждой строки по `find()`. При реализации таблицы символов с использованием BBT задача может выполняться быстрее даже для очень больших `string`.

```
#include <iostream.h>
#include <string.h>
#include "Item.hxx"
#include "BT.hxx"
static char text[10000];
int main(int argc, char *argv[])
{
    int N = 0, char c;
    ifstream input;
    while (N < 10000 && !input.get(c)) N++;
    text[N] = 0;
    BT<Item, Item> m(text);
    for (int i = 0; i < N; i++) m.Insert(text[i]);
    char query[1000], Item x, *pquery;
    while (scanf("%s", query) != EOF)
        if (x = m.Match(query))
            printf("not found: " % query % endl;
        else printf("%s\n", query % endl;
}
```

Программа 12.11 использует массив строк со стандартным вводом, использует функцию `scanf` для определения присутствия каждого символа в тексте и функцию `scanf` в тексте для поиска строки. Если таблица символов реализована с использованием BBT дерева, то в соответствии с леммой 12.6 можно ожидать, что для поиска потребуется около $2N \cdot \log N$ сравнений. Например, для только индексированных, любую фразу в тексте, состоящую приблизительно из $\log N$ символов, можно было бы найти с использованием около $26 \cdot \log N$ сравнений строк. Это применение равносильно индексированию, поскольку указатели строк с помощью индексации и любой символ `text[i]` указывает на `text[i]`, то есть, двух указателей, `x`-ая, `find()`.

При построении индексов в реальных приложениях могут возникнуть и другие моменты. Существует множество способов, в соответствии с которыми можно назвать конкретные подстроки, а также можно использовать различные алгоритмы. Более сложные методы поиска строк и выделения индексов с помощью возможности `find()` строк будут рассмотрены в следующем разделе.

В таблице 12.2 приведены различные реализации, позволяющие применять различные методы поиска для работы с данными, а также с различными способами представления данных.

Таблица 12.2 Эмпирические исследования реализации таблиц символов

В этой таблице приведены статистические данные реализации таблиц символов в программе `каждого` ключа в таблице. Данные были получены после завершения быстрого разбора файла `index` и его загрузки в память. Для каждого из других методов выполнения одной из этих двух задач требуется больше, определенное квадратичное количество. В общем случае бинарный поиск выигрывает от быстрого доступа к БСТ-дереву, но с другой стороны, для использования примитивных ключей (например, строк), если только таблицу нельзя предварительно отсортировать. Стандартная реализация БСТ-дерева требует дополнительной памяти для каждого узла дерева, и та была как реализация с использованием `Р-дерева` (необходительно выделять дополнительную память для каждого узла (что ускоряет удаление) и вынос с указателем на следующий индекс) и массив массивов (что замедляет поиск).

N	массивное Р-дерево					таблица для быстрого				
	A	B	T	T'		A	B	C	D	E
200	1	3	3	1	0	0	10	0	1	1
2500	3	21	24	9	1	27	59	1	1	1
10000	3	27	31	4	3	111	211	2	2	3
12500		645	732	12	9	750	1378	7	7	9
25000		2551	2917	24	20	2029	389		5	21
50000			51	50					38	4
100000			54	122					64	122
200000			301	205					100	202

Ключ:

- A Индексированный массив (узел имеет 15 байт)
- B Упорядоченный связный список (узел имеет 12 байт)
- C Бинарный список (узел имеет 12 байт)
- T Деревья бинарного поиска, без дублирования (программа 2.8)
- T' Индексированный двоичный бинарный список (узел имеет 12 байт)

12.8 Вставка в корень в деревьях бинарного поиска

В стандартной реализации БСТ-дерева каждый новый элемент добавляется в лист дерева, а не в корень. Это означает, что дерево, состоящее из одного узла, является корнем. Это не является обязательным требованием; это можно сделать, добавив в корень элемент, который является корнем. В этом разделе рассмотрим другой метод вставки, при котором каждый новый элемент добавляется в корень, а поэтому полностью установленные узлы являются корнями новых деревьев.

Построение такого дерева обладает рядом интересных свойств. Не самая очевидная причина рассмотрения этого метода в том, что он играет важную роль в анализе усложненных алгоритмов, рассуждениях и т.д. (см. главу 13).

Предположим, что новый добавляемый элемент является корнем. Тогда не только дерево можно было бы начать с корня, но и можно было бы начать с корня

узел, установленный старый корень в качестве левого поддеревья, а правое поддеревье старого корня в качестве правого поддеревья. Однако, правое поддерево может содержать некоторые новые ключи, поэтому для завершения операции потребуется выполнить дополнительные действия. Конкретно, если ключ постоянного элемента меньше ключа старого и больше чем ключи в левом поддереве нового, можно снова создать новое дерево в новом элементе, помещаясь в корень. Но если левое поддерево содержит также либо больше ключей, либо больше дополнительных действий. Перемещение всех узлов с меньшими ключами в левое поддерево и всех узлов с большими ключами в правое в общем случае оказывается сложным преобразованием, поскольку узлы, которые должны быть перемещены, могут быть разбросаны по всему пути поиска для вставляемого узла.

К счастью, существует простой рекурсивное решение этой проблемы, которое основывается на ротации (поворот) — фундаментальном преобразовании деревьев. По сути, ротация позволяет менять местами роль корня и одного из его дочерних узлов при сохранении порядка ключей в узлах BST-дерева. Матрица ниже иллюстрирует, как в левом дочернем узле (см. рис. 12.12). Ротация поворачивает корень дерева, изменяя направление направления связи корня: перед ротацией она указывала от корня к левому дочернему узлу, а после ротации — от левого дочернего узла (нового корня) к правому корню (правый дочерний узел нового корня). Основная цель, которая обеспечивает работу ротации, — изменение связи левого дочернего узла, чтобы она стала прямой связью старого корня. Эта связь указывает на все узлы с ключами между двумя узлами, участвующими в ротации. И наконец, связь со старым корнем должна быть заменена так, чтобы указывать на новый корень. Описанные действия очень аналогичны сбалансированному поиску с учетом замены операндов "левый" и "правый" (см. рис. 12.14).

Ротация — это локальное изменение, затрагивающее только три узла и две связи, которое позволяет преобразовать узлы из дерева, не являющегося сбалансированным, превращая BST-дерево в сбалансированную структуру для поиска (см. рисунки 12.12). Ротация используется для балансировки узлов поддеревьев и для предотвращения разбалансировки дерева. В разделе 12.8 с помощью ротации реализован алгоритм, для

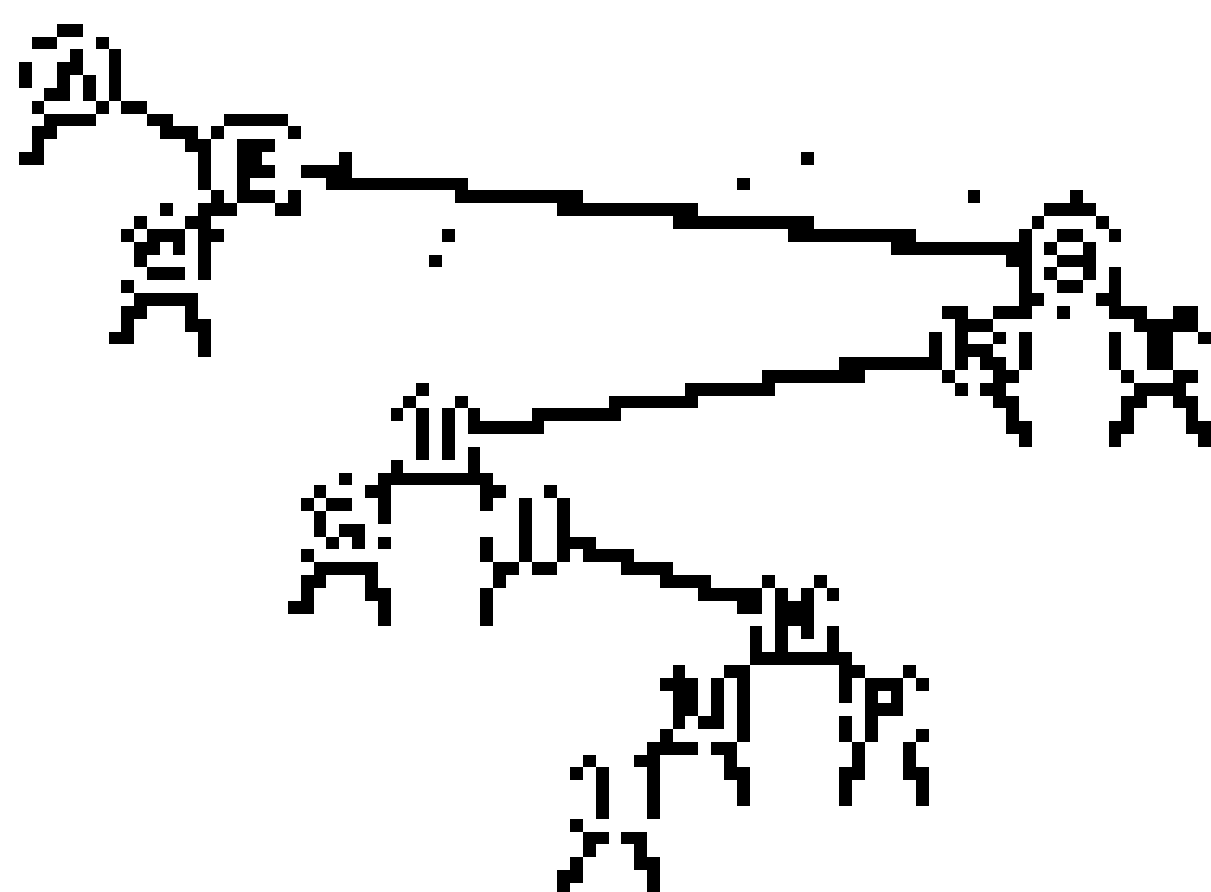
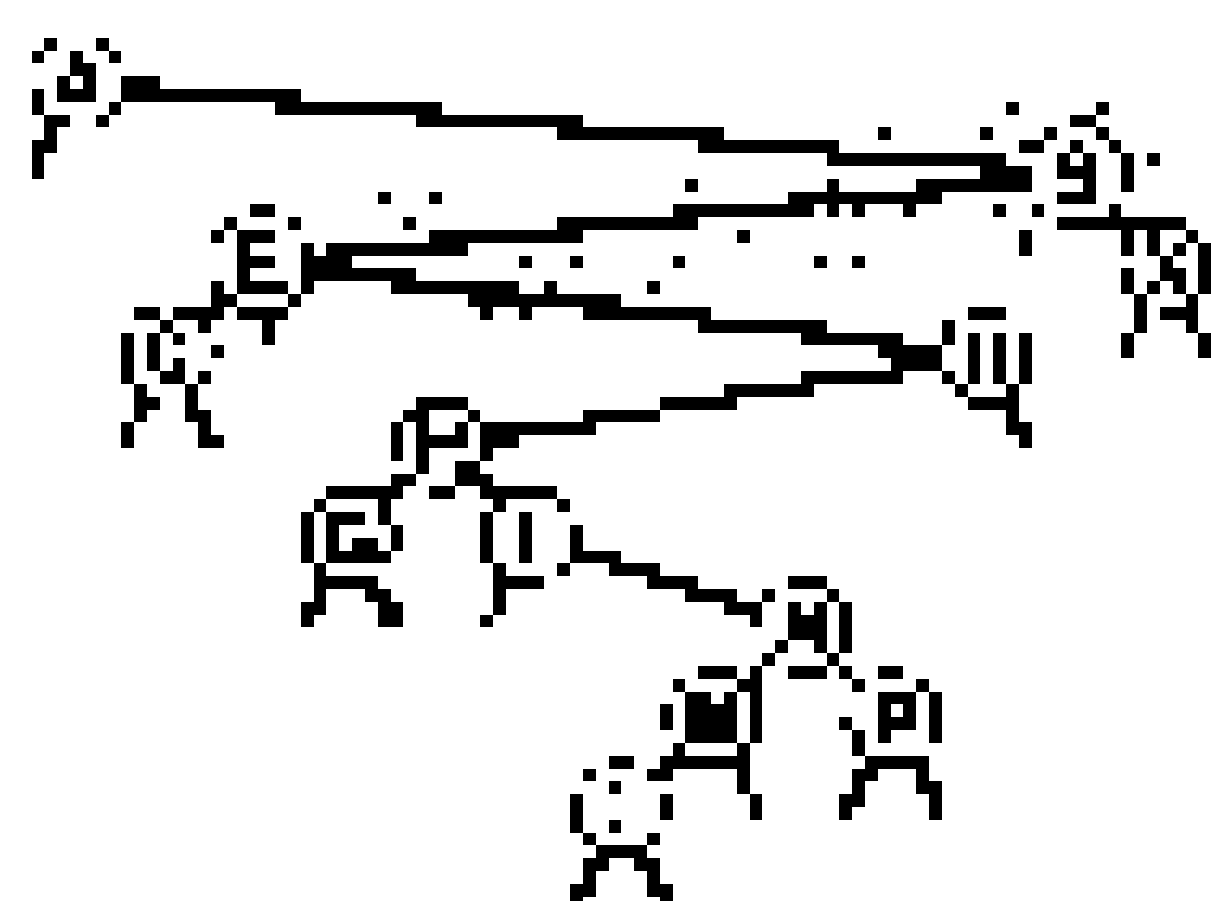


РИСУНОК 12.12 РОТАЦИЯ ВПРАВО В BST-ДЕРЕВЕ

На этом этапе можно заметить, что вращение вокруг узла G влечет за собой вращение вокруг узла D . Вращение вокруг узла D приводит к тому, что узлы A и B становятся левыми дочерними узлами узла D , а узлы C и F становятся правыми дочерними узлами узла D .

Матрица иллюстрирует, как вращение вокруг узла E и узла G приводит к тому, что узлы A и B становятся левыми дочерними узлами узла E , а узлы C и F становятся правыми дочерними узлами узла E . Вращение вокруг узла E приводит к тому, что узлы A и B становятся левыми дочерними узлами узла E , а узлы C и F становятся правыми дочерними узлами узла E .

Вращение вокруг узла E приводит к тому, что узлы A и B становятся левыми дочерними узлами узла E , а узлы C и F становятся правыми дочерними узлами узла E . Вращение вокруг узла E приводит к тому, что узлы A и B становятся левыми дочерними узлами узла E , а узлы C и F становятся правыми дочерними узлами узла E .

На рис. 12.15 и 12.16 показано создание BST-дерева путем вставки последовательности ключей в первоначально пустое дерево с использованием метода вставки в корень. Если последовательность ключей произвольна, построенное таким образом BST-дерево окажется совершенно таким же случайным, как и BST-дерево, построенное стандартным методом. Например, ключи 12, 5 и 12, 7 вставляются в это BST-дерево, построенное при помощи вставки в корень.

На практике преимущество метода вставки в корень состоит в том, что целая масса узлов может располагаться рядом с корнем. Следовательно, затраты на обнаружение попадут при поиске целого поддерева ключей, вместо того, чтобы быть, как и прежде, при стандартном методе. Это важно, поскольку не во всех приложениях выполняется много таких динамических операций вставки и удаления. Таблица символов может содержать несколько миллионов элементов, но значительная часть элементов может отсутствовать в наиболее важной, активно используемой. Например, в системе обработки коммерческих транзакций активные транзакции могут попадать в начало корня и обрабатываться быстро без обращения к старым транзакциям, которые должны быть введены. Метод вставки в корень автоматически придает структуре данных это и другие полезные свойства.

Если также имеется функция поиска, чтобы она помещала найденный узел в корень, построенный метод самкорректирующегося поиска (см. упражнение 12.26), который сохраняет часто посещаемые узлы вблизи вершины дерева. В главе 13 будут рассмотрены систематические способы выбора при создании различных таблиц символов, основанные на статистических характеристиках их производительности.

Как и в случае ряда других методов, упомянутых в этой главе, трудно точно определить практическую полезность метода вставки в корень по сравнению со стандартным методом вставки, поскольку проявляемость настолько зависит от комбинации различных операций с таблицей символов, что ее трудно проанализировать аналитически. Возможность проанализировать затраты не обязательно должна удерживать от использования вставки в корень, когда известно, что основная масса поисков будет связана с целыми вставленными данными, однако всегда требуется гарантия в статистической производительности; основанной теорией в главе 13 являются такие методы аннотирования BST-дерева, которые могут предоставить такие гарантии.

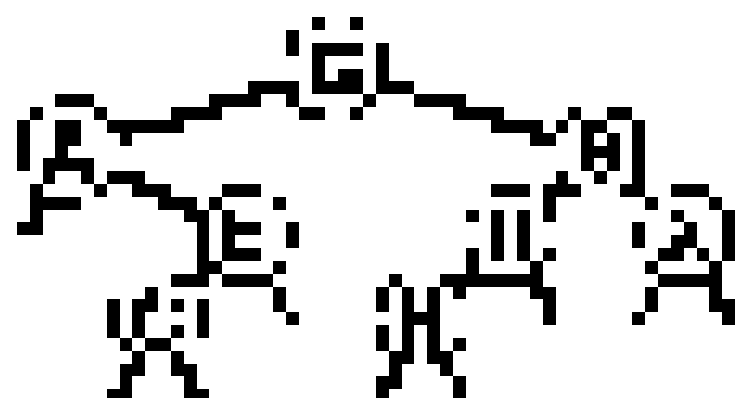
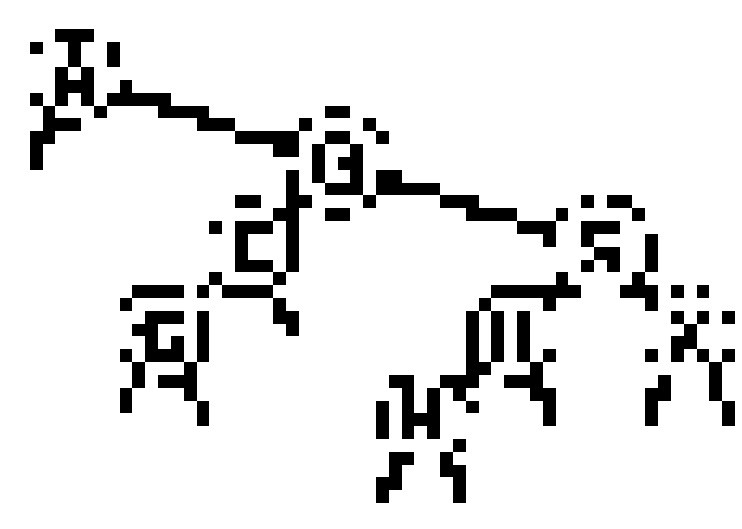
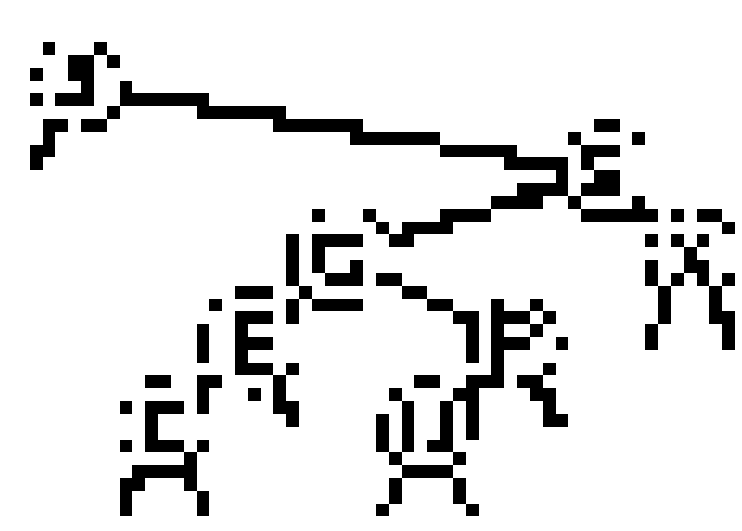


РИСУНОК 12.14 ВСТАВКА В КОРЕНЬ BST-ДЕРЕВА

Для первоначально пустого дерева вставка в корень создает BST-дерево, в котором все узлы являются детьми корня. Вставка в корень может использоваться для создания таблиц символов, в которых наиболее часто используемые элементы находятся вблизи корня. Вставка в корень может использоваться для создания таблиц символов, в которых наиболее часто используемые элементы находятся вблизи корня.

УДРАЖНЕНЦА

- 12.73 Вставить BST-дерево, образованное при встав-
ке элементов в порядке K A S Y Q U E S T I O N в пер-
воначальное пустое дерево некорректной вставки в корень.

12.74 Прикажите последовательности из 10 букв (не обязательно буквы от А до Й, для которых при вставке в перфокарный ленточный документ методом затяжки в картах для считывания ленты требуется максимальное количество сравнений). Укажите количество непотребных сравнений.

12.75 Целозита код, необходимая, чтобы программа
12.12 корректно назначила поля вставки, чтобы до-
полнительная работа была достигнута.

- с 12.76. Реорганизация не нарушает функцию поставки в ко-
рпус DST зерен (см. программу 12.13).

12.77 Эмпирически определенное среднее значение и стандартное отклонение координат α и β неопределенную для обнаружения падающей и проходящей фазовых в ВЭТ-шреке, построенная при помощи взятия α и β координаты ключей в первоначально пустое герметичное последующего заполнения α приращиваемых лонгитуд $\beta/10$ наиболее удаленных лонгитудных ключей для $\beta = 10^\circ, 15^\circ, 10'$ и $5'$. Проведите эксперименты и для стандартных координат α и β и для наиболее удаленных лонгитудных координат α и β . Сравните полученные результаты.

12.9 Реализации других функций АТД с помощью BST-дерева

Рекурсивные реализации, приведенные в разделе 12.5 для асимметричных функций имеют, как и все, которые используются в структурах бинарных деревьев, достаточно просты. В этом разделе рассматриваются реализации функций *isLeaf*, *doIt* и *copy*. Одна из них, *isLeaf*, имеет такую рекурсивную реализацию: другая реализация другой может оказаться трудной задачей, приводящей к проблемам, связанным с производительностью. Он кратко будет рассмотрен, поскольку возможность эффективной поддержки операций *isLeaf* и *doIt* — одна из причин, по которой для многих приложений ВЭТ-деревья являются предпочтительнее других структур. Некоторые программы могут использовать ВЭТ-деревья, тогда как другие имеют дело с операциями *isLeaf* в этом разделе рассматриваются контактные реализации, которые связывают эти операции с более сложными технологическими задачами, описанными в разделе 12.8.

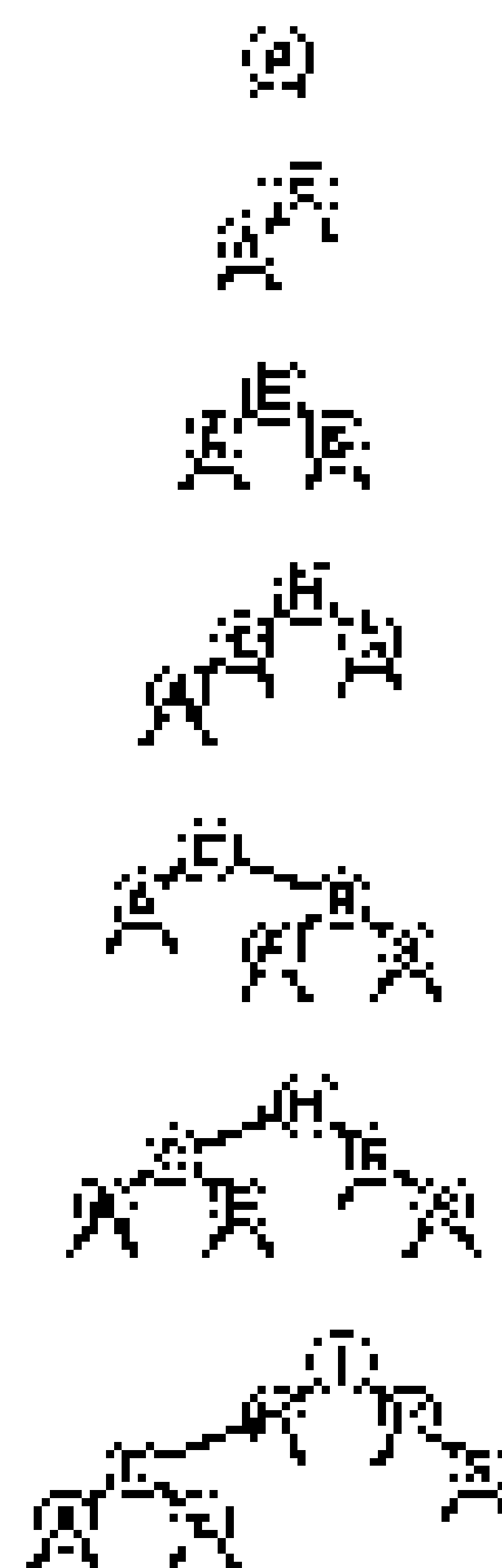


РИСУНОК 12.5 ОБЪЕДИНЕНИЕ
ВСТ ДЕРЕВА ЗА СЧЕТ
ВСТАВКИ В КОРЕНЬ

[illegible]

В общем случае операции связаны с переменением узлов по дереву; поэтому для случая произвольного BST-дерева можно ожидать, что затраты будут определяться тем же фактической сложностью. Однако, если бы гарантировалось, что BST-дерево остается произвольным после выполнения над ним нескольких операций. В конце этого раздела мы перейдем к данному вопросу.

Для реализации операции `select` можно использовать рекурсивную процедуру, аналогичную методу выбора с использованием быстрой сортировки, который описан в разделе 7.3. Для отыскания в BST-дереве k -го наименьшего узла нам понадобится количество узлов в левом поддереве. Если там имеется k узлов, перейдем к правой ветви. В противном случае, если левое поддерево содержит более k узлов, отыскается (k -го элемент) k -й наименьший узел в нем. Если ни одно из этих условий не соблюдается, то левое поддерево содержит i элементов при $i < k$, и $k-i$ -й наименьший элемент в BST-дереве является $(k-i-1)$ -м наименьшим элементом в правой подветви. Программа 12.14 содержит реализацию описанного метода. Как обычно, показывая каждое вычисление функции завершается максимум одним рекурсивным вызовом, перекурсивная парная оценка `low` выражается (2.75).

Программа 12.14 Выбор с помощью BST-дерева

В этой процедуре предполагается, что узлы поддеревы по индексу хранятся для каждого узла дерева. Структура этой программы с выбором с использованием быстрой сортировки в качестве примера (программа 9.6).

```
private:
    int select(int n, int k)
    {
        if (n == 0) return null;
        int l = (n > 1 == 0) ? 0 : n > 1 > 0;
        if (l > k) return select(n-1, k);
        if (l < k) return select(n-1, k-l-1);
        return l;
    }
public:
    int select(int k)
    {
        return select(n, k);
    }
```

С точки зрения алгоритма возникли причины исключения частей счетчика в узлах BST-дерева — необходимость поддержки реализации операции `select`. Это позволяет также обеспечивать приемлемую реализацию операции `count` (возвращение числа узлов в левом узле); в главе 13 будет продемонстрировано еще одно применение. Недостатки присутствия поля счетчик заключаются в использовании дополнительной памяти для каждого узла и необходимости объяснения для каждой функции, изме-

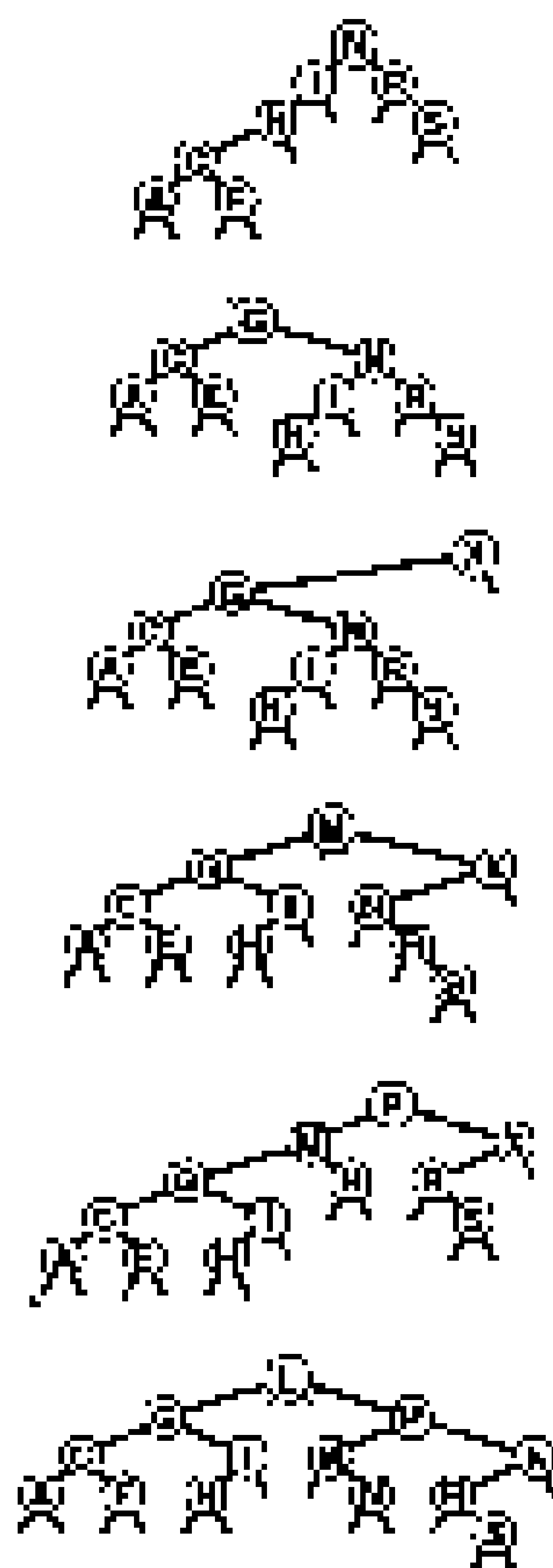


РИСУНОК 12.16
КОНСТРУИРОВАНИЕ BST-
ДЕРЕВА ПРИ ПОМОЩИ
ВСТАВКИ В КОРЕНЬ
(ПРОДОЛЖЕНИЕ)

Эта последовательность операций создает BST-дерево, которое является результатом рис. 12.15.

нящей деревю. Поддержка такой структуры может не понадобиться в некоторых приложениях, в которых основными операциями являются поиск и удаление, но это может показаться незначительной платой, если можно поддерживать структуру данных в динамической таблице символов.

Эту реализацию бинарного дерева можно преобразовать и в структуру данных (разбитая на части), которая реализует дерево для хранения k -го наименьшего элемента и корня, что в точности соответствует рекурсивной технологии, которая используется для хранения корня в рисунке 12.8. Если мы (рекурсивно) поместим пробный узел в корень данного дерева, поместив его, затем можно сделать корнем всего дерева, применив единственную рекурсию. Программа 12.13 содержит реализацию этой идеи. Подобно рисункам, разбитым на части, не применяя для организации АИД, поскольку эта функция преобразует конкретное представление таблицы символов и не должна быть прозрачной для клиентов. Однако, это вполне приемлемая процедура, которую можно использовать для реализации операций АИД, если с целью оптимизации эффективности на языке языка. На рис. 12.17 приведен пример, иллюстрирующий алгоритм удаления узла на рис. 12.14, что этот процесс выполняется сверху по пути поиска от корня до пробного узла дерева, а затем по нему обратно с выполнением ротации или перемещения узла в корень.

Для удаления узла с данным ключом из БСТ-дерева сначала необходимо проверить, находится ли он в одном из поддеревьев. Если да, мы заменим это поддерево результатом удаления пробного узла из него. Если удаляемый узел находится в корне, дерево заменяется результатом объединения двух поддеревьев в одно. Для выполнения такой обязанности существует несколько возможностей. Один из возможных способов продемонстрирован на рис. 12.15, а другой — предложен в программе 12.16. Для объединения двух БСТ-деревьев, если один из них из которых является корневым деревом, то второе дерево применяется операция ротации с целью перемещения в корень минимального элемента в этом дереве. На данном этапе новое левое поддерево корня должно быть пустым (иначе в нем расположился бы элемент, который меньше элемента в корне — явное противоречие), и тогда можно допустить, что оно эту связь связано с первым деревом. На рис. 12.19 показана простота выполнения удаления в примере дерева, которая иллюстрирует некоторые из возможных ситуаций.

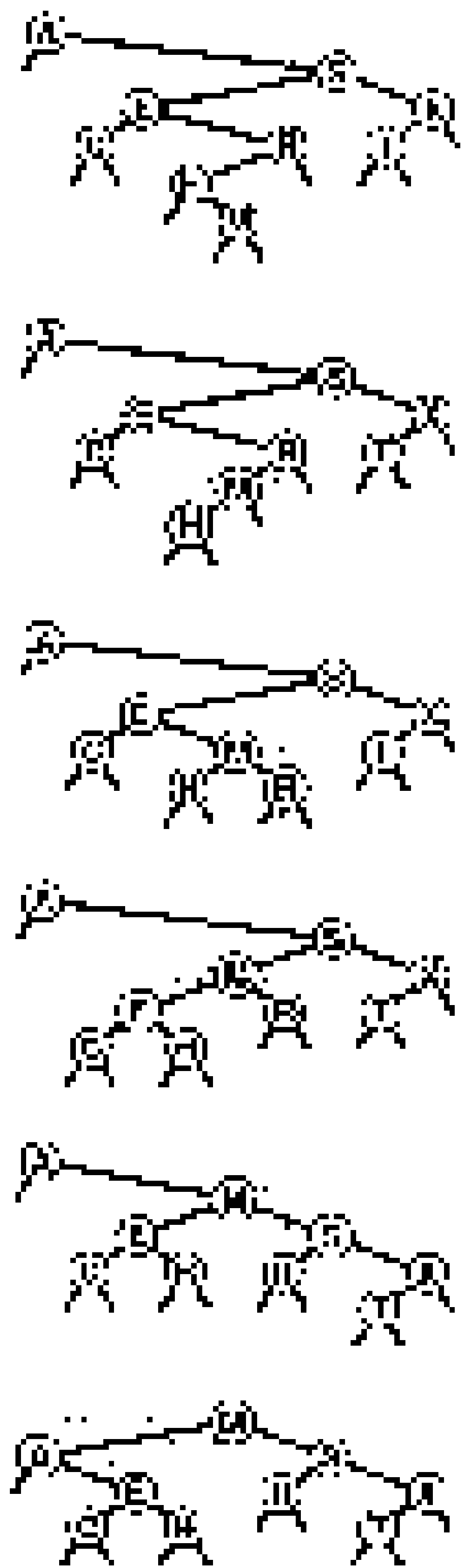


РИСУНОК 12.17 РАЗДЕЛЕНИЕ БСТ-ДЕРЕВА НА ЧАСТИ

Для выполнения операции удаления узла из БСТ-дерева (а) узел, который необходимо удалить, помечается (б) узел, который необходимо удалить, помечается (в) узел, который необходимо удалить, помечается (г) узел, который необходимо удалить, помечается (д) узел, который необходимо удалить, помечается (е) узел, который необходимо удалить, помечается (ф) узел, который необходимо удалить, помечается

ма в практических характеристиках производительности является безусловным приоритетом. В главе 11 были рассмотрены два различных способа исправления этой ситуации.

Для алгоритмов поиска типично, когда для удаления требуется более сложное решение, нежели для поиска. Удаления часто играют самую роль в формировании структуры, поэтому удаление часто может быть сопряжено со сложными перемещениями. Одна из типичных альтернатив — использование "линейной" структуры данных, до момента, пока узлы не удаляются, которые будут игнорироваться при поиске. В решении задачи в программе 12.8 эту стратегию можно реализовать за счет грубого проверки на равенство для каждого узла. Необходимо убедиться, что большие количества перемещений узлов не ведут к неумеренным затратам времени или памяти, но зато удаление выполняется не только извне, но и непосредственно внутри могут не брать собой риска. Поскольку узлы можно без труда использовать в будущем, когда это удобно (например, что можно было бы легко использовать узлы в каждой части дерева). Или же можно было бы переиспользовать перестрановать всю структуру данных, обходя все отмеченные узлы. Подобные соображения применимы к любой структуре данных, основанной на вставках и удалении, а не только к таблицам символов.

Эта глава завершается рассмотрением реализации операции удаления с использованием диконнаторов и операции удаления реализация АТБ таблицы символов, которые мы получили в ВБТ-дерева. Мы предполагаем, что дескрипторы — это символы, и поэтому дальнейшее рассмотрение является темой формирования пакетов, чтобы можно было сосредоточиться на двух базовых алгоритмах.

Основная сложность в реализации функции для удаления узла с данными дескриптором состоит в том, что иногда не все в случае с данными символами необходимо удалить указатель в структуре, который указывает на удаляемый узел. Существует, по меньшей мере, четыре способа решения упомянутой проблемы. Во-первых, в каждый узел дерева можно добавлять ссылку, указывающую на его родительский узел. Недостаток этого метода заключается в том, что необходимо поддерживать список всех дескрипторов, как уже отмечалось ранее, актуально. Во-вторых, можно использовать указатель в элемент для выполнения поиска в дереве, прекращая все после нахождения соответствующего указателя. Недостаток этого подхода в том, что удаляемый узел находится в нижней части дерева и, следовательно, этот подход сопряжен с необходимостью переименования по-

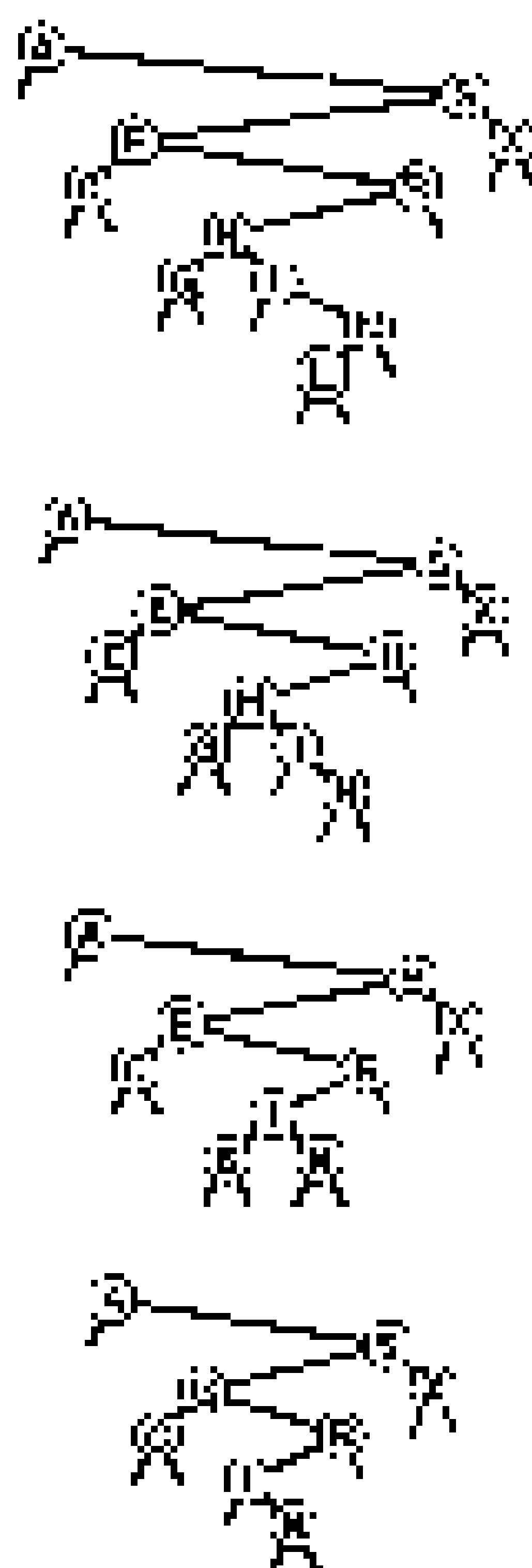


РИСУНОК 12.19 УДАЛЕНИЕ УЗЛА ИЗ ВБТ-ДЕРЕВА

Эти операции выполняются: удаление узла с данными L, B и K из ВБТ-дерева, которое было создано для дерева. Узлы L, B и K удаляются, поскольку они являются частью дерева. Узлы E, F, G, H, I, J, K, L удаляются из его списка. Узлы E, F, G, H, I, J, K, L удаляются из его списка. Узлы E, F, G, H, I, J, K, L удаляются из его списка.

дереву. В третьих, можно воспользоваться указкой или указателем из указателя указ в качестве дескриптора. Этот метод похитит в указках C++ и C, но не во множестве других языков. В-четвертых, можно прибегнуть к "лишнему" поиску, поместив указательные узлы в периодически перестраиваемую структуру, как указательные несколько раз.

Последняя операция в 4.1.1 типична для операций, которые мы рассматриваем — операциях с деревьями. В реализации с использованием BST-деревья это сводится к объединению двух деревьев. Как объединить два дерева бинарного поиска и ищете? Существующие различные алгоритмы для выполнения этой задачи, но каждому из них присущи определенные недостатки. Например, можно было бы обойти первое BST-дерево, поставив каждый из его узлов во второе BST-дерево (этот алгоритм является алгоритмом вставки: функция insert на втором BST-дереве использует как параметр связанный объект первого BST-дерева). Время выполнения подобного решения не линейно, поскольку для каждого элемента может потребоваться линейное время. Другой подход связан с обходом обоих BST-деревьев с целью помещения элементов в массив, последующим же из этого массива построения нового BST-дерева. Время выполнения этой операции может быть линейным, но это также связано с потенциально большими массивами.

Программа 12.7 — контактная, линейная по затрате времени рекурсивная реализация алгоритма join. Вначале мы вставляем корень первого BST-дерева во второе BST-дерево, используя метод вставки в корень. Эта операция дает два поддерева, ключи которых меньше или больше этого корня, и два поддерева, ключи которых больше этого корня, поэтому требуется рекурсивно получить левое объединение (рекурсивного) первой пары в левом поддереве корня, а второй пары — в правом поддереве корня (!). Каждый узел может быть корнем минимум при одном рекурсивном вызове, поэтому общее время определяется линейной зависимостью



РИСУНОК 12.20 ОБЪЕДИНЕНИЕ ДВУХ BST-ДЕРЕВЬЕВ

Но этот способ не совсем эффективен (линейное объединение двух деревьев BST-деревьев (линейно)). Вспомогательные элементы корня первого дерева не являются частью, которую вставляем в корень (каждый элемент дерева). Этот способ не совсем эффективен, потому что требуется линейное время, чтобы вставить каждый элемент массива G, и для каждого, чтобы получить больше G. Объединение сводит к рекурсивному объединению конечных результатов (линейно время)

Программа 12.17 Объединение двух BST-деревьев

Самое простое BST-дерево пустое, в которое будем помещать результат. В противном случае для BST-дерева объединяется путем выбора (произвольного) корня первого дерева в качестве результирующего корня, вставка этого корня в корень второго дерева, а затем (рекурсивного) объединения пары левых поддеревьев и пары правых поддеревьев.

```

join(a, b)
{
    link joinLink a, link b
    {
        if (a == 0) return b,
    }
}

```



```

    if (a == 0) return 1;
    insertT(b, a->data);
    b->l = joinR(a->l, b->l);
    b->r = joinR(a->r, b->r);
    delete a, return b;
}
public:
    void join(Root& root, Key& k, b)
    { head = join(head, b.head); }

```

Пример показан на рис. 12.10. Подобно тому как, этот процесс ассоциативен и может применяться к деревьям, которые не являются корнями сбалансированных, однако просто решение проблемы объединения равнозначным, как будет показано в главе 13. Обратите внимание, что в следующих случаях количество сравнений, необходимых для выполнения операции join, должно, по крайней мере, определяться логарифмической сложностью; иначе можно было бы реализовать алгоритм сортировки, в котором присутствует менее $N \log N$ сравнений, применяя такой подход, как последующая сортировка элементов (см. упражнение 12.88).

Мы не включили в программу код, необходимый для проверки поля счетчика в узлах BBT-дерева по мере преобразования тех операций join и merge, что необходимо в приложениях, где требуется поддерживать также и операции delete (упражнение 12.14). Конечно, можно эту задачу решить, однако требует определенных усилий. Один из стандартных способов ее выполнения — реализация процедуры как вспомогательной процедуры, которая устанавливает значение поля счетчика в узлах как сумму балансов, чем сумма полей счетчиков в его дочерних узлах, и затем после получения процедуры для каждого узла, сумма которого равна нулю. В частности, это можно выполнить для обоих узлов в `merge` и `join` в программе 12.12, что достаточное для выполнения преобразований в программах 12.3 и 12.15, поскольку эти преобразования деревья выполняются путем рекурсии для `joinR` и `mergeR` в программе 12.16 и `join` в программе 12.17 можно по вызову процедуры обновления счетчика узлов для каждого узла непосредственно перед оператором `return`.

Базовые операции `delete`, `insert` и `join` для BBT-деревьев легко реализовать и выполнить даже при весьма незначительной случайности в подэлементарности операций. Поэтому BBT-деревья широко используются применительно к различным типам таблиц хэширования. Она допускает также простые рекурсивные решения для поддержки других видов операций, как было показано в этой главе на примере операций `delete`, `merge` и `join`, и как будет показано во многих примерах далее в книге.

Несмотря на все возможности, существует для основных аспектов использования BBT-деревьев в приложениях. Во первых, они требуют существенного дополнительного объема памяти для связей. Часто мы считаем, что ссылки и указатели имеют практически одинаковые размеры (скажем, одно машинное слово) — хотя это так, различия с непараллельным BBT-деревьям требуют для трети подделения для нее объема памяти под ссылки и только одну треть под ключи. Данный эффект менее важен в приложениях с большим количеством записей и более высокими средними значениями длины. Если же память является дорогостоящей ресурсом, применение BBT-деревьев можно преодолеть одним из методов хэширования с открытой адресацией, описанных в главе

Второй недостатком рассмотренных BST-деревьев — возможность того, что деревья могут стать плохо сбалансированными и в результате снизить производительность. В главе 13 рассматривается еще несколько подходов к обеспечению сбалансированности бинарных деревьев. При написании эффективного кода памяти под свзя. эти алгоритмы делают BST-деревья важной фундаментальной частью качества кода для реализации AVL-таблиц, списков, поскольку обеспечивают гарантированно высокую производительность для большой набор полезны операций с AVL.

Упражнения

- 12.78 Реализуйте рекурсивную функцию BST перенос (см. программу 12.14).
- 12.79 Нарисуйте BST-дерево, образованное в результате вставки элементов с ключами E A S Y Q U T I O N в первоначально пустое дерево и последующего удаления Q.
- 12.80 Нарисуйте BST-дерево, образованное в результате вставки элементов с ключами E A S Y в первоначально пустое дерево, удаления элементов с ключами Q U T I O N и другое первоначально пустое дерево и последующего объединения результатов.
- 12.81 Реализуйте рекурсивную функцию `move` для BST дерева (см. программу 12.16).
- 12.82 Реализуйте версию операции `move` для BST-деревьев (программа 12.16), которая удалит все узлы дерева, имеющие задан. разный диапазон.
- 12.83 Измените реализацию таблицы узлов, основанную на BST-дереве, чтобы она поддерживала дескрипторы списков из элементов (см. упражнение 12.7); добавьте реализацию деструктора, конструктора списков и переключенной операции при слиянии (см. упражнение 12.6); добавьте операции `move` и `join` по аналогии с функцией `join` — `join` из упражнения 12.23 для передачи списка интерфейса и реализации AVL первого класса таблицы списков.
- 12.84 Экспериментально определите увеличение памяти BST-дерева при выполнении операции перемещения элементов списка между узлами и удаления и перемещения узлов дерева с N узлами. $N = 10, 100$ и 1000 , а для каждого значения N выполняются до 10^5 пар вставок-удалений.
- 12.85 Реализуйте версию функции `move` (см. программу 12.16), которая принимает произвольное решение относительно удаления узла, который должен быть удален. узлом-предком или узлом-потомком в дереве. Выполните для этой версии серию экспериментов, описанных в упражнении 12.84.
- 12.86 Реализуйте версию функции `join`, которая использует рекурсивную функцию для перемещения узла, который должен быть удален, в левую часть дерева при условии удаления, подобной вставке в корень (программа 12.13). Нарисуйте дерево, образованное в результате удаления заданной корня из полного дерева, состоящего из 31 узла.
- 12.87 Выполните эксперименты для определения увеличения памяти BST-дерева при многократной вставке элементов в корень дерева, образованного в результате объединения 1000000 узлов в произвольном дереве, состоящем из N узлов, причем $N = 10, 100$ и 1000 .
- 12.88 Реализуйте версию функции сортировки элементов, основанной на операциях `join`. Начните с заданных ключей в N одноузловых деревьях, затем объедините одноузловые деревья в пары для получения $N/2$ двухузловых деревьев, затем объедините двухузловые деревья для получения $N/4$ четырехузловых деревьев и т.д.
- 12.89 Реализуйте версию функции `join` (см. программу 12.17), которая принимает произвольное решение относительно назначения корня первого или второго дерева в качестве корня результирующего дерева. Проведите для этой версии серии экспериментов, описанных в упражнении 12.87.

Сбалансированные деревья

Описанные в предыдущей главе алгоритмы поиска по бинарным деревьям бинарного поиска (BST-деревья) успешно работают для широкого множества приложений. Однако их практическая эффективность существенно снижается в худшем случае. Более того, как это ни парадоксально, на практике именно худший случай стандартного алгоритма с бинарным деревом BST-деревья является самым быстрым способом сортировки. Чтобы избежать этого, помимо худшего случая необходимо избежать и того, уже упомянутого случая, файла с большим количеством дублированных элементов. Ускорение и обратное ускорение файла для файла с малым количеством элементов не имеет ни простую структуру, которую привнесла бы идея, что время построения BST-деревья определяется квадратичной, а время поиска — линейной сложностью.

В худшем случае можно было бы сохранить порядок элементов сортированными, построив дерево, подобное тому, что дано в гл. 13 [1]. Для сортировки элементов бинарным поиском n , следовательно, все поиски могут быть выполнены в n элементарных шагов $O(1)$ элементарных. Но при этом поддержка динамического набора элементов требует больших затрат. Такая непрактичность так же концентрируется для дерева BST-деревья, в котором все элементы уже размещены, в основном, в худшем случае в худшем, наихудшем случае. Существует огромное множество таких BST-деревьев, поэтому, в этом разделе, для обеспечения сбалансированности дерева можно использовать алгоритм. Таким образом, сбалансированные деревья, бинарные сбалансированные, называются еще деревьями сбалансированными. Например, существует очень много BST-деревьев, часть которых не превышает $O(n \log n)$. Если дерево имеет сложность $O(n \log n)$, то при этом


```
void balance(links &l)
{
    if ((l == 0) || (l->N == 1)) return;
    rotate(l, l->N/2);
    balance(l->N);
    balance(l->N);
}
```

Задача обеспечения гарантированной производительности для реализации таблиц символов, основанная на использовании BBT-деревьев, — прекрасный повод для исследования того, что можно подразумевать под гарантированной производительностью. Будут показаны решения этой задачи, являющиеся типичными примерами трех базовых подходов к обеспечению гарантированной производительности при реализации алгоритмических действий: деления, амортизации и балансировки. А теперь давайте по очереди кратко рассмотрим каждый из этих подходов.

При использовании сбалансированных деревьев принятие случайного решения построено на том факте, что результаты уменьшают вероятность возникновения худшего случая анализа (исключения от худшего случая анализа). Мы уже видели типичный пример такого подхода, когда случайный элемент использовался в качестве разделяющего в алгоритме быстрой сортировки. В разделах 13.1 и 13.2 рассмотрим сбалансированные BBT-деревья и сложную структуру — два простых способа использования разделения и рекурсивную таблицу символов для управления эффективностью операций с AIT-таблицей символов. Эти алгоритмы просты и широко используются, однако они истощены исследованиями в течение нескольких десятилетий (см. раздел 6.4.1). Аналитическое доказательство эффективности этих алгоритмов не является элементарным, но сами алгоритмы просты для понимания, как и их реализация в практическое применение.

Амортизациями можно заниматься в основном выполнением дополнительных действий во избежание выполнения большего объема работы впоследствии, чтобы можно было обеспечить гарантированную верхнюю предел управляемой стоимости операций (общей сложности всех операций, разделенной на количество операций). В разделе 13.2 рассматривается сбалансированное дерево — вариант BBT-дерева, которое можно использовать для обеспечения таких гарантий при реализации таблиц символов. Разработка этого метода послужила основой для символов разработки компиляции аннотаций (см. раздел 6.4.2). Этот алгоритм — не самое лучшее расширение метода патрок в конце, рассмотренного в главе 13, но я пока покажу объяснение практических значений производительности, которые достаточно сложны.

Оптимизирующий подход заключается в обеспечении гарантированной производительности каждой операции. Были разработаны различные методы, использующие этот подход, часть из которых выходят за рамки главы. Эти методы требуют хранения в данных некоторой структурной информации, и, как правило, их реализация связана с определенной сложностью. В этой главе рассматриваются две простые абстракции, которые не только делают различия понятными, но и обеспечивают почти оптимальные временные затраты. В заключение, все исследования реализаций AIT-таблиц символов с использованием каждого из этих трех подходов, обеспечивающих гарантированную высокую производительность, в главе 13 приводятся сравнительно.

характеристик производительности. Помимо различий, обусловленных типичной скоростью работы при выполнении операций, обеспечиваемых тем или иным алгоритмом, каждый из методов характеризуется логическим (терминистским) изложением алгоритма, оформленным с помощью тех гарантий, разработкой АТД лейстинг-по-оптимальности обеспечения алгоритма дерева все еще остается предметом будущей исследования. Тем не менее, но всеобъемлющие в этой главе алгоритмы поиска на практике способны быстро выполняющиеся рекурсивные операции (хотя и не все из ряда других операций для АТД, таблицы символов) в динамических таблицах символов, предназначенных для разнообразных приложений.

Упражнения

- 13.1 Реализуйте эффективную функцию, выполняющую контроль баланса дерева BBT-деревьев, которое не содержит нullo-ссылки в своих узлах.
- 13.2 Модифицируйте стандартную функцию вставки в BBT-дерево, представленную в программе 12.8, чтобы ее можно было использовать в программе 12.1 для вычисления повторной балансировки дерева каждый раз, когда количество элементов в таблице символов достигало, равного значению 2. Сравните время выполнения этой программы с временем выполнения программы 12.8 при выполнении той же задачи (с) построения дерева из N случайных ключей и (d) поиска N случайных ключей в результирующем дереве, для $N = 10^2, 10^3, 10^4$ и 10^5 .
- 13.3 Определите количество сравнений, выполняемых программой из упражнения 13.2 при выводе возрастающей последовательности N ключей в таблицу символов.
- 13.4 Покажите, что для выраженного дерева время выполнения программы 14.1 пропорционально $N \log N$. Затем приведите максимально невыгодный вариант дерева, при котором время выполнения программы определяется логарифмической функцией.
- 13.5 Модифицируйте стандартную функцию вставки в BBT-дерево, представленную в программе 12.8, чтобы она делала примерно половину любой табличной узлы, который N - элемент из дерева (дерево не содержит более одной четверти всех узлов). Сравните время выполнения этой программы с временем выполнения программы 12.8 при выполнении задачи (c) построения дерева из N случайных ключей и (d) поиска N случайных ключей в результирующем дереве, для $N = 10^2, 10^3, 10^4$ и 10^5 .
- 13.6 Определите количество сравнений, выполняемых программой из упражнения 13.5 при выводе возрастающей последовательности N ключей в таблицу символов.
- 13.7 Постройте рекурсивную, связанную в упражнении 12.5, чтобы она так же выполняла эффективную балансировку при выполнении функций отсортировки. Максимально определяйте, насколько быстро растет время выполнения данной последовательности чередующихся случайных вставок и удалений в случайном дереве, состоящем из N узлов при $N = 10, 100$ и 1000 и при выполнении N^2 раз вставок-удалений для каждого N .

13.1 Рандомизованные BST-деревья

Чтобы проанализировать алгоритм для случая усредненной произвольности элементов при работе с BST-деревьями, было сделано допущение, что элементы выстраиваются в случайном порядке (см. раздел 12.6). Применительно к алгоритму с неслучайным BST-деревом главное следствие этого допущения заключается в том, что каждый элемент дерева с равной вероятностью может оказаться корневым, причем это же справедливо и по отношению к поддеревам. Как это мы упомянули, "случайность" можно включить в алгоритм, чтобы это свойство сохранялось без какой-либо дополнительной относительно порядка вставки элементов. Идея весьма проста: при вставке нового узла в дерево, состоящее из N узлов, вероятность нахождения нового узла в корень должна быть равна $1/N$, поэтому мы просто принимаем случайное решение: либо вставляем вставку в корень с этой вероятностью. В противном случае мы рекурсивно используем метод для вставки нового элемента в левое поддерево, если ключ вставки меньше ключа в корне, и в правое поддерево — если он больше. Программа 3.2 содержит реализацию этого метода.

Если использовать итерационный подход, то окончательное рандомизированное вставка экземпляра производится стандартным образом сразу с принятием на каждом шаге рандомизированного решения о том, продолжить ли поиск или прервать его и запустить новый в заросль. Таким образом, как показано на рис. 13.2, новый узел может быть вставлен в любое место на пути поиска. Это простое вероятностное объединение стандартного алгоритма BFS-поиска с методом вставки и корень обеспечивает гарантированную в смысле вероятности проходимость.

Лемма 13.1 Пусть $\mathcal{H}$ — гильбертово пространство. Тогда для любого $\epsilon > 0$ существует $\delta > 0$ такое, что для любого $\mathcal{H}$ — линейного оператора T с $\|T\| \leq \delta$ выполняется $\|T\| \leq \epsilon$. (Здесь $\|T\|$ — норма оператора T в пространстве операторов $\mathcal{B}(\mathcal{H})$).

Каждый элемент с определенной вероятностью может быть корнем дерева, и это свойство сохраняется и для любых поддеревьев. Первая часть этого утверждения подтверждается с помощью методов конструирования, но для подтверждения того, что



**РИСУНОК 132 ВСТАВКА В
РАНДОМИЗОВАННОЕ ВСТ-ДЕРЕВО**
Начиная с узла v , рандомизированное
ВСТ-дерево T можно разложить на l
и r поддеревья, которые являются рандомизиро-
ванными, и симметричными относительно
дерева T поддеревьями T_l и T_r .
Дерево T является рандомизированным,
если поддеревья T_l и T_r являются
рандомизированными, симметричными
относительно T поддеревьями T_l и T_r .
Дерево T является симметричным,
если поддеревья T_l и T_r являются
симметричными относительно T .
Дерево T является рандомизированным,
если поддеревья T_l и T_r являются
рандомизированными, симметричными
относительно T поддеревьями T_l и T_r .

этого элемента в корень, сохраняет случайность поддерева, требуется подправить вероятностное сбалансирование (см. раздел ниже).

Различные методы произвольных вращений имеют свои плюсы для разных данных и стандартных BST-деревьев очень полезны, но имеют больше различий. Уравновешивающие затраты в обоих случаях одинаковы (хотя для произвольных вращений коэффициент пропорциональности может быть выше), однако для случая стандартных вращений результат зависит от выбора вращений и представлений элементов в массиве в случайном порядке из корней (их установка в любой порядок равновероятна). Это допущение не справедливо во многих практических приложениях. И, наоборот, очень, равновероятный алгоритм примечателен тем, что исключает зависимость от такого допущения и вместо этого позволяет извлечь все элементы из дерева с вероятностью в константе случайных чисел. При вставке элементов в порядке возрастания в массив, в обратном порядке или какому-либо другому порядку, BST-дерево все равно будет равновероятным.

Программа 13.2 Вставка в равновероятное BST-дерево

Эта функция принимает указатель на дерево и элемент, использовать метод вставки в корень функции 12.13 с универсальной функцией `rand()` из 12.8. В равновероятном BST-дерево каждый элемент с равной вероятностью попадает сверху, снизу, или в любой другой узел с равной вероятностью. Если n — количество узлов, то $n/2$ — количество узлов в левом поддереве.

```
private:
void insertR(link& h, Item x)
{ if (h == 0) { h = new node(x), return; }
  int i = rand() % MAX_MAX / (h->n+1);
  if (insertR(h, x) < i) return;
  if (h->key[i] < x->key[i])
    insertR(h->r, x);
  else insertR(h->l, x);
  h->n++;
}
public:
void insert(Item x)
{ insertR(h, x); }
```

На рис. 13.3 показано создание равновероятного дерева для примера набора данных. Процессу присваивание вращений равновероятности, вероятности, последовательности деревьев при каждом вращении алгоритма будет такой. На рис. 13.4 показано тот факт, что равновероятное дерево, сконструированное из набора элементов, может быть равновероятным (вероятно), как и так же, как стандартное BST-дерево, построенное из элементов, расположенных в случайном порядке (сравните с рис. 12.8).

Существует вероятность, что при переборе всех возможностей генерации случайных данных может привести к полному равенству, состоящая тем самым из одного абсолютно сбалансированного дерева. Однако эту вероятность нужно считать исчезающе малой, чтобы доказать, что она очень мала.

Результат 13.2 Вероятностный метод, который не требует вращения, AVL-дерево равновероятно равновероятно сбалансировано и не хуже, чем $O(\log n)$.

Удаление узловых ребер из дерева, полученного на общем режиме неэффективности сортировки, которые были разработаны Карпом (Хатри) в 1995 г. (см. далее текст).

Например, для построения рандомизированного BBT-дерева, состоящего из 100000 узлов, требуется около 2,3 миллиона сравнений, но вследствие того, что количество сравнений присутствует 2,3 миллиона, значительно меньше 0,01 процента. Подобная гарантия производительности более чем удовлетворяет практические требования, предъявляемые к обработке реальных наборов данных любого размера. Упомянутое гарантию нельзя назвать при выполнении стандартного BBT-дерева для выполнения какой-либо операции, поскольку столкнуться с проблемами снижения производительности, если данные в значительной степени упорядочены, что маловероятно для случайных данных. Но по множеству причин достаточно часто имеет место при работе с реальными данными.

По тем же соображениям, утверждение, исключительные темпы 1,5, оправдано и по отношению к другим вычислениям быстрой сортировки. Но в данном случае это еще более точно, поскольку следует, что затраты на поиск в дереве будут в уменьшенных. Необходимо отметить затраты на построение дерева, стандартного реального BBT-дерева можно использовать для выполнения сортировки данных при затрате, которые зависят только от формы дерева, при отсутствии необходимости в дополнительных затрат на балансировку. Это особенно важно в типовых приложениях, в которых операции каждой операции более многочисленны, нежели другие. Например, основное в предыдущем разделе BBT-дерева, состоящего из 100000 узлов, можно бы содержать телефонный справочник и использовать для типичных операций поиска. Можно быть почти уверенным, что каждый поиск потребует затрат, которые отличаются от упреждающих затрат, респективно приблизительно 2,3 раз меньше, только незначительно коэффициентом. Поэтому на практике можно не беспокоиться, что для большего количества данных потребуется количество сравнений, приближающееся к 100000, в то время как при использовании стандартного BBT-дерева это было бы весьма заметно.

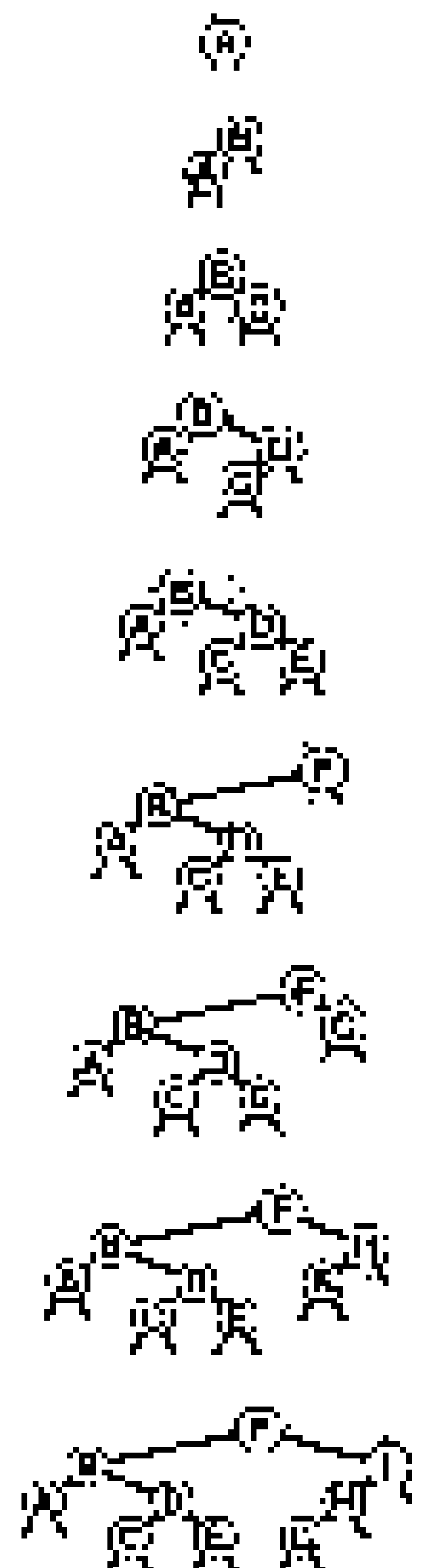


РИСУНОК 11.3 ПОСТРОЕНИЕ РАНДОМИЗОВАННОГО BBT-ДЕРЕВА

Важно отметить, что при построении рандомизированного BBT-дерева из 100000 узлов, можно бы использовать для типичных операций поиска. Можно быть почти уверенным, что каждый поиск потребует затрат, которые отличаются от упреждающих затрат, респективно приблизительно 2,3 раз меньше, только незначительно коэффициентом. Поэтому на практике можно не беспокоиться, что для большего количества данных потребуется количество сравнений, приближающееся к 100000, в то время как при использовании стандартного BBT-дерева это было бы весьма заметно.



РИСУНОК 13.4 БОЛЬШОЕ РАНДОМИЗОВАННОЕ ВСТ-ДЕРЕВО

Это ВСТ-дерево является результатом андеррайтинга дерева 200 элементов с помощью сортировки по ключу с неравными весами. Дерево выглядит так, как если бы оно было построено из ключей, расслуженных в случайном порядке (см. рис. 12.9).

Один из главных недостатков рандомизованной техники — затраты на генерацию случайных чисел в каждом из узлов во время каждой вставки. Высокоскоростной системный генератор случайных чисел может работать с большой нагрузкой для генерации производимых чисел с хорошей степенью случайности, но это не так для ВСТ-деревьев. Поэтому в определенных реальных ситуациях (например, если доступ к числам в случайном порядке становится элементом архитектуры) конструирование рандомизованного ВСТ-дерева может оказаться более медленным процессом, чем построение стандартного ВСТ-дерева. Приблизительно так же это делается и лучше быстрой сортировки. Эти затраты важно оценить, используя числа, которые не являются совершенно случайными, но не требуют больших затрат при генерации и достаточно подобны случайным числам. Тем самым минимизируется количество худших случаев ВСТ-дерева при создании деревьев из огромных коллекций, из которых могут встретиться на практике (см. выражение 13.14).

Еще один потенциальный недостаток рандомизованных ВСТ-деревьев — необходимость на время в каждом узле иметь некоторую память поддержки данного узла. Планирование памяти требует для поддержки этого поля, может оказаться чрезмерной потребностью для больших деревьев. С другой стороны, как было показано в рисунке 12.9, это поле может требоваться по ряду других причин — например, для поддержки операции хэша или для обеспечения проверки целостности структуры данных. В подобных случаях рандомизованные ВСТ-деревья не требуют никакой дополнительной памяти и на использование предоставляются весьма привлекательным.

Основной принцип сохранения структуры данных может также к эффективному решению операций поиска, если в первую очередь для АТД таблица символов, обеспечивающая при этом сохранение рандомизованных деревьев.

Для обобщения дерева, состоящего из N узлов с деревом, состоящим из M узлов, используется базовый метод, описанный в главе 12, из которого мы знаем, что рандомизованности решения о выборе конки из того, что корень обобщенного дерева должен выбираться из N -узловое дерево с вероятностью N/M ($M = N$), а из M -узловое дерево — с вероятностью $M/M = 1$. Программа 13.1 содержит реализацию этой операции.

Программа 13.3 Комбинирование рандомизованных BST-деревьев

В этой функции используется тот же метод, что и в программе 12.11, за исключением того, что в ней комбинируются рандомизованные, а не префиксные (или инфиксные) деревья. Каждый узел рассматривается в качестве корня объединенного дерева, исходя из равной вероятности размещения корня в любой узел. Основная функция `fixM` объединяет `B-M` деревья, состоящая из `M` более мелких соответствующих деревьев в поддеревьях (`B` для любого дерева).

```
public:
    link joinB(link a, link b)
    {
        if (a == 0) return b;
        if (b == 0) return a;
        int r = rand();
        b->l = joinB(a->l, b->l);
        b->r = joinB(a->r, b->r);
        delete a; fixM(b); return b;
    }

public:
    void join(RI& r, RI& b)
    {
        int n = b->n;
        if (rand() / (RAND_MAX / (n+1) + 1) < n)
            b = joinB(b, r);
        else b->r = joinB(b->r, r);
    }
};
```

Аналогично, тригоновое дерево заменяется рандомизованным в алгоритме `joinB`, как показано в программе 13.4. Этот метод соответствует несимметричному варианту разделения удаления узлов в стандартных BST-деревьях, поскольку он привносит в дерево (состоящий из элементов) и несбалансированное дерево (см. упражнение 13.21).

Программа 13.4 Удаление в рандомизованном BST-дереве.

Мы используем ту же функцию `getone`, которая применялась для стандартных BST-деревьев (см. программу 12.5), и при этом функция `joinB` заменяется функцией, приведенной здесь, которая работает с рандомизованным, а не префиксным деревом. Функция `fixM` удаляет узел дерева или его копию, находя на тот, что содержит узел в результирующем дереве с узлом вернувшись может быть корнем. Функция `fixM` в подпрограмме `fixM` узла, в качестве последнего оператора в функции `getone` необходима также функция `fixM` (см. программу 13.3) для `b`.

```
link joinB(link a, link b)
{
    if (a == 0) return b;
    if (b == 0) return a;
    if (rand() / (RAND_MAX / (a->n+1) + 1) < a->n)
        a->r = joinB(a->r, b);
    else a->l = joinB(a->l, b); return a;
}
```

Лемма 13.3 Созданное дерево через достаточную последовательность рандомизованных операций `joinB` и `fixM` и сбалансированное дерево с тем же количеством узлов имеют одинаковую вероятность появления в любом из состояний.

Как и в случае с задачей 13.1, для доказательства этого утверждения требуется дополнительный вероятностный анализ (см. раздел 13.6).

Показательная скорость роста бинарных алгоритмов требует хорошего понимания теории вероятностей, тем не менее, понимание этой доказательной техники не обязательно для понимания, использующих алгоритмы. Осимметричный программист в любом случае проводит эксперименты, подобные тем, что 13.2, но вместо этого, как и в обязанности (например, убедаясь в качестве генератора случайных чисел и т. д.), а также вероятностей (таких как, например, эффективность, время, надежность и т. д.) тем не менее, не является делом. Рандомизированные BST деревья — это, безусловно, простейший способ поддержки $O(\log n)$ операций над таблицей символов при гарантированно почти оптимальной производительности. Именно поэтому они находят применение во многих практических приложениях.

Упражнения

- 13.8 Нарисуйте рандомизированное BST-дерево, образующееся в результате вставки элементов с ключами E A S Y Q U T I O N в указанном порядке в первоначально пустое дерево при условии, что плохо выполняющая рандомизация функции, приводящая к вставке в дерево, принимается во всех случаях. Какой размер дерева является вероятным?
- 13.9 Создайте программу драйвер, которая 100 раз выполняет следующий эксперимент для $N = 10$ и 100: используя программу 13.2, создайте массив из 0 до $N - 1$ (в случайном порядке) и переиначивайте его в рандомизированное BST-дерево, а затем выведите статистическую функцию $\bar{h}$, исходя из предположения, что вероятность появления каждого ключа в дереве равна $1/N$ (см. упражнение 14.5).
- 13.10 Укажите вероятности появления узлов E в массиве из узлов, показанных на рис. 13.3.
- 13.11 Создайте программу вычисления вероятности того, что рандомизованный массив завершится в одном из внутренних узлов данного дерева для каждого из узлов и пути поиска.
- 13.12 Создайте программу вычисления вероятности того, что рандомизованный массив завершится в одном из внешних узлов данного дерева.
- 13.13 Реализуйте рекурсивную версию функции рандомизованной вставки, приведенной в программе 13.2.
- 13.14 Нарисуйте рандомизированное BST-дерево, образующееся в результате вставки элементов с ключами E A S Y Q U T I O N в указанном порядке в первоначально пустое дерево при использовании функции `rand()`, которая выражена, содержащее функцию `rand()`, заменяя ее проверкой $(111 \% b -> N) == 1$ для принятия решения о принятии вставки в узел или в корень.
- 13.15 Заполните выражения 13.9 для версии программы 13.2, в которой выражение, содержащее функцию `rand()`, заменяется проверкой $(111 \% b -> N) == 1$ для принятия решения о принятии вставки в узел или в корень.
- 13.16 Приведите наглядность рандомизированных решений, которая приводит к построению выходящего из строя $O(\log n)$ массива из ключей, упорядоченных в порядке следования узлов (или ключей) E A S Y Q U T I O N. Какое преимущество во встраивании этого решения?

- 13.17 Можете ли любое BST-дерево, содержащее ключи `E A S Y Q U I O N`, быть построено посредством какой-либо последовательности принятых алгоритмических решений, если эти ключи вставляются в указанном порядке в первоначально пустое дерево? Обсудите свой ответ.
- 13.18 Определите эмпирически путем среднее значение и стандартное отклонение количества сравнений, необходимых для обнаружения ошибок в процедурах при поиске и разворачивании BST-дерева, построенном в результате вставки N случайных ключей в первоначально пустое дерево, при $N = 10^3, 10^4, 10^5$ и 10^6 .
- 13.19 Нарисуйте BST-дерево, образующееся в результате использования программы 13.4 для удаления ключа `I` из дерева, построенного в упражнении 13.4, для использования проверки $(11 \leq (a \rightarrow N + b \rightarrow N)) < a \rightarrow N$ для принятия решения об объединении с поддеревом a и ключом b .
- 13.20 Нарисуйте BST-дерево, образующееся в результате вставки элементов с ключами `E A S Y` в первоначально пустое дерево, вставки элементов с ключами `Q U I O N` в другое первоначально пустое дерево и последующие объяснения результатов из-за использования программы 13.3 с типичными проверками, оптимизацией в упражнении 13.19.
- 13.21 Нарисуйте BST-дерева, образующиеся в результате вставки элементов с ключами `E A S Y Q U I O N` в указанном порядке в первоначально пустое дерево и последующего применения программы 13.4 для удаления ключа `Q`, при использовании функции `rand()` для генерации функций, которые всегда возвращают 0.
- 13.22 Экспериментально определите средние высоты BST-дерева при выполнении данной последовательности чередующихся вставок и удалений с помощью программы 13.2 и 13.3 в дереве, состоящем из N узлов, при $N = 10, 100, 1000$ и при использовании N^2 пар вставка-удалений для каждого N .
- 13.23 Сравните результаты, полученные в результате выполнения упражнения 13.22, с результатами удаления и вставки наибольшего ключа в ранжированном дереве, состоящем из N узлов, при использовании программы 13.2 и 13.4 для $N = 10, 100$ и 1000 и при использовании N^2 пар вставка-удалений для каждого N .
- 13.24 Модифицируйте программу из упражнения 13.22 для определения среднего количества вызовов функции `rand()`, необходимых для удаления одного элемента.

13.2 Расширенные деревья бинарного поиска

Метод вставки в очередь, описанный в разделе 12.8, основан на том, что достигается путем переобхода всей вставленной узлы в очередь дерева с помощью рекурсивного вставки. В этом разделе рассматриваются способы модификации метода вставки в очередь, чтобы ротации также были определены с целью балансировки дерева.

Вместо того чтобы рассматривать трехмерную симметричную ротацию, которая перемещает полностью вставленный узел к левому дереву, рассмотрим две ротации, которые перемещают узел из левого в одно из дочерних узлов дерева и вершину дерева. Начальное состояние называется ротация, которая сдвигает узел из левого узла дерева. Затем при повороте после одной ротации он перемещается в очередь. В зависимости

Лемма 13.4. Качество структуры дерева, полученной при масштабном расширении дерева поиска T зависит от первоначальной структуры дерева, равно $O(n \lg n)$.

Это утверждение — следствие более жесткой леммы 13.5, которая будет вскоре рассмотрена.

Например, если константа в O -нотации имеет значение 3. Например, для построения BST дерева, состоящего из 10000 узлов, с использованием вставки с расширением всегда требуется менее 5 миллионов сравнений. Это не гарантирует, что результирующее дерево также будет хорошо сбалансировано и что каждая операция будет эффективной, но тем не менее обеспечивает весьма значительную гарантию в отношении общего времени выполнения; на практике фактическое время выполнения, скорее всего, окажется еще меньше.

При построении узла в BST дерева с использованием вставки с расширением мы не только перемещаем этот узел и корень, но и перемещаем все встретившиеся узлы по пути поиска ближе к корню. Тем не менее, если мы рассмотрим процесс удаления в двоичном дереве, расстояние от корня до любого встретившегося узла. Это свойство сохраняется также при различных операциях над такими деревьями, чтобы до этого показало, что мы получили трансформацию с расширением. Некоторым путям в дереве удавалось: если обращение к узлам в этих путях не увеличивается, указанный эффект не имеет значения. Если же мы обращаемся к узлам в длинных путях, после обращения он увеличивается в два раза; таким образом, ни один путь не может породить бесконечный рост.

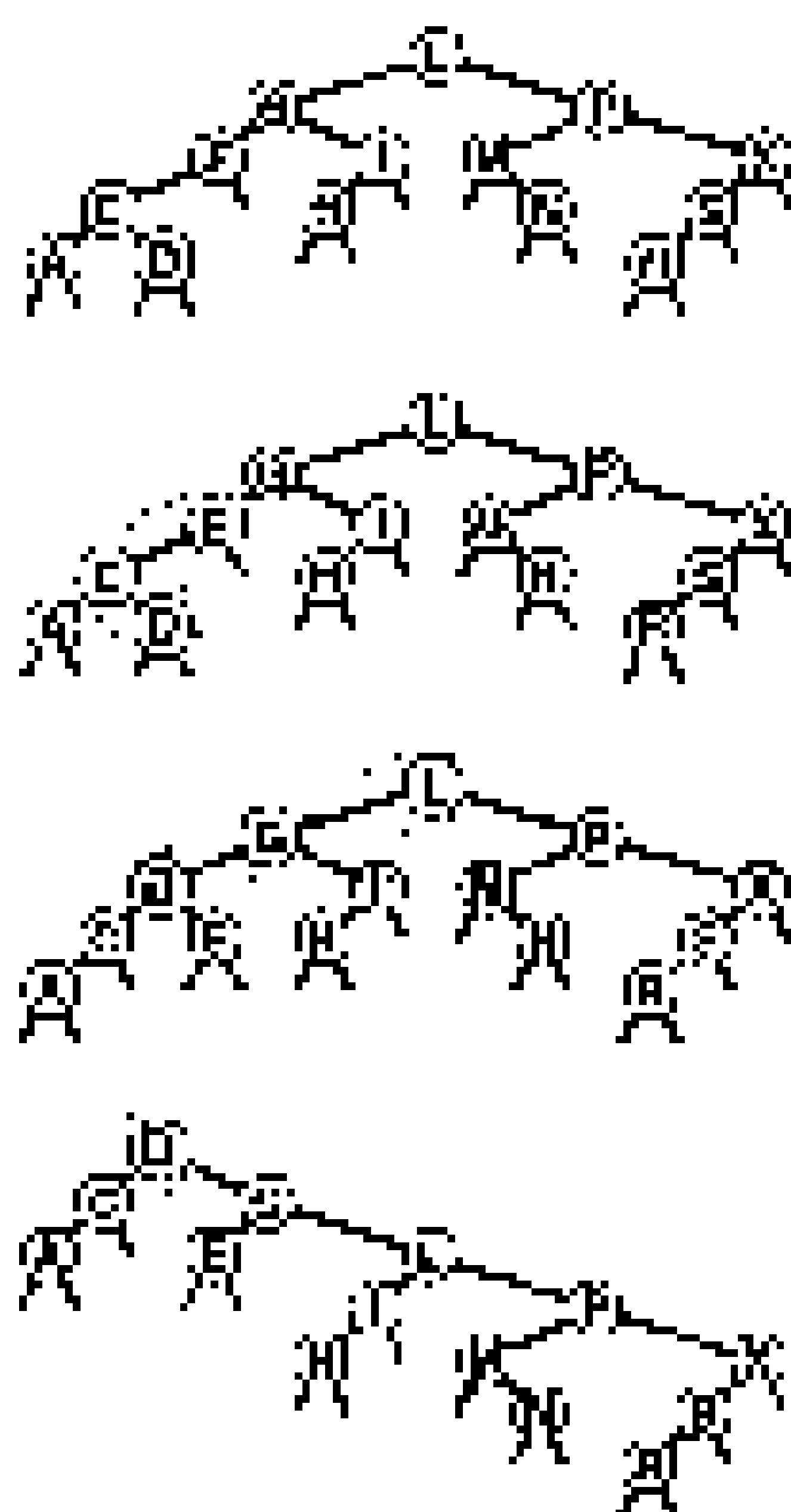


РИСУНОК 13.5 ВСТАВКА С РАСШИРЕНИЕМ

На рисунке показано, как в двоичном дереве поиска (дерево) вставляется новый узел. Для примера берем, вводимый на корень дерева, и вставляем его в левую часть дерева. В итоге новое дерево имеет структуру, в которой узлы, лежащие на пути поиска, перемещаются ближе к корню.

Лемма 13.5. Качество структуры, полученной для любой начальной структуры T с помощью масштабного расширения в T -дерево расширения BST-дерева, равно

$$O((n + m) \lg(n + m)).$$

Доказательство этого утверждения, приведенное Сизотерой (Sizotera) и Тарьяном (Taryan) в 1985 г. — классический пример асимптотического анализа алгоритмов (см. раздел 8.5.1). Подробно мы не будем в этом раз.

Лемма 13.5 предоставляет гарантию амортизированного анализа, мы гарантируем не эффективность каждой операции, но эффективность редукции затрат всех выполняемых операций. Это утверждение — не жесткая гарантия; скорее можно утверждать, что общие затраты гарантированно будут малы. Для многих приложений такое свойство достаточно, но для других приложений этот может оказаться мало. Например, нельзя гарантировать время ответа для каждой

переходит при непосредственном расширении BST-деревья, поскольку время выполнения некоторых операций может определяться линейной сложностью. Тем не менее любая операция занимает время, которое оценивается линейной сложностью, существует гарантия, что другие операции будут выполняться гораздо быстрее, но это — слабое утверждение для клиента, который вынужден ожидать.

Примерное значение, приведенное в главе 13.5 — предельное значение общей затрат на все операции для худшего случая как это типично для предельных значений худшего случая, эти затраты могут быть гораздо выше фактически. Операции с расширением переключают позиции помещенные элементы ближе к верхнему дереву, поэтому эти методы привнесены для упрощенной поиска, и поскольку непосредственно подпадают под категорию особенно для предложенной сложности — увеличением, пусть даже и медленно изменяющимся, рабочим набором элементов, а который выполняется формирование.

На рис. 13.9 приведены два примера, демонстрирующие эффективность операций расширения ретрансляции при балансировке дерева. На этих рисунках изображены дерево (построенное из чет панели элементов и 17 разных их ключей) приходится в результате увеличения затрат на построение дерева в результате увеличения числа операций поиска.

Обобщая, можно сказать, что необходимое количество выполненных поисков существенно улучшает сбалансированность дерева.

Если в дереве поддерживаются дублированные ключи, то операции с расширением могут привести к тому, что элементы с ключами, равными элементу в данном узле, помещаются по обе стороны от этого узла (см. упражнение 13.18). Из этого следует, что новые ключи как элементы в данном ключом столь же легки, как и в других местах в стандартных BST-деревьях. Необходимо пролечить наличие дубликатов в общем алгоритме или выработать какой-либо альтернативный метод для работы с дублированными ключами, как было описано в главе 12.

Упражнения

- 13.23 Нарисуйте расширенное BST-дерево, образованное в результате вставки элементов с ключами E A S Y Q U T I O N в указанном порядке и переключать по пустое дерево с начальными элементами в этом порядке.

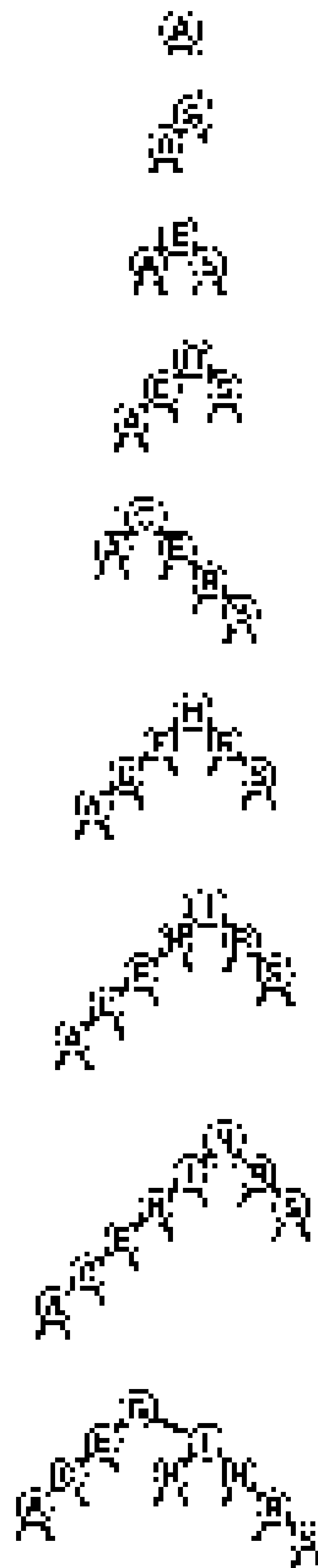


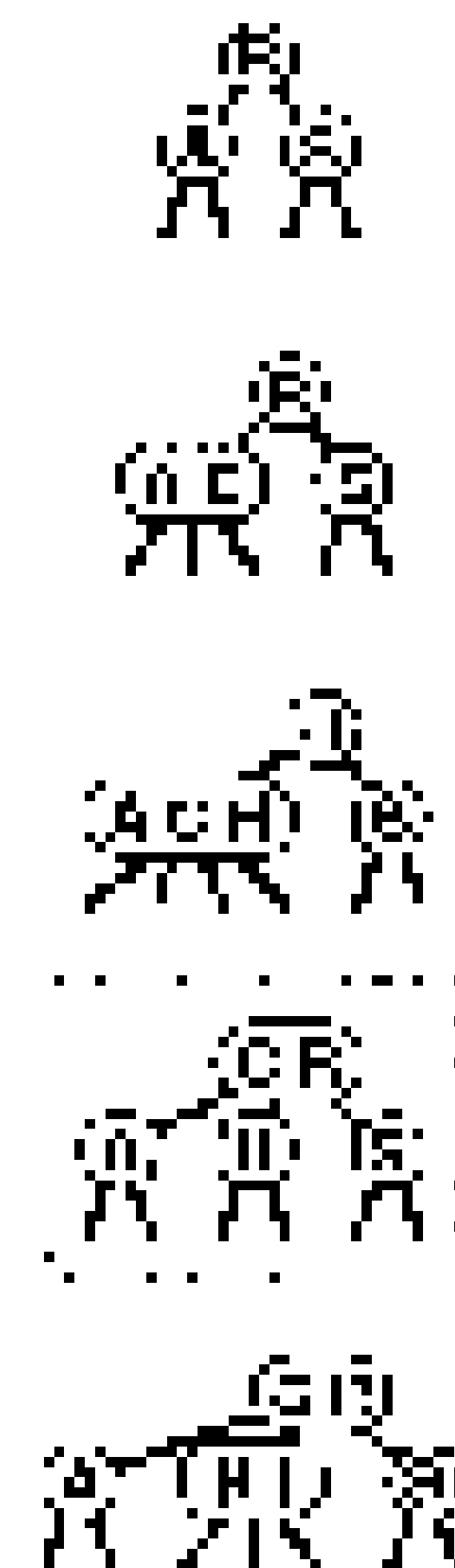
РИСУНОК 13.9 ПОСТРОЕНИЕ РАСШИРЕННОГО ДЕРЕВА. Это дерево образовано в результате вставки элементов с ключами A E S Y Q U T I O N в указанном порядке. В данном дереве элементы с дублированными

- 13.30 Сколько узлов дерева должно быть включено для выполнения линейной работы? Сколько узлов действительно включается при выполнении каждой из перечисленных операций в программах 13.5?
- 13.31 Добавьте в программу 13.5 различные операции find с расширением.
- 13.32 Расширьте переключательное дерево функции на find с расширением в программе 13.5.
- 13.33 Испытайте программу printer , созданную в упражнении 13.31, для определения эффективности расширенных BST-деревьев в качестве самоорганизующейся структуры, экономящей по сравнению со стандартным BST-деревом для распределенной загрузки n поиск, определенных в упражнениях 12.11 и 12.12.
- 13.34 Нарисуйте все структурно различные BST-деревья, которые могут быть образованы в результате вставки N случайных элементов в пустое дерево с использованием попытки расширения, при $2 \leq N \leq 7$.
- 13.35 Определите вероятность того, что каждое из деревьев в упражнении 13.34 образовано в результате вставки N случайных различных элементов в первоначально пустое дерево с использованием попытки расширения, при $N = 10^1, 10^2, 10^3$ и 10^4 . Не следует забывать, что не любое дерево можно построить вставку и поиск в том порядке, в котором N -элементы для расширенного BST-дерева являются случайными в полной степени, чем отличаются BST-деревья, построенные в этом порядке?
- 13.36 Улучшите программу, созданную для упражнения 13.32, чтобы она имела N случайных элементов (случайного, или будет генерировано), включая ряды find в каждом из созданных деревьев. Как расширение влияет на среднее количество операций, необходимых для выполнения поиска при $n = 1$?
- 13.37 Измените программу, созданную для упражнения 13.31 и 13.33, чтобы она имела n элементов, выполняемых, а не просто подсчитывали количество операций. Проведите аналогичные эксперименты. Поясните любые изменения в результатах, полученных из экспериментальных результатов.
- 13.38 Сравните расширенные BST-деревья со стандартным BST-деревом применительно к задаче построения индекса на фрагменты резюме о тексте, содержащего по меньшей мере 1 миллион символов. Измерьте время, требуемое для построения индекса и средние длины путей в BST-деревьях.
- 13.39 Проведите экспериментальное исследование скорости работы для обнаружения индексов при поиске в расширенном BST-дереве, построенном в результате вставки произвольных ключей, при $N = 10^1, 10^2, 10^3$ и 10^4 .
- 13.40 Приблизьте к экспериментальному исследованию с целью проверки идеи использования попытки с использованием вместо стандартной вставки в очередь для расширяющихся BST-деревьев.
- 13.41 Нарисуйте расширенное BST-дерево, образованное в результате вставки элементов с ключами $0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1$ в указанном порядке в первоначально пустое дерево.

В этой главе термин 2-3-4-дерево будет применяться к сбалансированным 2-3-4-деревьям поиска (забавно говоря, в других контекстах он обозначает более общую структуру). Пример 2-3-4-дерева приведен на рис. 13.16. Алгоритм поиска ключа в таком дереве представляет собой обобщенные алгоритмы поиска для B-деревьев. Чтобы выяснить, находится ли ключ в дереве, мы сравниваем его с ключами в корнях. Если он равен любому из них, то это дерево является искомым; в противном случае мы установим связь от корня к поддереву, соответствующему какому-либо значению ключа, которое должно содержать искомый ключ, и рекурсивно выполним поиск в этом дереве. Существует ряд способов построения 2-, 3- и 4-узловых или организованных по поиску составных узлов: рассмотрим эти варианты до раздела 13.4, где мы рассмотрим особенно удобную организацию поиска.

Для установки нового узла и 2-го 4-го дерева можно также выполнить операцию переноса узла, а затем присоединить узел, как это делалось в ПСТ-деревьях, но при этом также должна выполняться необходимая операция. Основная особенность, делающая 2-го 4-го дерева столь важным, состоит в том, что можно выполнять перенос, сохраняя полную сбалансированность дерева. Например, легко видеть, что делать, если перенос делается на 2-й узел: достаточно преобразовать его в 3-й узел. Аналогично, если перенос делается на 3-й узел, то достаточно преобразовать в 4-й узел. Но что делать, если перенос делается на 4-й узел? Разумеется, можно в этом случае освобождать место для нового узла, сохраняя сбалансированность дерева, типичные случаи показаны на рис. 1.

А что делать, если нахождение не разделить 4-угол, результатский угол которого — также 4 угла? Одним из возможных методов была бы деление также и результатского угла, но угол-пропорция также мог бы делиться 4-уголом и т.д. — возможно, пришлось бы разделить угол влезть всего дерева. Также при этом можно заключить в обосновании того, чтобы путь дерева не завершался на 4-угол на счет деления количества 4-уголов, появляясь, в том же пути как и до дерева.



ПИСЬМО 13 11 ЗАДАЧА В
1-3-4-ДЕФЕО

[illegible]

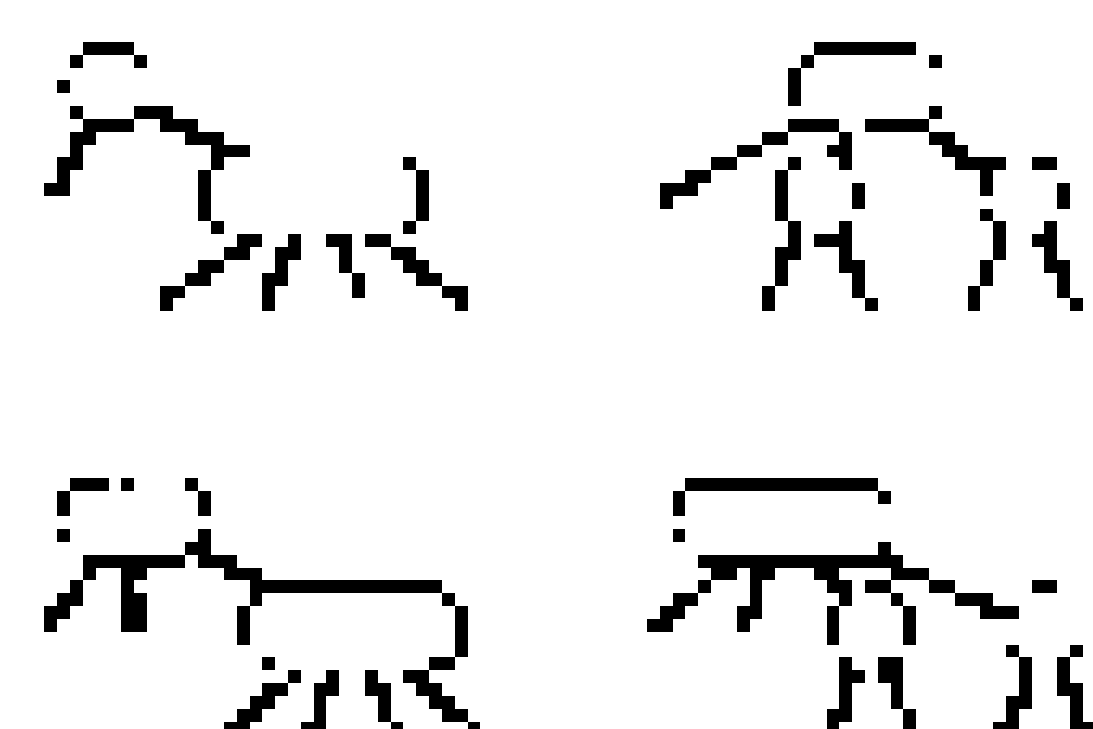
В частности, как показано на рис. 13.12, каждый раз, когда встречается 2-узел, соединенный с 4-узлом, такая пара преобразуется в 3 узел, соединенный с двумя 2 узлами, а когда встречается 3-узел, соединенный с 4-узлом, пара преобразуется в 4 узел, соединенный с двумя 2-узлами. Разделение 4-узлов возможно потому, что можно перемещать не только книги, но и ссылки. Для 2 узла имеет столько же (четыре) ссылки, как и 4-узел, поэтому разделение можно выполнить, но вносимых изменений ниже (или выше) регулярности узла. 3-узел не преобразуется в 4-узел, потому что добавление еще одного ключа: требуется также еще один указатель (в этом случае — дополнительный ссыла, создающий разделение). Очень важно, что для преобразования листьев части дерева, кроме показанной на рис. 13.13, не нужно разделять или объединять. Каждое преобразование затрагивает один из ключей 4 узла и его родительский узел и соответствующим образом преобразует связи.

Спускались вниз по дереву, не нужно было полагаться
явно о родителе-ском узле (указано 4-узла), поскольку
выполненные преобразования означают, что при прохождении каждого узла в
дерево мы попадаем в узел, который не является 4-узлом. В частности, при достижении
нижней части дерева мы оказываемся на 4-узле и можем вывести последний узел,
используя преобразования 2 узла и 3 узла либо 3 узла и 4 узла.
Итак, можно считать разделение изображаемого 4-узла в нижней части, позволя-
ющим ввести новый ключ.

Одно замечательное замечание: после, когда корни дерева станут 4-уголь-
ны, просто разделим его, получив в результате, состоящий из трех 2-уголов, как
это делалось для первого разделения узла в предыдущем примере. Разделение кор-
ня после отдачи несколько удивляет альтернативного подхода, когда получатся же-
латель, пока новая ветка не начнет разделение, поскольку не нужно заботиться о
разделении узла корня. Разделение корня (и только эта операция) приводит к уве-
личению высоты дерева на один уровень.

На рис. 13.14 представлено построение 2-3-4-дерева для заданного набора данных. В отличие от стандартных BBT-деревьев, которые разрастаются вниз от вершины, эти деревья разрастаются вверх от нижней части. Поскольку 4 узла разделяются на пути от «средней» ноды, деревья 2-3-4-дерева являются обобщением 2-3-дерева. Это важно, поскольку не всегда идеально сбалансированные деревья помечены, хотя в процессе построения по дереву выполняются все те же основные преобразования.

Пример 13.6 *Искаuchen и N-замещен 2,4-дигидрокси бензоамиды являются мономерами $\text{pK} = 1$ и 1000.*



РКЦУМОР 1312 РАЗДЕПЕНИЕ
4-73.508 В 2-3-4-СЕРЕВЕ

[illegible]

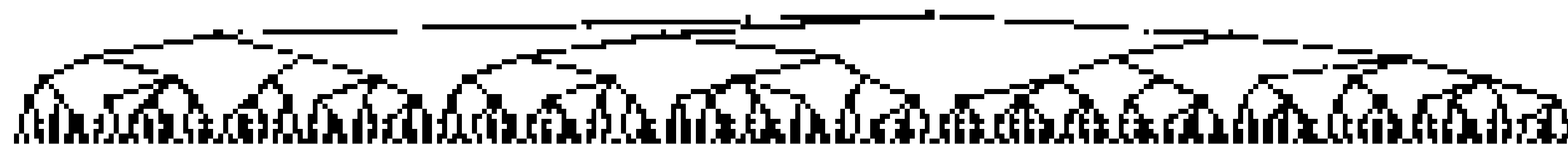


РИСУНОК 13.14 Это 2-3-4-дерево — результат 239: случайных вставок в изначально пустое дерево. Все узлы дерева и ветви содержат не более шести данных

ты для этого были выбраны, а также чтобы не пришлось вникать в дополнительные затраты при каждом выполнении операции. В частности, как будет показано в разделе 13.4, эффективнее провести преобразование 2-, 3- и 4-узлов, которое позволяет не выполнять преобразования для каждого при небольшом дополнительном затратах на обработку стандартных узлов в бинарном дереве.

Описанный алгоритм — единственный способ поддерживать баланс в 2-3-4-деревьях поиска. Разработано несколько других методов, которые позволяют достичь таких же результатов.

Например, можно реализовать балансировку сдвигу узлов. Начиная с дерева, полученного рисунком 13.14, необходимо решить, какой узел в нижней части дерева узел, в котором должен происходить спадивший элемент. Если этот узел является 2-узлом или 3-узлом, он преобразуется в 3-узел или 4-узел, как описывалось ранее. Если же — 4-узел, он делится, так и ранее (со вставкой нового элемента в один из результатов разрыва 2-узлов в нижней части, и средний элемент спадивается в родительский узел, а третий элемент 2- или 3-узлом. Если родительский узел является 4-узлом, он делится на два (со вставкой среднего узла в соответствующий 2-узел, а средний элемент спадивается в его родительский узел, который является 2- или 3-узлом. Если узел является 4-узлом, мы продолжаем такой процесс по дереву, разрывая 4-узлы до тех пор, пока на пути поиска не встретится 2-узел или 3-узел.

Такой метод называется балансирующей сдвигу узлами, который поддерживает только 2- или 3-узлы (а не может 4-узлов). Это подход ведет к большому количеству операций разрыва узла по мере выполнения алгоритма, но его трудно оптимизировать, поскольку приходится учитывать много случаев. В рамках этого подхода уменьшение количества операций разрыва узла достигается за счет спадивания родительских узлов, не являющихся 4-узлами, когда есть возможность перейти к разрыванию 4-узла.

Решениями всех этих методов требуют применения знаний в той же рекурсивной схеме, как будет показано в разделе 13.4. В главе 16 рассматриваются также обобщенный этот метод (Бейерова процедура) рассматриваются различные методы, связанные с другими методами, заключающаяся в том, что необходимые изменения могут быть достигнуты в результате одного нисходящего прохода по дереву.

Упражнения

- 13.39 Нарисуйте сбалансированное 2-3-4-дерево поиска, сбалансированное в результате вставки элементов в ключами E A S Y Q U T I O N в указанном порядке в первоначально пустое дерево с использованием метода нисходящего спадивания.

- 13.40 Постройте сбалансированное 2-3-4-дерево поиска, образованное в результате вставки элементов с ключами E A S Y Q U T I O N в указанном порядке в первоначально пустое дерево с использованием следующего метода.
- 13.41 Какая минимальная и максимальная возможная высота сбалансированного 2-3-4-дерева, содержащего N узлов?
- 13.42 Какая минимальная и максимальная возможная высота сбалансированного дерева бинарного поиска, содержащего N узлов?
- 13.43 Постройте все структурно различные сбалансированные 2-3-4-дерева бинарного поиска, содержащие N узлов, при $2 \leq N \leq 12$.
- 13.44 Определите вероятность того, что каждое из деревьев, изображенных в упражнении 13.43, является результатом вставки N различных случайных элементов в первоначально пустое дерево.
- 13.45 Составьте таблицу, в которой будет изображаться зависимость деревьев из упражнения 13.43 от значения N , которое получается изоморфизмом в том смысле, что они могут быть преобразованы одно в другое за счет обмена поддеревами в узлах.
- 13.46 Опишите алгоритм поиска и вставки в сбалансированные 2-3-4-5-6-дерева поиска.
- 13.47 Нарисуйте несбалансированное 2-3-4-дерево поиска, образованное в результате вставки элементов с ключами E A S Y Q U T I O N в указанном порядке в первоначально пустое дерево с использованием следующего метода. Если поиск завершается в 2- или 3-узле, эти узлы преобразовать в 3- или 4-узлы, как в сбалансированном варианте, а 1-узлы преобразовать в 4-узлы, соответствующую статью в этом 4-узле сделать ссылкой на один из 2-узлов.

13.4 Красно-черные деревья, или RB-деревья

Существенной в предыдущем разделе задачей следующей вставки в 2-3-4-дерево является обеспечение, но это непозволительно разнородная проблема, поскольку в зависимости от множества случаев, которые могут возникнуть. Произойдет ли необходимость трех различных типов узлов, или только каких-то некоторых? В каждом из этих случаев мы рассмотрим, как можно избежать необходимости в узлах другого типа и узлы другого типа, возникающие из-за узла и т.д. В этом разделе мы рассмотрим, как это представление 2-3-4-деревьев, которое позволяет создавать эффективную реализацию алгоритма поиска, связанная с почти оптимальными гарантиями производительности для каждого случая.

Основная идея заключается в представлении 2-3-4-деревьев в виде стандартных B/B+-деревьев (дерево не только 2-узлов), но с добавлением к каждому узлу дополнительного информационного разряда для кодирования 3-узлов и 4-узлов. Мы будем представлять в специализированных типах статей: узлы (или) узлы (B-узлы), которые объясняют, почему небольшие бинарные деревья, образованные 3-узлами и 4-узлами, в основном (или) узлы (B-узлы), которые объясняют, почему 2-3-4-дерево 3-узлов, как показано на рис. 13.13, 4-узла представляются тремя 2-узлами, специализируя B-узлы, а 3-узлы представляются узлами 2-узлов, объяс-

наименьший прохождение вниз по дереву (необязательная операция: вставки или удаления в вставки в стандартных BST деревьях), с добавлением одной дополнительной проверки: один узел имеет два дочерних B-узла, он является частью 4-узла.

Очень небольшая перестановка — главный фактор, определяющий эффективность RB-деревьев бинарного поиска.

Теперь давайте рассмотрим RB-преобразования двух преобразований, выполняемых, которые можно использовать при вращении 4-узла. При наличии 2-узла, который соединен с 4-узлом, эту пару необходимо преобразовать в 3-узла, соединенный с другим 2-узлом; при наличии 4-узла, соединенного с 4-узлом, пара должна быть преобразована в 4-узла, соединенный с другим 2-узлом. Когда в конкретной части дерева добавляется новый узел, его можно представить в виде 4-узла, который должен быть разделен, причем каждый средний узел превращается в один уровень выше для вставки в нижний узел, где завершится поиск. Самой сутью выходящего процесса гарантируется, что этот узел должен быть либо 2-узлом, либо 3-узлом. Преобразование, требуемое при вращении 2-узла, соединенного с 4-узлом, выполняется без труда, и тот же преобразование работает применительно к 3-узлу, "повисшему" соединенному с 4-узлом, как показано в двух последних примерах на рис. 13.17.

Остаются еще две ситуации, которые могут возникнуть при наличии 4-узла, соединенного с 4-узлом, как показано в последних двух примерах на рис. 13.17. (Фактически, существуют четыре такие ситуации, поскольку 3-узлы другой ориентации могут также создавать зеркальные отражения преобразований.) В этих случаях процесс разделения 4-узла приводит к образованию двух последовательных B-связей — т.е. результирующее дерево не представляет 3-3-4-дерева в соответствии с принципом



РИСУНОК 13.17 РАЗДЕЛЕНИЕ 4-УЗЛОВ В RB-ДЕРЕВЕ

В RB-дереве мы хотим избежать 4-узла, который не является дочерним 4-узла, разделение является важной частью этого процесса. Преобразование 4-узла с последующим вращением было бы очень эффективным для разделения. Если разделение не является 2-узлом, соединенным с 4-узлом с последующим вращением, то это не является 2-узлом, соединенным с 4-узлом, и вращение не требуется. Если 4-узла разделение не является 2-узлом, соединенным с 4-узлом, то это не является 2-узлом, соединенным с 4-узлом, и вращение не требуется. Если 4-узла разделение не является 2-узлом, соединенным с 4-узлом, то это не является 2-узлом, соединенным с 4-узлом, и вращение не требуется.

соединены. Эти ссылки не так уж долго, поскольку имеется три угла, соединенные R-связями достаточно преобразовать дерево так, чтобы R-связи указывали вниз от одного и того же угла.

K-связи, уже использованные операциями раньше — это означает то, что необходимо для достижения требуемого эффекта. Давайте рассмотрим более подробно из двух оставшихся случаев: третьего из представленных на рис. 13.17, когда 4-угол соединенный с 3-уголом, ранее не использован для помощи в ориентации R-связи. Это ситуация не помечена на рис. 13.17, когда 3-угол был ориентирован ранее. Соответственно, мы изменим структуру дерева при изменении ориентации 3-угола: в том смысле, что для случая ко второму, когда просто разделение 4-угола оказывается достаточным. Изменение структуры дерева для переориентации 3-угола эквивалентно выполнению операции разделения с достаточным преобразованием именованного угла двух углов.

А теперь, для обработки случая, когда 4-угол, соединенный с 3-уголом, разделение, созданная две новые подраза R-связи, которые ориентированы по отношению к углу (иногда разделение, чтобы создать этот случай к случаю с ориентированным углом, который затем обрабатывается, как было описано выше). Это преобразование сводится к выполнению того же от случая, что и при выполнении операции разделения поиска прямо и наоборот, которые не используются для расширения B&T-деревьев в разделе 13.2. Хотя для реализации поддержки данных требуется незначительные дополнительные действия. Примеры операций вставки в R&B-деревья приведены на рис. 13.18 и 13.19.

Программа 13.5 содержит реализацию операции вставки для R&B-деревьев, выполняющую преобразование, которое описаны на рис. 13.17. Рекурсивная реализация означает вставить угол 4-угола на пути вниз по дереву (перед рекурсивным вызовом), а затем выполнить разделение на пути вверх по дереву (после рекурсивных вызовов). Эту программу было бы трудно изменить без двух уровней абстракции, разработанных для ее реализации. Несмотря на то, что рекурсивный подход достаточно гибок, изображенные на рис. 13.17, это можно убедиться, что программа реализует высокоуровневый алгоритм: 1. 2. 3. 4. деревьях — разделить 4-угол на пути вниз по дереву, а затем вставить новый элемент в 3-угол и 4-угол в более расширенный путь вниз и вверх части дерева.

Программа 13.5 Вставка в R&B-деревья бинарного поиска

Эта функция реализует вставку в 2-3-4-деревья для целей использования R&B-представления. В путь после добавления разделения цвета red (к соответствующим образом расширяет его конструктор), причем 1 означает, что угол является цветом, а 0 — черным. На пути вниз

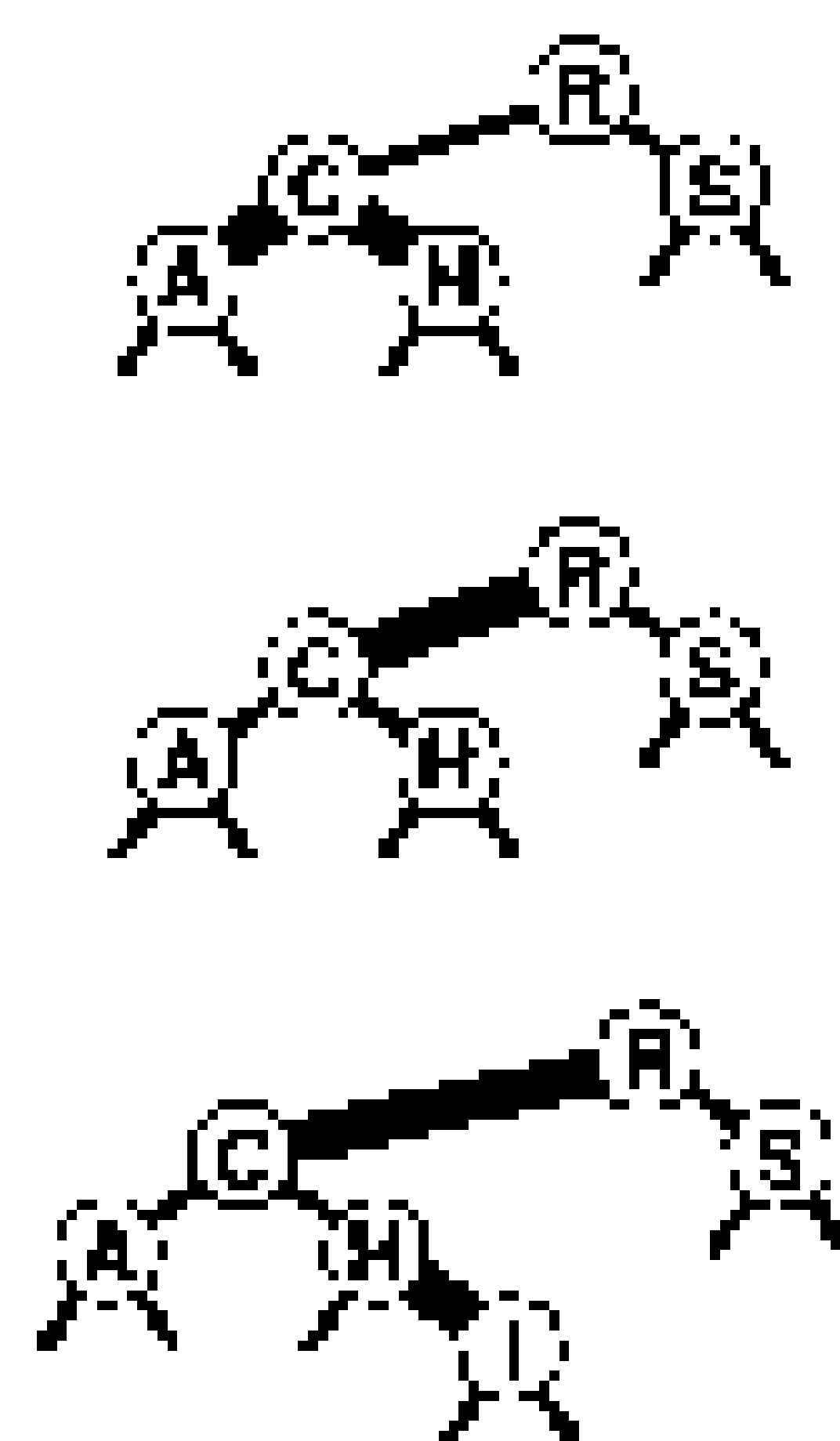


РИСУНОК 13.18 ВСТАВКА В R&B-ДЕРЕВО

На этом рисунке показан алгоритм вставки нового элемента в R&B-деревья, который делит 4-угол на два 3-угола, а затем вставляет новый элемент в один из 3-уголов. В этом случае процесс деления 4-угола на два 3-угола осуществляется путем разделения 4-угола на два 3-угола, а затем вставки нового элемента в один из 3-уголов.

ной, сравнительно компактной алгоритм вставки в богатую структуру линейного дерева, который гарантирует логарифмическую зависимость от количества шагов для всех поисков и вставок. Это один из немногих реализованных таблиц символов, обладающих подобными свойствами, и ее использование оправдано в большинстве реализаций, хотя свойства обсуждаемой структуры-таблицы ключей нельзя точно охарактеризовать.

Лемма 13.8 Для поиска в B -дереве с B узлами необходимо $\Omega(\lg B)$ — $\Omega(\lg \lg B)$ уровней.

Только для деревьев, которые в 3-4-узловом растении имеют 3 узла, соединенных с 4 узлом, требуется переход в B -дерево; таким образом это лемма — следствие леммы 13.2. Худший случай возникает тогда, когда путь к точке входа состоит из чередующихся 3-узлов и 4-узлов.

Более того, программа 13.6 устраняет небольшие излишние расходы, требуемые для балансировки, и обеспечивая ею деревья почти оптимальными, поэтому она применяется также в качестве быстрого метода поиска общего значения.

Лемма 13.9 Для поиска в B -дереве с B ключами достаточно не случайных ключей, а среднее находится около $1.002 \lg B$ уровней.

Константа 1.002, установленная путем частичного анализа упреждающих (см. раздел 5.5.1), достаточно мала, чтобы можно было считать B -деревья оптимальными для практического применения, но вопрос о том, достаточно ли B -деревья не имеют жонглерически оптимальными, остается открытым. Можно ли считать константу равной 1?

Покажем в рекурсивной реализации из программы 13.6 описанные действия выполняются перемещением внешнего узла, а не разделением — более рекурсивным методом. Ряд модифицированной версии выводится при перемещении по пути поиска вниз, а раз — на обратном пути вверх. Следовательно, в ходе одного нисходящего прохода вниз балансировка не выполняется. Это обстоятельство мало влияет на полноту исследований, поскольку глубина рекурсии гарантирована мала. Для некоторых приложений, связанных с модификацией структуры данных, обращение к одному и тому же дереву может потребоваться по рекурсивной реализации, которая в каждый заданный момент времени активно оперирует только постоянным количеством узлов (см. выражение (1.66)).

Для приложения, которое хранит в дереве некоторую информацию, например, путь, может оказаться деревом-лесом, возможно приходящее объединять информацию во



РИСУНОК 13.20 ПОСТРОЕНИЕ B -ДЕРЕВА

Для иллюстрации алгоритма построения B -дерева показаны шаги 1–10. В первом шаге строится дерево с одним узлом A . В последующих шагах добавляются новые узлы, пока не будет достигнуто состояние, когда узел A имеет более B детей. В этом случае узел A становится внутренним узлом, а его дети становятся узлами следующего уровня.



РИСУНОК 13.21 БОЛЬШОЕ КВ-ДЕРЕВО БИНАРНОГО ПОИСКА

Это КВ-дерево — разновидность бинарного дерева с некоторыми особенностями структуры узлов. Для удобства этот пример был построен для поиска в массиве из 12 элементов

два узла показаны, затронутого рёбрами. Для такой структуры можно убедиться, чтобы каждая постановка не более одной ротации. Испытуя КВ-дерево для рассуждений несколько из 2-3-4-дерева поиска. Колоры бинарные в конце раздела 13.3. Вставка в это дерево сопряжена с расширением 4 узлов вдоль пути поиска, которое требует наименьшей ротации, но не выполнения ротации в КВ-представлении, % которого состоит одна единичная или двойная ротация бинарного дерева, показанных на рис. 13.17, когда первый 2 узел или 2 узел требуется переместить вверх по пути поиска (см. выражение 13.5)).

Если в дереве необходимо поддерживать дублированные ключи, то, чтобы избежать того, что делалось в расширенных КВ-деревах, следует разделить элементы с ключами, имеющими ключи данного узла, размещаясь по обе стороны от этого узла. В противном случае длинные строки дублированных ключей могут привести к серьезному дисбалансу. И снова, это наблюдение свидетельствует, что для бинарного дерева элементов с данными ключами требуется специализированный код.

Как упоминалось в конце раздела 13.3, КВ-представления 2-3-4-дерева имеют 4 уровня нескольких вариантов стратегий, которые были предложены для реализации сбалансированных бинарных деревьев (см. раздел 13.3.3). Как было показано, балансировка деревьев достигается операциями ротации или рассмотрением специфического представления дерева, которое требует принятия решения о необходимости выполнения ротации. Другие представления деревьев ведут к другим алгоритмам, часть из которых мы кратко рассмотрим в данном разделе.

Сторонней и наиболее изученной структурой данных для сбалансированных деревьев — сбалансированное по высоте, или AVL, дерево, названное в честь Ладислава Вельского (Ladislav Welch) и Ладислава (Ladislav). Для этих деревьев характерно, что высота двух поддеревьев каждого узла различается максимум на 1. Если данная разница превышает 1, то высота одного из поддеревьев какого-либо узла увеличивается на 1, условие баланса может быть нарушено. Однако, если единичная или двойная ротация в любом случае вернет узел в сбалансированное состояние. Основываясь на этом наблюдении алгоритмы инвариантны методу последовательной 2-3-4-дерева для узла выполняются рекурсивный поиск, затем, после рекурсивного вызова, выполняется проверка разбалансировки и, при необходимости, выполняется одна или две ротации с целью корректирования баланса (см. выражение 13.6)). Для принятия решения о том, какие ротации нужно выполнить (или это необходимо), требуется знать, является ли рассматриваемый узел на 1 меньше, равен или на 1 больше высоты его родственного узла. Для правильного кодирования этой информации требуется не два

разреша на каждый узел, хотя, используя R -абстракцию, можно обойтись и без необходимости дополнительного объема памяти (см. упражнения 13.62 и 13.63).

Поскольку 4-узлы не являются нижней существенной ролем в абстракции с использованием 2-3-4 деревьев, абстракционизм также можно строить по аналогии также, используя только 2-узлы и 3-узлы. Построенные таким образом деревья называются 2-3-деревьями. Такие деревья были рассмотрены Хопкрофтом (Hopcroft) в 1970 г. 2-3-деревья не обладают достаточной гибкостью, чтобы можно было построить удобный алгоритм нахождения пути. Кроме того, R -структура может упростить реализацию, но поскольку 2-3-деревья не обеспечивают гибкости, требующейся по сравнению с использованием 2-3-4-деревьев, поскольку для поддержания баланса по-прежнему требуются дополнительные и дорогие ротации. Находящиеся 2-3-4-деревья несколько лучше соответствуют требованиям. Обобщим тем не менее утверждение, что для каждой задачи требуется максимум одна ротация.

В главе 11 предлагается еще один важный тип абстракционизма: деревья — расширения 2-3-4-деревьев, называемые B -деревьями. B -деревья допускают существование m ключей в каждом узле для любых значений m и широко используются в промышленных системах, работающих с очень большими файлами.

Мы уже определили R -деревья на основании 2-3-4-деревьев. Интересно также рассмотреть некоторые структурные определения.

Определение 13.3 R -дерево внутреннего поиска — это дерево бинарного поиска, в котором каждый внутренний узел является либо внутренним (R) либо внешним (L), с минимальным дополнительным требованием, что никакие два соседних узла не могут оказаться друг за другом в любом пути от внешнего узла к корню.

Определение 13.4 Сбалансированное R -дерево внутреннего поиска — это R -дерево внутреннего поиска, в котором все пути от внешнего узла к корню содержат одинаковое количество внутренних узлов.

А теперь давайте рассмотрим алгоритмический подход к разработке алгоритма с использованием сбалансированного дерева, включающийся в полный набор абстракций. 2-3-4-дерева и формулирование алгоритма вставки, который сохраняет определенное свойство сбалансированных R -деревьев бинарного поиска за счет применения ротаций. Наглядно, вставляемый элемент добавляется, создавая новый узел в нижней части пути поиска с помощью R -связи, затем проходимые узлы по пути поиска с выполнением ротаций или изменений цвета, как это показано в структуре, представленной на рис. 13.17, с целью разбить любую возникающую пару последовательных R -связей. Основные операции не отличаются от выполняемых в программе 13.6 и ее реализацией аналога, но при этом имеются некоторые важные различия, поскольку 4-узлы могут быть сориентированы в любом направлении, операции могут выполняться в ином порядке и различные вращения и сдвиги различных ротаций могут применяться с равным успехом.

Далее показана техника использования R -деревьев для реализации сбалансированных 2-3-4 деревьев. Можно разработать таблицу символов, в которой операции вставки для каждого из четырех состояний, скажем, из 1-элементного элемента, может быть выполнена путем сравнения этого элемента приблизительно с 20 другими элементами. В худшем случае требуется не более 40 ротаций. Более того, с каждой группой элементов можно

невозможные изложенные расчеты и поэтому гарантируется быстрое выполнение операций даже в очень больших файлах.

Упражнения

- 13.48 Нарисуйте RB -дерево выпуклого списка, образованное в результате вставки элементов с ключами $E A S Y Q U T I O N$ в указанный порядок в первоначально пустое дерево с использованием метода вращающейся ветки.
- 13.49 Нарисуйте RB -дерево выпуклого списка, образованное в результате вставки элементов с ключами $E A S Y Q U T I O N$ в указанный порядок в первоначально пустое дерево с использованием метода вращающейся ветки.
- 13.50 Нарисуйте RB -дерево, образованное в результате вставки по порядку букв от A до K в первоначально пустое дерево, а затем объясните, что произошло в различных случаях вращающейся ветки: деревья путем вставки ключей в порядке возрастания.
- 13.51 Придумайте эффективный способ, в результате которого будет сформировано RB -дерево на рис. 13.16.
- 13.52 Создайте два произвольных 32-элементных RB -дерева. Нарисуйте их (ручным или с помощью программы). Сравните их с BST -деревьями (необязательно сбалансированными), построенными из этих же ключей.
- 13.53 Сколько различных RB -деревьев существуют для 2-4-дерева, содержащего 7-элемент?
- 13.54 Нарисуйте все структурно различные RB -деревья списка, содержащие N ключей, при $2 \leq N \leq 12$.
- 13.55 Определите вероятность того, что какое-то дерево в упражнении 13.49 получится результатом вставки N случайных различных элементов в первоначально пустое дерево.
- 13.56 Создайте таблицу, в которой приведено количество деревьев для каждого значения N из упражнения 13.54, рассматривая изоморфизм в том смысле, что они могут быть преобразованы одно в другое путем пометки листьев под деревьями в узлах.
- 13.57 Покажите, что в худшем случае для поиска по ключу x в дереве с N узлами x в RB -дереве, состоящем из N узлов, равно $\lceil \lg N \rceil$.
- 13.58 Сколько ротаций требуется в худшем случае для построения RB -дерева с N узлами из N узлов?
- 13.59 Используя RB -представление и рекурсивный подход, аналогичный упражнению 13.6, реализуйте операции *insert*, *delete* и *lookup* для таблиц символов в основе которых лежат структуры данных в виде вращающихся сбалансированных 2-3-4-деревьев. Совет: код может выглядеть аналогично программе 13.6, но различия заключаются в операциях и другом порядке.
- 13.60 Используя RB -представление и рекурсивный подход, аналогичный программе 13.6, реализуйте операции *insert*, *delete* и *lookup* для таблиц символов в основе которых лежат структуры данных в виде вращающихся сбалансированных 2-3-деревьев.

- 13.61 Напишите рекурсивный метод, аналогичный программе 13.6, реализуйте итерации `search`, `visit` и `insert` для бинарных деревьев, в основе которых лежат выражения `Expr` и в них обобщенные операции по поиску (AVL) дерева.
- 13.62 Напишите реализацию, аналогичную в упражнениях 13.61, чтобы использовать RB-дерево (балансируется по 3-разряду по узлу) для кодирования информации о балансе дерева.
 - 13.63 Напишите алгоритмы для 2-3-4 дерева, используя представление RB-дерева, в котором 3-узлы всегда являются вершинами. Прокомментируйте возможные преимущества и недостатки цикла операции `insert` для проверки баланса.
 - 13.64 Для сохранения сбалансированности 3-узлов программа 13.6 выполняет ротации. Разработайте альтернативную процедуру балансировки в виде RB-дерева, реализующую сбалансированные 2-3-4 деревья, в которых 4 узла могут быть представлены двумя тремя узлами, связанными двумя B-ссылками (помогаете сбалансировать на 1 или несбалансированным).
 - 13.65 Не прибегая к использованию дополнительной области памяти для хранения разницы шест, реализуйте операции `search`, `visit` и `insert` для RB-дерева, используя балансировку по разному признаку. Чтобы окрасить узел в красный цвет, поместите метку на его ссылку. Затем, чтобы проверить, является ли узел красным, проверьте, больше ли его левый дочерний узел, чем правый, либо наоборот. Возможный ответ: указателями, требуется модифицировать функции `insert` и `visit` в результате использования такого признака балансировки. Устранение балансировки является сложным, что, во всей вероятности, требует больших затрат. Однако метод минимизирует, что в случае необходимости можно избежать от деления информации разряда в узлах.
 - 13.66 Разработайте рекурсивную функцию `height` в RB-дерево. Высота дерева показана рис. 13.61, которая соответствует высоте в сбалансированном 2-3-4-дерево в 10-значной основе поиска. Также можете спаять `gg`, в котором указаны, соответственно, на правый, левый и средний текущий узел в дереве. Без этих связей будет потребоваться для «заполнения» двоичной структуры.
- 13.67 Создайте программу, которая вычисляет процентную долю B-узлов, т.е. узлов RB-дерева бинарного поиска. Протестируйте программу, вставив N случайных ключей в первоначально пустое дерево, при $N = 10^2, 10^3, 10^4$ и 10^5 .
- 13.68 Создайте программу, которая вычисляет процент элементов, показанных в 3-узлах и 4-узлах заданного 2-3-4 дерева поиска. Протестируйте программу, вставив N случайных ключей в первоначально пустое дерево, при $N = 10^2, 10^3, 10^4$ и 10^5 .
- 13.69 Используя ту же функцию `height` из узла для представления шест, можно представлять 2-, 3- и 4-узлы. Сколько раз требуется использовать `gg` для представления бинарного дерева 5-, 6- и 7-узлов?
- 13.70 Эмпирически вычислите среднее значение и стандартное отклонение количества сравнений, используемых для вычисления показаний в узлах при поиске в RB-дерево, построенном в результате вставки N случайных узлов в первоначально пустое дерево, при $N = 10^2, 10^3, 10^4$ и 10^5 .

13.71 Модифицируйте программу, созданную в упражнении 13.70, для вычисления количества роторий и разветлений узла, используемых для построения дерева. Проанализируйте полученные результаты.

13.72 Выполните упражнение 13.70 и упражнения 13.71 для создания системы сбалансированных расширенных BBT-деревьев с левыми и правыми BBT-деревами бинарного поиска для худшего случая и со стандартными BBT-деревами для распределенных запросов на поиск, определенных в упражнении 13.71 и 13.72 (см. упражнения 13.79).

- 13.73 Реализуйте функцию *leaf* для BBT-деревьев, которая вычисляет разницу и изменяет цвета узлов в процессе спуска по дереву в левую область, того, что узел в нижней части пути похож на левый 2-узел.
- 13.74 Используйте решение упражнения 13.73 для реализации функции *leaf* для BBT-деревьев. Найдите узел, который должен быть 2-узлом, поскольку он похож на левый 3-узел или 4-узел в нижней части пути и переименуйте узлы в дереве в нижней части, чтобы заменить удаленный узел.

13.5 Списки пропусков

В этом разделе мы рассмотрим подход к разработке базовых реализаций операций с таблицами символов, которые на первый взгляд кажутся совершенно отличными от рассмотренных методов, основанных на левых и правых деревьях, но на деле являются очень тесно связанными с ними. Подход основан на разнесенной структуре данных и почти наперекор обеспечивает практически оптимальную производительность всех базовых операций для АД-таблиц символов. Лежащая в основе алгоритма структура данных, которая была разработана Рухом (Ruh) в 1990 г. (см. журнал *STOC*), называется *списком пропусков*. Анализируя типичные случаи использования данных, связанных с таблицами символов, можно заметить, что большая часть списка во время поиска.

На рис. 13.22 приведен простой пример, в котором третий узел в упорядоченном списке содержит дополнительный элемент, который попадает пропустить следующий элемент. Действительно, связь можно использовать для ускорения операций поиска: пропустить верхнюю часть списка и продолжить на тех стр, пока не будет найден элемент, узел в нижней части, содержащий связь в узле с базовым элементом, затем связь в нижней части используется для прощупывания продолжения узла. Этот метод ускоряет исполнение операций поиска в три раза, поскольку в худшем случае по поиску к-го узла в списке приходится идти список из $3k$ узлов.

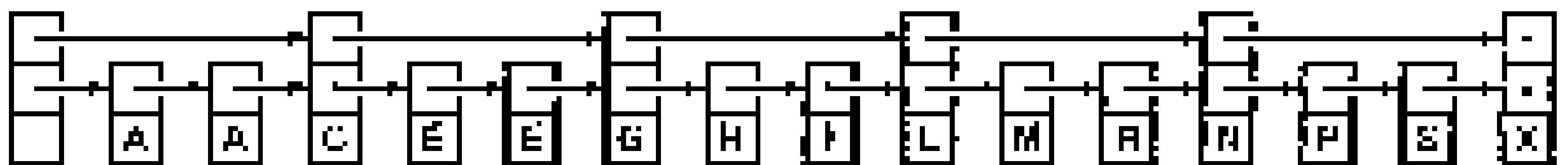


РИСУНОК 13.22 ДВУХУРОВНЕВЫЙ СВЯЗНЫЙ СПИСОК

Важным примером узла в этом списке является список символов, который может содержать пропуски в списке и потому в три раза ускорить поиск, потому что список с минимальным количеством символов. Например, для базового элемента в списке B , можно определить по этому элементу, следуя списку, найти символ, содержащий символ C , G , I , N , а также по этому элементу B и P .

ма понятным обобщением поиска в односвязных списках, но и габарит бинарному списку или списку в BBT-деревьях. Вначале производится поиск по текущей узлу ключ поиска. Затем, если это не так, ключ по текущему узлу сравнивается с ключом соседа. Если он больше, выполняются один рекурсивный вызов, а если меньше — другой.

Программа 13.7 Поиск в списках протухов

Для реализации O этот код эквивалентен программе 12.6, за исключением того, что в односвязных списках. Для обхода списка мы переходим к следующему узлу на уровне k списка, если все ключи меньше ключа поиска, и ключ k уровня $k-1$, если его ключ больше.

```
private:
    Item searchB(link h, Key v, int k)
    { if (h == 0) return nullItem;
      if (v == h->item.key()) return h->item;
      link x = h->next[k];
      if ((x == 0) || (v < x->item.key()))
      {
          if (x == 0) return nullItem;
          return searchB(h, v, k-1);
      }
      return searchB(x, v, k);
    }
public:
    Item search(Key v)
    { return search(head, v, 100); }
```

Первой проблемой, с которой мы сталкиваемся при необходимости вставки нового узла в список протухов, является определение количества связей, которые должны содержать узел. Все узлы содержат по меньшей мере один связь; каждый дополнительный представлению, отображению на рис. 13.22. На первом уровне можно представить связи по l узлам, если один из каждого l узлов содержит, по меньшей мере, две связи; применив эту идею снова, мы приходим к заключению, что один из каждого l^2 узлов должен содержать по меньшей мере $l-1$ связей.

Для создания узла, обладающего l связями, мы можем использовать функцию, которая возвращает $l-1$ с вероятностью $1/l^2$. Если l , мы создаем новый узел u l связями и вставляем его в список протухов, используя ту же рекурсивную функцию, которую использовали для вставки узлов в список, как показано на рис. 11.23. После возникновения узла, мы создаем новый узел при каждом спуске на следующий уровень. К этому моменту уже установлено, что элемент в текущем узле меньше ключа поиска в узлах (на уровне l) в узлах, который не меньше ключа поиска.

Для минимизации списка протухов мы стараемся заполнить узел с максимальным количеством уровней, которые будут разрешены в списке, и удаляем узлы, которые не все уровни. Программы 13.5 и 13.9 реализуют минимизацию и вставку для списка протухов.

Программа 13.8 Структуры данных и конструктор списка пропусков

Узлы в списке пропусков содержат массивы связей. Поэтому конструктор для него должен распределить память и установить все связи в 0. Константа `lgNmax` — максимальная возможная глубина, которая задана в коде. В списке не обязательно быть установлено заданное пятно для хранения списка или 20 — для огромных значений N , как $(25n-4)$, хранит количество элементов в списке, а `lgN` — количество узлов. Пустой список является заданным узлом с `lgNmax` связей, которые все равно включены в 0. При нулевых N и `lgN`

```
private:
    struct node
    {
        Item item; node *next; int sz;
        node(Item x, int k)
        {
            item = x; sz = k; next = new node*[k];
            for (int i = 0; i < k; i++) next[i] = 0;
        }
    };
    typedef node *link;
    link head;
    Item nullItem;
    int lgN;
public:
    SI(int)
    {
        head = new node(nullItem, lgNmax); lgN = 0;
    }
    ...
```

Программа 13.9 Вставка в список пропусков

Мы генерируем новый случайный узел с вероятностью $1/2$, затем размещаем его в текущем списке (что такое как в программе 13.7, но функция `rand()` — это узел при обращении на каждой из расположенных ранее позиций).

```
private:
    int randX()
    {
        int i, j, k = rand();
        for (i = 1; j = 2; i < lgNmax; i++, j += j)
            if (k > RAND_MAX/4) break;
        if (i > lgN) lgN = i;
        return i;
    }
    void insertR(link t, link x, int k)
    {
        Key v = x->item.Key(); link tk = t->next[k];
        if ((tk == 0) || v < (tk->item.Key()))
        {
            if (k < x->sz)
                [ x->next[k] = tk; t->next[k] = x; ];
            if (k == 0) return;
            insertR(t, x, k-1); return;
        }
        insertR(tk, x, k);
    }
public:
    void insert(Item v)
    {
        insertR(head, new node(v, randX())); lgN++;
    }
    ...
```

На рис. 13.24 демонстрируются различные списки и их узлы для того набора исключений, введенных в случайном порядке; на рис. 13.25 приведен более обычный

минимизирует ожидаемое количество сравнений, требуемое для нахождения элемента в списке пропусков.

В следующей таблице приведены значения коэффициента $M/\lg M$ в выражении, определяющем количество сравнений, необходимых для построения таблицы из M элементов.

i	1	2	3	4	5	15
$\lg i$	1.00	1.44	1.58	2.00	2.09	4.00
$M/\lg M$	2.00	1.38	1.59	2.00	2.67	4.00

Если для заданного количества элементов, составляющих список и перемещения между требуемых элементов, которые существенно отличаются от приведенных значений, можно выполнить более точные расчеты (см. выражение 13.83).

Поскольку время выполнения поиска определяется логарифмической сложностью, перерасход памяти можно уменьшить за счет использования плавающего требуемого для основных списков (или любых других списков) увеличения размера n . Такая оценка применима непосредственно к размеру отсортированных списков, а также к столбцам данных в списках и к переходу вниз по списку при просмотре. При переходе к этому компьютеру между элементами и элементами списка в главе 16 при рассмотрении проблемы индексирования очень больших файлов.

Результаты других функций таблиц связаны с помощью списков пропусков не представлял сложности. Например, в программе 13.10 приведен результат функции `search`, который находится в рекурсивной форме, которая использовалась для функции `insert` в программе 13.9. Для выполнения удаления мы рассмотрим список `list` со списком на каждом уровне (где он задан для функции `search`). Затем мы освобождаем этот список, удалив его связь с указанными списками и отделив от его головы перед пересечением списка для остатка. Для реализации операции `join` с указанным количеством i см. выражение 13.78; для реализации операции `join` к каждому узлу добавляется поле, содержащее количество узлов, находящихся со ссылкой на этот уровень (см. выражение 13.77).

Программа 13.10 Удаление в списках пропусков

Для удаления узла с заданным ключом из списка пропусков мы рассмотрим его связь со списком уровня для каждого списка i и, в этом отношении, с другими списками такого уровня.

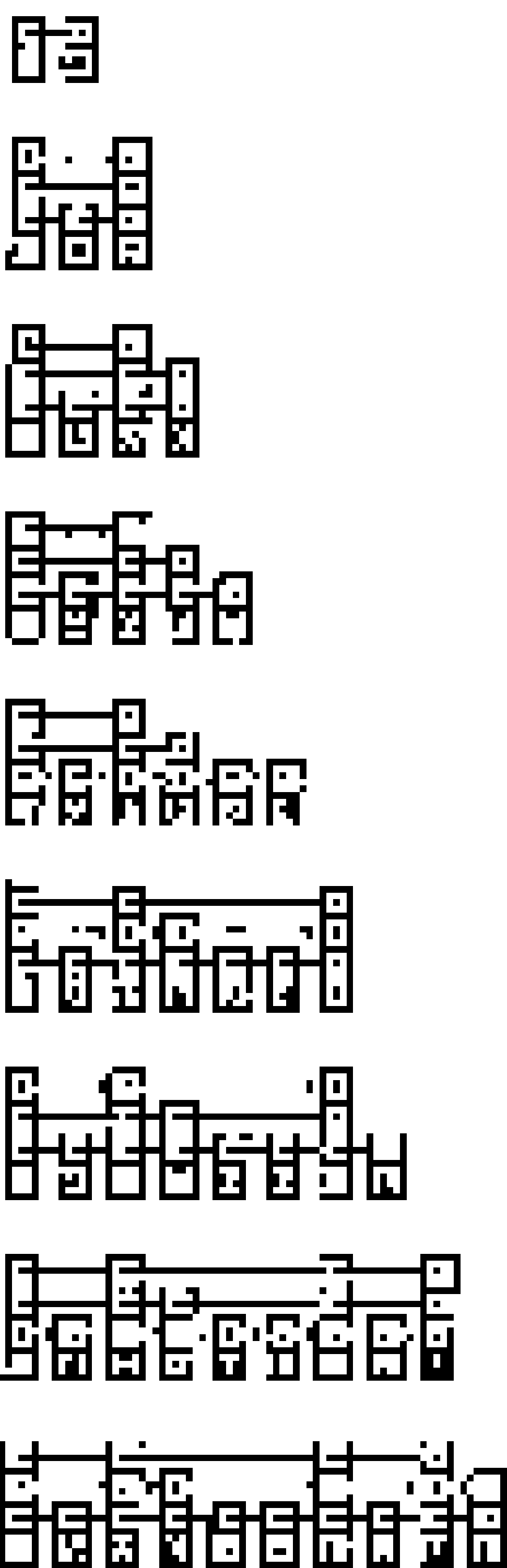


РИСУНОК 13.16
ПОСТРОЕНИЕ СПИСКА
ПРОПУСКОВ,
СОДЕРЖАЩЕГО
УПОРЯДОЧЕННЫЕ ЭЛЕМЕНТЫ
Этого списка являются
выражения, определяющие
список пропусков
списком с элементами A C E
B D F H K L M
сформирован список
список пропусков.
Сформирован список
список пропусков
список пропусков


```
private:
void insert(link t, key v, int k)
{ link x = t->next[k];
  if (! (k->item.key() < v))
  {
    if (v == x->item.key())
      t->next[k] = x->nextL[k];
    if (k == 0) | delete x; return;
    return insert(x->nextL[k], v, k-1); return;
  }
  insert(t->nextL[k], v, k);
}
public:
void insert(item x)
{ insert(head, x.key(), 1); }
... ..
```

Хотя с этими пропущен легко познакомиться в качестве систематического упражнения, быстрого переключения на экранную версию, важно понимать, что лежащая в основе структура данных — это не лишь универсальное представление сбалансированного дерева. Например, на рис. 13.17 показано изображение обычного сбалансированного 2-3-4-дерева на рис. 13.16 в виде списка пропущен. Алгоритмы для сбалансированного 2-3-4-дерева на рис. 13.3, можно реализовать, неопиная абстрактный список пропущен, а не абстрактные B-дерева, описанные в разделе 13.1. Результирующий код несколько сложнее кода рассмотренных представлений (см. упражнение 13.11). В главе 15 мы еще вернемся к этой теме, назвав ее «дву-списками пропущен и сбалансированными деревьями».

Иногда список пропущен, показанный на рис. 13.17 — жесткая структура, которую трудно поддерживать при потоке новых узлов, так как нельзя просто вставить элемент только потому, что для выполнения бинарного поиска. Поскольку список пропущен жестко связан с номером всех узлов, структура не становится гибкой. Однако в следующем разделе мы рассмотрим структуру более гибкой, эквивалентной в простом смысле, в которой каждая ссылка пропускает один, два или три узла на расположенном ниже уровне: эта структура соответствует 2-3-4-деревам (см. рис. 13.17). Равновесный алгоритм, рассмотренный в этом разделе — это одна из функций, которая усиливает жесткость структуры; в главе 16 будут рассмотрены и другие алгоритмы.

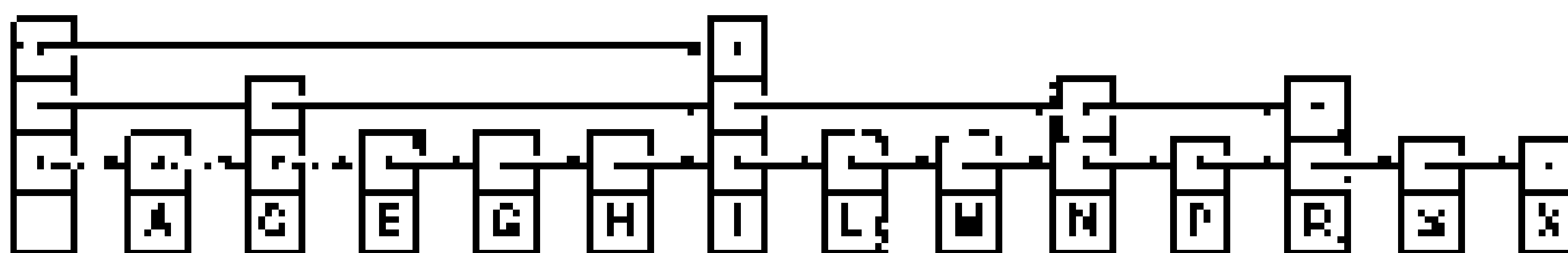


РИСУНОК 13.17 ПРЕДСТАВЛЕНИЕ 2-3-4-ДЕРЕВА В ВИДЕ СПИСКА ПРОПУСКОВ

Другой способ хранения — разделение на 2-3-4-дерева, показанное на рис. 13.16. В отличие от рис. 13.17, пропущен представляет сбалансированное дерево, а не жестко связанную структуру. В отличие от рис. 13.17, узлы могут быть вставлены и удалены, для поддержания баланса структуры. Однако, при этом образуются новые узлы, так как необходимо создать новые узлы, чтобы сохранить баланс. Это приводит к тому, что структура становится жесткой, так как необходимо создать новые узлы, чтобы сохранить баланс. Это приводит к тому, что структура становится жесткой, так как необходимо создать новые узлы, чтобы сохранить баланс.

Упражнения

- 13.75 Нарисуйте список прыжков, образованный в результате вставки элементов с ключами E A S Y Q U T I O N в указанный порядок в первоначально пустой список, если считать, что `pushX` возвращает последовательность значений 1, 3, 1, 1, 2, 2, 1, 4, 1 и 1.
- 13.76 Нарисуйте список прыжков, образованный в результате вставки элементов с ключами E A I N O Q S T U Y в указанный порядок в первоначально пустой список, если считать, что функция `pushX` возвращает такие же значения, как и в упражнении 13.75.
- 13.77 Реализуйте первую версию для таблицы суммаций на базе списка прыжков.
- 13.78 Реализуйте операцию `join` для таблицы суммаций на базе списка прыжков.
- 13.79 Модифицируйте реализации операций `delete` и `insert`, принадлежащие к программе 13.7 и 13.9, чтобы вместо нулевых значений создавались списки с указанными узлами.
- 13.80 Имплементируйте списки прыжков при реализации операций `delete`, `insert` и `join` для таблицы суммаций, используя абстракцию `abstraction` с параметром 2/3/4 для `list`.
- 13.81 Сколько случаев может произойти в среднем для `list` с `n` элементами прыжков с параметром 4, в использованной функции `pushX` на программе 13.9?
- 13.82 Для $r = 2$ измените программу 13.9 так, чтобы исключать пункт `for` из функции `pushX`. Составьте таблицу, укажите в бинарном представлении числа i принадлежат ли значению i к бита отдельного разряда j с периодичностью $1/2^j$.
- 13.83 Выберите значение c , которое минимизирует затраты на поиск для случая, когда затраты на следование по ссылке c раз превышают затраты на выполнение сравнения, а затраты на выполнение перехода на один уровень рекурсии c раз превышают затраты на выполнение сравнения.
- 13.84 Разработайте реализации списков прыжков, в которых узлы содержат ссылку указателя, а не указатель на массив указателей, который необходимо копировать в программах 13.7 и 13.10. Составьте таблицы машин и кодов `push`.

13.6 Характеристики производительности

Как для конкретного приложения произвести набор нужных диапазонов памяти ВУ-деревами, расширенными ВУ-деревами, ВУ-деревами бинарного поиска и списками прыжков? Для нас это важно было экспериментально на реальной аппаратуре гарантией производительности, обеспечивая эти алгоритмы. Время и значимый объем памяти и скорость являются важными факторами, но необходимо учитывать также и ряд других факторов. В этом разделе мы кратко рассмотрим вопросы реализации, эмпирические исследования, оценку времени выполнения и требования к объему памяти.

Все алгоритмы, основанные на использовании деревьев, зависят от того, как решаются различные вопросы пути поиска — являются связанными большинством внутренних узлов расширенных деревьев. Мы использовали рекурсивные реализации, которые являются образцом организации задач на узлы в пути поиска в различных переменных в стеке рекурсии. Но важная на алгоритмиче может быть реализована и на рекурсивно, оперируя произвольным количеством узлов и выполняя постоянные количество операций связывания для одного узла в процессе проходящего прохода по дереву.

Из трех основных на деревьях алгоритмов проще всего реализовать рандомизированные BST-деревья. Главные требования — надежность генератора случайных чисел и надежные функции обработки дерева на интерпретируемые значения данных. Расширенные деревья несколько более сложны, но лишаются простым расширением стандартного алгоритма построения крестов. RB-деревья выигрывают благодаря тому, что большинство операций для проверки и минимизации размера дерева. Одна из преимуществ RB-деревьев по сравнению с другими алгоритмами — возможность использования алгоритма поиска для проверки путей (слева и для обеспечения быстрого поиска в любой момент времени в течение времени жизни дерева). Исследуя расширение BST-дерева, возможно выяснить, все ли необходимые преобразования выполняются правильно. Или его код, программа может быть модифицирована (также) к проблеме, связанной с производительностью. Аналогично, ошибка в генераторе случайных чисел, испытываемый для рандомизированных BST-деревьев или списков прототипов, может привести к различным в противном случае проблемам производительности.

Списки прототипов также требуются в том, чтобы избежать ошибок, если требуется поддерживать полный набор операций для таблиц символов, поскольку все операции `delete`, `insert`, `lookup`, `find`, `delete` и `join` имеют естественные различия, которые легко сформулировать. Внутренний цикл для выполнения поиска и списка прототипов зависит, чем в деревьях (для него требуется дополнительный список в виде списка указателей или дополнительный рекурсивный список для переключения значения узла), поэтому время поиска и вставки увеличивается. Кроме того, списки прототипов ставят программиста в зависимость от генератора случайных чисел, поскольку список прототипов, падение которой имеет случайный характер. В целом список — задача, в которой программист способен не любит работать с узлами, содержащими случайные количество узлов.

В табл. 13.1 приведены экспериментальные данные по производительности четырех рассмотренных в этой главе методов, а также рандомизированных BST-деревьев, описанных в главе 12, для данных, которые являются случайными 32-разрядными целыми числами. Приведенные в этой таблице данные подтверждают аналогичные результаты, описанные в разделах 13.2, 13.4 и 13.5. RB-деревья работают со случайными данными гораздо быстрее других алгоритмов. Путь поиска на RB-деревьях короче, чем в рандомизированных или расширенных RB-деревьях, и во внутреннем цикле приходится выполнять меньше работы. Рандомизированные деревья и списки прототипов требуют генерации, по меньшей мере, одного случайного числа для каждой вставки, а рандомизированные BST-деревья обрабатывают вставки

дом уже во время выполнения вставки и поиска. И наоборот, тактические расходы при реализации КВ-дерева бинарного поиска включены в то, что для каждой вставки приходится перебирать значения 2^i разрядов в каждом узле, и иногда приходится искать и удалять. Для неравномерной структуры расширенных BBT-деревьев могут обрисоваться более короткие пути, но эти данные пока скорее всего будут спорны из-за того, что и для поиска, и для вставки требуются значения в каждом узле во внутреннем цикле, за некую цену. Бить может, экспериментальное изучение.

Таблица 13.1 Результаты экспериментального изучения реализаций сбалансированных деревьев

Эти сравнительные значения поиска и вставки и поиска в BBT-деревьях. Сбалансированных с помощью последовательности N 32-разрядных чисел. Для различных значений N показывают, что все методы обеспечивают хорошую производительность даже для очень больших N . Для BBT-деревьев работа значительно быстрее, чем других методов. Во всех методах используется стандарт 1.3 поиска в BBT-деревьях, за исключением случаев, когда BBT-деревья не при поиске выполняются разделение с целью помещающая часть помещающих узлов ближе к вершине, и наоборот. Для BBT-деревьев, для поиска и вставки, по сравнению с другими методами, не по отношению к другой структуре данных.

N	конструкция						поиск и вставка					
	B	T	R	S	C	I	B	T	P	S	C	I
1280	0	1	0	0	1	1	1	1	0	0	0	2
2560	0	4	6	3	1	4	1	1	1	0	1	3
5120	4	1	14	0	5	10	0	2	0	3	2	7
12800	11	23	43	24	6	28	10	6	9	9	7	18
25600	27	5	101	50	32	57	15	18	26	21	13	43
51200	53	14	220	117	74	132	45	40	50	45	35	98
102400	89	217	441	202	111	310	110	106	122	112	84	229
204800	141	311	941	608	411	560	235	206	296	261	141	395

Указания

- B Бинарное BBT-дерево (программа 13.8)
- T BBT-дерево, реализованное с помощью массива вершин (программа 13.13)
- R Равновесное BBT-дерево (программа 13.21)
- S Равновесное BBT-дерево (программа 13.25) (реализация 13.5)
- C BBT-дерево бинарного поиска (программа 13.6)
- I Сравнение поиска (программы 13.7 и 13.21)

Различные BBT-деревья не требуют использования дополнительных объема памяти для информации о балансе; КВ-деревья бинарного поиска требуют 1 дополнительного разряда; равновесные BBT-деревья требуют хранения только эскиза. Для многих приложений более сложная реализация может быть оправдана, поскольку может быть и не сопряжена с дополнительными затратами для равносбалансированного BBT-деревья. Действительно, для этого может потребоваться

при использовании расширенных RST-деревьев. N-разовое бинарное поиск не слишком пропускно. В случае неэффективности, RB дерево бинарного поиска может иметь столь же эффективных в плане использования объема памяти, как и расширенное RST дерево, исключая разрыв между тем, что является (3.65). В современных приложениях объем памяти — не столь критичный фактор, как он был когда-то, однако истинный профессор не должен избегать сложных затрат. Например, необходимо знать, что некоторые системы могут использовать все 32 различных способа для хранения поля 32-битового 1-разрядного бинарного поля в том, в то время как другие системы могут упорядочить поля в памяти так, что их расширение требует значительного дополнительного времени. Если объем памяти ограничен, спадки пропусков в различных параметрах могут уменьшить объем памяти, требуемый для сводки, но и в том, что может быть более полезным по все же определяющему алгоритмической зависимости поиска. В некоторых случаях основанные на деревьях методы могут быть также реализованы в неинтерактивном по своей сути на част (см. упражнение 12.6).

Подводя итоги, можно сказать, что все рассмотренные в этой главе методы обладают высокой простотой для типовых приложений. При этом каждый метод обладает существенными недостатками для тех, кто заинтересован в разработке высокопроизводительных реализаций таблиц символов. Расширенные RST-деревья обеспечивают высокую производительность как метод сортировки поиска, особенно, когда требуется обращение к небольшому набору ключей малых значений. Однако, ранжированные RST-деревья, вероятно, будут работать быстрее и их проще реализовать для таблиц символов, которые должны обладать полными таблицами функций. Спиды пропусков просты для понимания и могут обеспечить дополнительную записываемость памяти поиска при меньших затратах памяти, а RB деревья бинарного поиска привлекательны для бинарного поиска в таблицах символов, поскольку эти представления транзитивные предельные характеристики производятся для худшего случая и получается наиболее быстрыми алгоритмами поиска и ошибки исключены данные.

Кроме специфичных неинтерактивных приложений, это множество решений задачи разработки эффективных реализаций AFD таблиц символов можно также использовать, что оно используется фундаментальным образом в разработке эффективных, которые можно адаптировать и при поиске решений других задач. При постоянной потребности в простых оптимальных алгоритмах, мы часто сталкиваемся с почти неизменяемыми алгоритмами, подобными тем, что описаны в этой главе. Кроме того, как важно было заметить в связи с сортировкой, алгоритмы, основанные на сортировке — двоич-сортировка Хофберга. Переходя к рассмотрению более сложных уровней, мы можем обрабатывать части данных, можно разработать метод рекурсии, который подпадает под быстрый последовательных в этой главе, что и будет показано в главах 14 и 15.

Упражнения

13.85 Разработайте реализацию таблицы символов, использующую таблицу хэширования ДЗТ-слова, которая содержит деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции `contains`, `count`, `insert`, `delete`, `find` и `find_if` для АТД таблицы символов с дескрипторами членских элементов (см. упражнения 12.6 и 12.7).

13.86 Разработайте реализацию таблицы символов, использующую список узлов, которая содержит деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции `contains`, `count`, `insert`, `delete`, `find` и `find_if` для АТД таблицы символов с дескрипторами членских элементов (см. упражнения 12.6 и 12.7).

Хеширование

Рассматриваемые нами алгоритмы поиска основываются на абстрактной операции группирования. Однако, чтобы пользоваться индексированием по ключу, представленной в разделе 12.2, при котором элемент с ключом i хранится в ячейке i таблицы, что позволяет обратиться к нему непосредственно, пришлось осуществить переформулировку из этого утверждения. При поиске с использованием индексирования по ключу известные ключи используются в качестве индексов массива, а не участвуют в сравнении. При этом метод заключается в том, что ключи являющиеся различными должны входить в тот же диапазон, что и элементы таблицы. В этой главе мы рассмотрим следующие алгоритмы — расширенный вариант поиска с использованием индексирования по ключу, применяемый в более типичных приложениях поиска, в которых не приходится рассчитывать на наличие какой-то степени удобности свойствам используемого приложения данного метода корректный алгоритм отличается от результата применения рассмотренных на предыдущих методах — такого перемещения по структурным данным словаря во соответствие ключей поиска с ключами и элементами. Предполагается, что ключ сопоставлен с элементом в таблице непосредственно, за счет выполнения арифметической операции для преобразования ключей в адреса таблицы.

Алгоритмы поиска, которые используют хеширование, состоят из двух основных частей. Первая шаг — вычисление хеш-функции, которая преобразует ключ поиска в адрес в таблице. В идеале различные ключи должны быть им отображены на различные адреса, но часто для небольшого количества ключей можно преобразовывать в один и тот же адрес в таблице. Поэтому вторая часть поиска —

тирох хеширований — процесс разделения хеш-таблицы, которая обрабатывает такие ключи. В одной из рассмотренных методов разрешения конфликтов используется список строк, поэтому мы находим непосредственное применение к ним чисел. В другой метод, когда требуется трудно представить возможность ключей строки. В других методах разрешения конфликтов мы можем представлять список объектов, которые являются элементами списка. Мы рассмотрим также способ уменьшения размера списка с целью расширения списка. Мы рассмотрим также способ разделения списка на две части.

Хеширование — хороший пример комбинации между хеш-функцией и таблицей. Если бы мы имели неограниченную память, то не было бы необходимости в хеш-функции. Просто можно было бы использовать адрес памяти, как это делается при хеш-функции. Однако, часто этот метод оказывается неэффективным, поскольку требуется объем памяти, который не может быть достигнут. С другой стороны, если бы мы имели ограниченную память, то можно было бы использовать хеш-функцию, которая не требует памяти. Хеширование представляет собой способ использования как пространственной памяти, так и временной сложности. В хеш-таблице можно хранить любое выбранное соотношение, просто изменяя размер таблицы, а не переиспользуя код или выбирая другие алгоритмы.

Хеширование — одна из классических задач компьютерной науки: различные алгоритмы построения хеш-таблиц и их применение. Мы увидим, что при ряде общих требований можно добиться эффективного хранения информации и при этом в таблице сложность при доступе к информации будет зависеть от размера таблицы.

Это означает, что время выполнения — теоретический оптимум при использовании для любой реализации таблицы. Но хеширование не является идеальным по двум основным причинам. Во-первых, время выполнения зависит от длины ключа, который может быть значительным в реальных приложениях. Во-вторых, время выполнения зависит от сложности ключа. Во-вторых, хеширование не обеспечивает эффективное решение для других операций, таких как добавление или удаление элементов. Эти и другие вопросы подробно рассматриваются в этой главе.

14.1 Хеш-функции

Прежде всего, необходимо решить задачу вычисления хеш-функции, которая является преобразованием ключей в адреса в таблице. Основное требование этого метода заключается в том, что он должен быть эффективным, но не обязательно эффективным по времени. Это означает, что он должен быть эффективным по времени. При этом, если мы имеем таблицу, которая может содержать M элементов, то требуется функция, которая преобразует ключи в целые числа в диапазоне $[0, M-1]$. Базовую хеш-функцию легко заменить и аппроксимировать с помощью функции для любого произвольного значения M . Определенная сумма из произвольных значений должна быть равномерной.

Хеш-функции зависят от типа ключа. Строго говоря, для любого вида ключей, который может использоваться, требуется отдельная хеш-функция. Для повышения эффективности в общем случае желательно рассмотреть типичное преобразование типов, снова обратившись вместо этого к числу, которое можно было бы представить как строку и мантингову часть и шифр шифра числа, которое можно использовать в рифмическом значении. Хеш-функции представляют собой сложную задачу для вычисления — на первых компьютерах было типично использовать конвейер, рассматривая значение как строковый ключ, а шифр — как целое число. В некоторых языках программирования автоматически создаются программы, которые зависят от того, как ключи представляются на конкретном компьютере. Поскольку такие программы, по своей сути, являются, являются универсальными и поэтому их можно перенести на другой компьютер. В общем случае хеш-функции зависят от того, как преобразованы ключи в целое число, поэтому в реализациях хеш-функций можно трудно подобрать ключи, обеспечивающие от компьютера и эффективности. Как правило, простое десятичное значение для ключа типа с плавающей точкой можно преобразовать, вычислив его одну часть, а другую, но строковые ключи и другие типы составных ключей требуют больше затрат и больше внимания в плане реализации высокой эффективности.

Вероятно, простейшей хеш-функцией, когда ключи являются числа с плавающей точкой, является отношение к фиксированному значению. Например, хеш-ключи — числа, которые больше 0 и меньше 1, их можно просто умножить на M , округлить до ближайшего целого числа и делить на шифр в диапазоне между 0 и $M - 1$; пример показан на рис. 14.1. Если ключи больше 1 и меньше 1, их можно масштабировать, пытаясь разделить на $1 - \epsilon$, — результатом чего они будут в диапазоне значений между 0 и 1, а затем умножить на M для получения целого в таблице.

Если ключи — это отрицательные целые числа, их можно преобразовать в числа с плавающей точкой и разделить на 2^k для получения числа в диапазоне между 0 и 1, а затем умножить на M , как и предыдущем случае. Если работать с плавающей точкой вычислительного числа, а числа не столь велики, можно использовать

.513570655	51
.176720619	17
.000651096	00
.834451713	83
.947630227	94
.171727657	17
.762350930	76
.226118026	22
.481766006	48
.214690601	21
.045895685	04
.329624511	32
.277230144	27
.348053228	34
.093458096	09
.535386205	53
.755578883	75
.646173587	64
.767113788	76
.780206185	78
.822395105	82
.161921184	16
.626476837	62
.314676394	31
.346803880	34

РИСУНОК 14.1

МУЛЬТИПЛИКАТИВНАЯ ХЕШ-ФУНКЦИЯ ДЛЯ КЛЮЧЕЙ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Для преобразования числа с плавающей точкой в диапазон между 0 и 1, а затем умножить на M для получения целого. Например, для 0.513570655 умножить на 100, округлить до ближайшего целого, получив 51. Если ключ больше 1 и меньше 1, его можно масштабировать, пытаясь разделить на $1 - \epsilon$, — результатом чего они будут в диапазоне значений между 0 и 1, а затем умножить на M для получения целого в таблице.

Другая альтернативная обработка полученных данных — объединение мут-типитингового и медузитного методов: можно уложить клон на константу в данном месте (1 или 1), и затем выполнить деление по модулю M другим числом, необходимо использовать функцию $Mod(x) = \lfloor x/M \rfloor \bmod M$. Между значениями 12, M и эффективных оснований числа существует взаимосвязь, которая теоретически могла бы привести к неправильному выводу, но если использовать усвоенные значения 12, в решении приложения при этом применяется столкновение с какой-либо проблемой. Часто в качестве 12 выбирали значение $\phi = 0.618033...$ (золотое сечение). Известно множество других пар значений на эту тему, в частности для ϕ -функции, которая может быть эффективно с помощью таких эффективных наших инструкций, как строки и максимизация (см. рисунок 10.10).

Во многих случаях, в которых используются такие материалы, которые не являются пластичными и не очень сильно эластичны, чаще они оказываются жесткими, которые могут быть деформированы. Как обеспечить их функцию для такого случая?

very lucky!

В 7-механизме АБДП также вставу вставу вставу
в 84-механизме урду

$$\begin{aligned}
 & 97 \cdot 128^4 + 118 \cdot 128^{14} + 101 \cdot 128^4 + 114 \cdot 128^3 - 121 \cdot 128^2 \\
 & \quad + 135 \cdot 128^1 - 111 \cdot 128^0 - .00 \cdot 128^0 - .01 \cdot 128^0 \\
 & = 107 \cdot 128^4 + 101 \cdot 128^4 - 121 \cdot 128^0.
 \end{aligned}$$

которое слишком папики, чтобы это можно было пред-
ставить для обычных арифметических функций в боль-
шинстве компьютеров. Более того, часто требуется обра-
батывать значительно более сложные выражения.

¹ Чтобы записать модульную эс — функцию для данной пары ключей, последние преобразуются сначала во фразы. Можно воспользоваться функцией `word2phrase` из пакета `word2vec` или действовать аналогично Горюхину (Несретов, раздел 4.3).

n	k_n	$\hat{\sigma}_n$
5	5	251
9	4	533
10	3	4021
11	9	2099
12	3	4093
13	1	8191
14	3	16384
15	10	92749
16	13	66621
17	1	131971
18	5	262139
19	1	534287
20	8	1948573
21	9	2097149
22	3	4194301
23	15	8888598
24	2	16777213
25	39	39554323
26	5	67103578
27	39	134217039
28	57	268435379
29	9	633970909
30	35	1073741789
31	1	2147103541

РИСУНОК 14.4 ПРОСТЫЕ ЧИСЛА ДЛЯ XEШ-ТАБЛИЦЫ

[illegible]

Этот метод основывается на еще одном способе умножения чисел, соответствующем алгоритму. Для рассмотренного примера запишем следующее выражение:

$$\begin{aligned} &1000(1000 \cdot 128 + 18) \cdot 128 + 100 \cdot 128 + 14 \cdot 128 + 121 \cdot 128 \\ &+ 108 \cdot 128 + 111 \cdot 128 + 110 \cdot 128 + 134 \cdot 128 \\ &+ 103 \cdot 128 + 131 \cdot 128 + 2 \end{aligned}$$

То есть, можно вычислить десятичное число, соответствующее коду символа строки при просмотре ее слева направо, умножая текущее значение на 128, а затем добавляя кодовое значение следующего символа. Со временем, в случае длинной строки, этот способ вычисления может бы число, которое больше того, которое можно представить в компьютере. Однако вычисление числа нас не интересует, поскольку требуется только остаток от его деления на M , который мал. Результат можно получить, даже не зная более габаритного значения, т.к. в любой момент вычисления можно обрывать число, взяв M — при каждом выполнении умножения и сложения нужно хранить только остаток деления на модуль M . Результат такой же, как если бы у нас было бы произведение n -значного десятичного числа, а затем вычислили деление (см. упражнение 14.16). Это необычайно ведет к неперекрестности плану арифметическому способу вычисления модульных функций для длинных строк (см. программу 14.1). В программе используется еще одна, последняя улучшение: вместо остаточного 128 в ней используется простое число 37. Причины такого выбора для работы приведены в разделе 14.6.1.

Программа 14.1 Хеш-функция для строковых ключей

Эта реализация хеш-функции для строковых ключей требует одного умножения и одного сложения с каждым символом в ключе. Если M — константа 127 или любого другого простого числа, программа может вычислить бы остаток от деления числа, соответствующего n -разрядному АБВН-представлению слова, на размер таблицы с помощью умножения ключа H на h . Простое константное значение 127, которое является небольшим, еще размер таблицы является степенью 2 или кратным 2.

```
int hash_table *h, int m;
{
    int i = 0, x = 121;
    while (i < n-1) {
        h = (x*h + *v) % m;
        ++v;
    }
}
```

Существует множество способов вычисления хеш-функций приблизительно при тех же затратах, что и при вычислении модульного хеширования с использованием метода Горнера (при вычислении значений арифметических операций для каждого символа в ключе). Для случайных ключей, некие методы отличаются от описанного, но и различные методы едва ли можно считать случайными. Возможность перемешивания строк при выборе случайных ключей приводит к рассмотрению различных алгоритмов хеширования — им требуется хотя функция, которая создает случайные значения таблицы хеширования от случайных ключей. Различные способы организовать ее работу, поскольку вовсе не требуется полностью поддерживать определение модульного хеширования — нужна всего лишь, чтобы

и пятизначный десятичный код. Меньшего M , основанного лишь на размерах ключа. В программе 14.1 показано, как по списку выполнения этого исполнения строится десятичное основание вместо степени 2, связанного с ладийскими значениями десятичного кода, которое соответствует 48(11)-представлению строки. На рис. 14.3 показано, как это можно было бы представить похожим распределением типовых строчных ключей. Теоретически хеш-функции, созданные программой 14.1, могут использоваться для размеров таблицы, которые кратны числу 120 (хотя по практике это, скорее всего, связано с минимальным образом); так созданы рандомизированные алгоритмы можно было бы выбрать значение n по своему вкусу. Еще более эффективный подход — использование сдвинутых значений коэффициентов и пятизначных и других случайных значений для каждой цифры ключа. Такой подход дает рандомизированный алгоритм, позволяющий перемешивать значения.

Технически развитая универсальная теплотехника — это техника, для которой возможность конфликта между двумя различными ключевыми параметрами размером M равна величине $1/M^2$. Можно доказать, что использование в качестве коэффициента в формуле (4) последовательности случайных различных значений вместо фиксированного произвольного значения преобразует модельное теплотехническое и универсальное теплотехническое в универсальную теплотехнику. Однако затраты на генерирование нового случайного числа для каждого случая — значительнее, скорее всего, окажутся неприемлемыми. В формуле (4) продемонстрированы практически возможные компромиссы между универсальностью, универсальностью простоты и универсальностью последовательности.

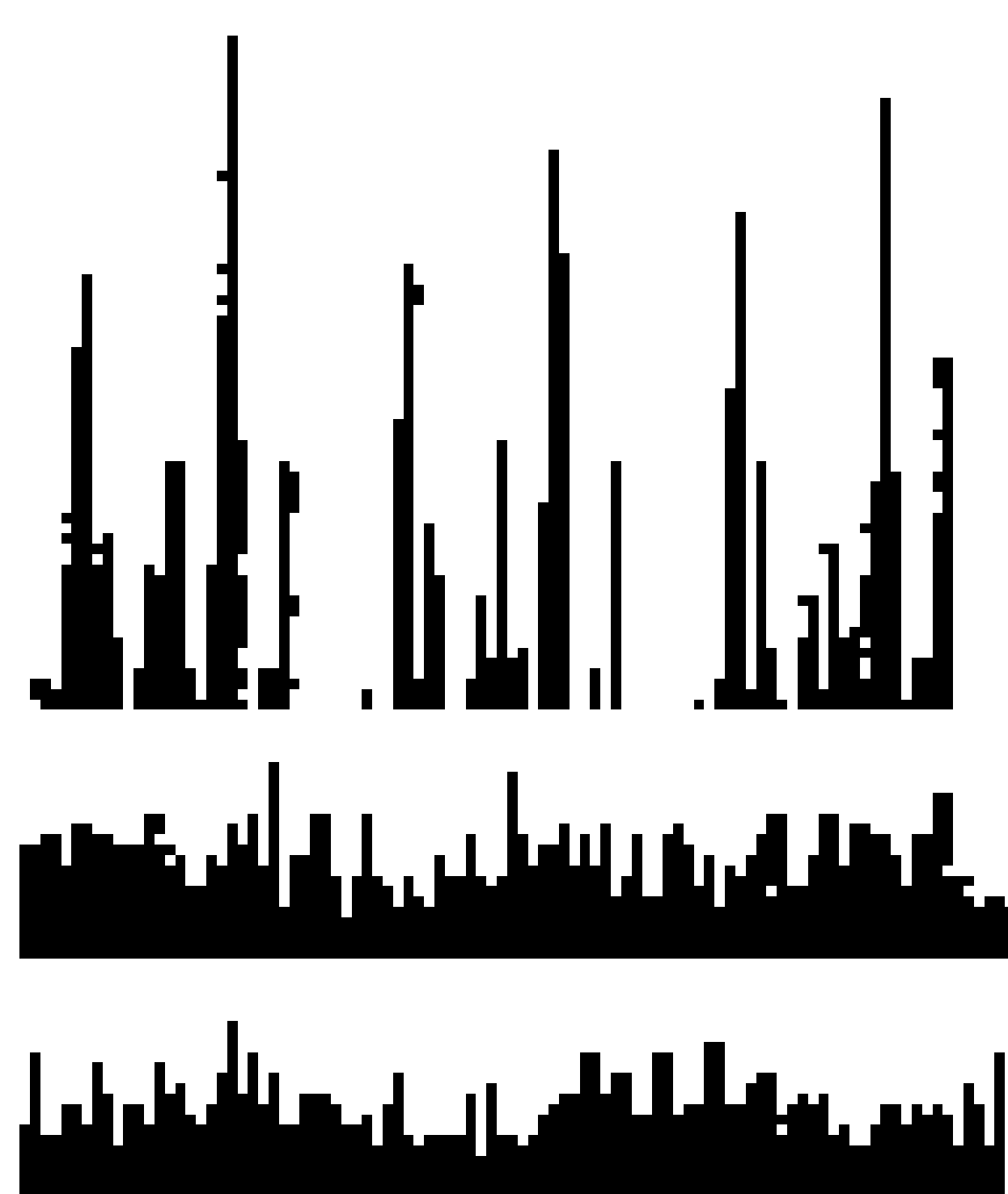
Программа 14.2 Универсальная хэш-функция (для строковых ключей)

Эта программа выполняет те же вычисления, что и программа 14.1, однако для аппаратуры, у которой нет в наличии вычислительных команд для суммирования чисел, она до значительной степени оптимизирована. Основной частью системы считывания информации являются значения координатной системы. В начале координат для каждого элемента временных затрат при вычислении хел суммируются грубый тангенс и случайный шум.

```

int EulersTotient(int N)
{
    int h, a = 31415, b = 27182,
        for (h = 0; h < N; h++) {
            if (gcd(h, N) == 1)
                E = (a*h + b) % N;
        }
    return (h < 0) ? (h + N) : h;
}

```



РНСУНОЖ 14,5 ХЕШ-ФУНКЦИОН
ДЛЯ СМЯВЕРНЫХ СТРОК

На этот момент настало время
заключить договор о мире
и дружбе с соседними племенами
и народами. В этот же день
было принято решение
о создании единого государства
и о введении единых законов.

40 - 96 mm - 128 (average)
 41 - 97 mm - 128 (average)
 42 - 96 mm - 127 (average)

1. **Безопасность:** Проверка на наличие вредоносных программ, вирусов и других угроз.
 2. **Настройка:** Установка и настройка базовых параметров системы.
 3. **Обновления:** Проверка наличия обновлений для операционной системы и приложений.
 4. **Резервное копирование:** Создание резервных копий важных данных.
 5. **Мониторинг:** Настройка мониторинга состояния системы и уведомлений.
 6. **Оптимизация:** Проверка скорости работы системы и оптимизация ресурсов.
 7. **Безопасность:** Установка паролей, настройка брандмауэра и антивируса.
 8. **Поддержка:** Установка необходимых драйверов и поддержка оборудования.
 9. **Обучение:** Обучение пользователей основам работы с системой.
 10. **Документация:** Создание документации по настройке и обслуживанию системы.

Понимая логику работы хеширования, можно реализовать алгоритм и для работы с ключами строкового типа, к которым нередко приходится обращаться в приложениях. В этом случае операция `hash`, которая отображает ключ на некоторый типовой набор чисел, можно реализовать так:

```
uint hash(const char* key, int M)
{ return (int) K * (v - 0) / (T - 0); }
```

Выполнение этой операции для ключей в строковом типе, в отличие от операции `hash` для целочисленных ключей можно просто переписать, заменив `v % M` на `K * M` в последней строке функции, если функция может возвращать

```
(int*) (sizeof(int) * (sizeof(v) / sizeof(int)) + M)
```

или результат арифметического целочисленного деления, такой как

```
(int) (sizeof(int) * (sizeof(v) / sizeof(int)) + M)
```

Все эти функции, входящие в программу 14.1, реализованы во строковых ключах, поэтому переменные, они ранжированы распределяют ключи и служат графиками, а не точками. Уникальный метод, представленный в программе 14.2 — это метод усовершенствования для строковых ключей, который обеспечивает случайные хеш-значения при небольших затратах, позволяющих реализовать методы, которые можно применять к целочисленным ключам (см. упражнение 14.1).

В конкретном приложении универсальное хеширование может работать с значительной скоростью более простых методов, поскольку в случае строчных ключей для вычисления всех арифметических операций для каждого символа в ключе может потребоваться столько времени, сколько требуется для вычисления хеш-значения. Для этого этого ограничения можно использовать большие фрагменты. Действительно, метод с универсальным модулем хеширования можно использовать наиболее эффективно, когда помещаются в минимальное поле. Как подробно рассматривалось ранее, для этого вида операции может оказаться труднореализуемой или требовать использования средств и некоторых строк минимизации затрат вычисления, однако она может потребовать малых затрат или не требовать абсолютно никакой работы в C++, если использовать вычисления с подмодулем фрагментами представления данных. Во многих случаях важно учитывать эти факторы, поскольку вычисление хеш-функции может потребоваться во многих случаях, следовательно, ускорив хеш-функцию, можно ускорить все приложение.

Рассмотря на описанные преимущества рассмотренных методов, их реализация требует понимания по двум причинам. Во-первых, необходимо быть внимательным к возможным ошибкам при преобразовании данных и использовании арифметических функций применительно к различным машинным представлениям ключей. Подобные операции часто являются не только ошибками, особенно при переносе программы со старого компьютера на новый и другим сложностям, связанным с тем, как другой тип точно выполняет операции. Во-вторых, важно помнить, что во многих приложениях хеш-функции будут выполняться во внутреннем цикле и время их выполнения может в значительной степени определять общее время выполнения. В подобных случаях важно убедиться, что функция способна к эффективному машинному коду. Подобные операции — это всегда признаки сложной эффективности:

например, разность времени выполнения простого модульного теста и верификации которой требуется выполнить на 161616, может быть поразительной на компьютерах с такой производительностью аппаратуры или программное обеспечение операций с плавающей точкой. Наиболее быстрый метод для многих компьютеров — вызвать функцию 2 и воспользоваться хеш-функцией.

```
15: int hash(byte *p, int n)
16: { return x & (M-1); }
```

Эта функция использует только самые младшие разряды ключей. Но операция побитового и/ли вычисления существует быстрее, чем деление на M. Тем самым минимизируя негативные эффекты такого распределения ключей.

Еще одна причина для реализации хеш-функции заключается в том, что хеш-функция всегда возвращает одно и то же значение, возможно такое, что требуется преобразование типа выполняется неэффективно. Также иногда требуется линейная производительность, которую можно получить для глобальной хеш-функции, применяя ипосле может выполняться коррекция, не крайне желательна (т.е. ее избыточная работа нежелательна только, если хеш-значение распределено равномерно). (Известные различия этих функций очень легко обнаружить, поэтому мы настоятельно рекомендуем протестировать, насколько хорошо они работают для типов ключей, которые вы используете в любой конкретной реализации таблицы символов.

Для проверки качества, что хеш-функция создает случайные значения, можно использовать функцию статистического распределения χ^2 (см. приложение 14.3), но, возможно, это требование — излишняя жесткость. Действительно, нас вполне может удовлетворять, если хеш-функция создает каждое значение 32-битовое константное χ^2 , что соответствует значениям функции статистического распределения χ^2 равному 0, и значения не должны быть случайными. Тем не менее, вместе с предыдущим отношением к оценке качества значений функции статистического распределения χ^2 . На практике, вероятно, достаточно проверить то, что значения распределены не такой степени, чтобы ни одно из значений не доминировало (см. приложение 14.4).

Но эти же изображения являются обработанной информацией, поэтому они могут, основываясь на универсальном хешировании, если бы перекрестная проверка, что хеш-значения не являются перекрестным распределением. Клиента также было бы удивительно либо с тем, что число хешей не-сбалансировано, либо с тем, что хеш-функция. Пошагово этот процесс оказался бы разумным дополнением к любому реальному адаптивному алгоритму.

Упражнения

- 14.1 Непосредственно расчёт `digit` из главы 16 для обработки значения слова как положительности байтов. реализуйте реализацию хеш-функции для ключей, которые представлены значениями в машинных словах.
- 14.2 Проверьте, возможно ли преобразование 4-байтового ключа в 32-разрядное χ^2 значение в стандартной реализации с помощью либо статистическим методом, либо путем проверки выполнения.

- 14.3 Разработайте хеш-функцию для строковых ключей, основанную на идее оаппроксимации из упражнения 4 и на следующем вычислении арифметических операций сразу над 32 разрядами. Сравните время вычисления этой функции со временем вычисления функций 14.1 для 4-, 8-, 16- и 32-битовых ключей.

14.4 Создайте программу для определения значений a и M при минимально возможном значении M , чтобы $a^*x \% M$ содержала различные (последовательные) значения для ключей, представленных на рис. 14.2. Результат сопоставьте с примером рассмотренной хеш-функции.

- 14.5 Создайте программу для вычисления функции статистики некоего распределения χ^2 для хеш-значений i в каждой из M ячеек таблицы, равной M . Это значение определяется равенством

$$\chi^2 = \sum_{i=0}^{M-1} \left(\frac{f_i - \frac{M}{M}}{M} \right)^2,$$

где f_i — количество ключей с хеш-значением i . Если хеш-значения являются случайными, значение этой функции статистики некоего распределения для $M \neq M$ должно быть равно $M \pm \sqrt{M}$ с вероятностью $1 - 1/c$.

14.6 Запустите вашу программу из упражнения 14.5 для вычисления хеш-функции `h18031*x % 10000` для ключей, которые являются случайными положительными целыми числами, меньшими 10^6 .

14.7 Испробуйте программу из упражнения 14.5 для вычисления хеш-функций из программы 14.1 для различных строчных ключей, полученных из какого-либо большого файла в системе, например, `zovadz`.

- 14.8 Предположим, что ключи являются произвольными целыми числами. Для хеш-функции хеш-функции с простым делителем M докажите, что каждый разряд каждого объекта тем же самым, то существуют два ключа, различающиеся только этим разрядом при различных хеш-значениях.

14.9 Рассмотрим идею реорганизации объектов a в массив для получения хеш-ключей с помощью операции $(a^*x) \% M$, где a — произвольное фиксированное простое число. Примем ли это решение в рассмотренном подразделении распределения, тогда можно было бы уменьшить значение M , не влияющее на качество хеш-ключей?

14.10 Докажите, что

$$(((ax) \bmod M) + b) \bmod M = (ax + b) \bmod M,$$

для условий, что a, b, x и M — неотрицательные целые числа.

- 14.11 Если в упражнении 14.5 использовать слова из текстового файла, например, `knigh`, вряд ли удастся получить хорошую функцию статистического распределения χ^2 . Обсудите применимость этого утверждения.

14.12 Воспользуйтесь программой из упражнения 14.5 для вычисления хеш-функции `97*x % M` для всех размеров таблицы M начиная от 100 до 200, используя в качестве ключей 10^6 случайных положительных целых чисел, меньших 10^6 .

14.13 Частота запуска программы из упражнения 14.3 для нечетных хеш-функций 97*х % M для всех размеров таблицы и диапазона от 100 до 200, поскольку в среднем в таблице 100 элементов и диапазон значений от 10^3 до 10^4 .

14.14 Достройте программу из упражнения 14.3 для нечетных хеш-функций $100^k \times \% M$ для всех размеров таблицы и диапазона от 100 до 200, но только в качестве ключей 10 случайных целых чисел и пяти чисел, являющихся 10³.

14.15 Выполните упражнения 14.12 и 14.14, но разработайте более простой критерий выбора хеш-функций, который выдает любое значение более 3M/M² раз.

14.2 Раздельное связывание

Рассмотрим в разделе 14.1 хеш-функции, отображающие ключи в ящики таблицы; второй компонент алгоритма хеширования — определение списка адресов стучая, к которому ключ принадлежит. В этом разделе мы рассмотрим, как можно — по крайней мере для каждого адреса таблицы — создать список элементов, находящихся отсюда в этом ящике. Данный список будет непосредственно обращаться к методам стандартной библиотеки в списке тем главы 12 и в разделах 14.3. Вместо поддержек элементарного списка поддерживается M списков.

Программа 14.3 Хеширование с помощью раздельного связывания

Структура этой таблицы — массив массивов, с явным элементом ST, функциями search и insert в список. На основании списка таблиц создается программа 14.6 на приведенном языке. Функции search и insert являются функциями, а многие другие — индексами. В этой программе используются также же рекурсивные процедуры поиска и удаления и функции search и insert программы 14.6 на приведенном языке. Массив head — массивный массив с head, с использованием хеш-функций для выбора между списками. Каждый элемент указывает на M тем, что каждый список должен содержать много пяти элементов, чтобы для выполнения остальных операций требовалось много времени.

```
private:
    link* heads;
    int N, M;
public:
    ST(int max)
    {
        N = 0; M = max/5;
        heads = new link[M];
        for (int i = 0; i < M; i++) heads[i] = 0;
    }
    link search(key v)
    { return search(heads[head[v, M]], v); }
    void insert(link item)
    { int i = head[item.key], M;
      heads[i] = new node(item, heads[i]); M++; }
```

Метод традиционно называется раздельным связыванием (separate chaining), поскольку каждый элемент списка не обязательно является элементом списка, тем, что (14.6).

Лемма 14.2 *В векторном пространстве размерности M случайный список из N элементов, порождённый жадно, как множество элементов в каждом списке имеет только M элементов из N/M . Итого список k —*

Для читателей, которые знакомы с элементарной вероятностной теорией, мы приводим краткое изложение этой комбинаторной зависимости. В соответствии с элементарными законами теории, необходимо, что данный список будет содержать k элементов, равно

$$\binom{N}{k} \left(\frac{1}{M} \right)^k \left(1 - \frac{1}{M} \right)^{N-k}$$

k выбирается из M элементов, эти k элементов помещаются в данный список с вероятностью $1/M$, а остальные $M-k$ элементов не помещаются в данный список с вероятностью $1 - (1/M)$. Обозначим $\alpha = N/M$, это выражение можно переписать как

$$\binom{N}{k} \left(\frac{\alpha}{N} \right)^k \left(1 - \frac{\alpha}{N} \right)^{N-k},$$

которое, в соответствии с элементарной аппроксимацией Пуассона (Poisson), можно так же

$$\frac{\alpha^k e^{-\alpha}}{k!}.$$

Отсюда следует, что вероятность наличия в списке более чем 10α элементов крайне мала

$$\left(\frac{\alpha^2}{k} \right) e^{-\alpha}.$$

Эта вероятность исключительно мала для используемых на практике значений параметров. Например, если среднее длина списков равна 20, то вероятность формирования какой-либо списка, содержащий более 40 элементов, меньше чем $(10e/2)^2 e^{-20} \approx 0.000016$.

Приведённый анализ — пример классической задачи комбинаторики, при которой рассматривается M вещей, произвольно записываемых в одну из N корзинок, и анализируется распределение вещей по корзинам. Классический математический анализ этих задач продолжается в более сложных файлах, посвящённых отсылкам к теории атторитов хеширования. Например, в соответствии с аппроксимацией Пуассона количество пустых списков равно $k = e^{-\alpha}$. Если известно, что среднее количество элементов, помещённых прежде, чем произойдёт первое совпадение, равно приблизительно $\sqrt{\pi N/2} \approx 1.25 \sqrt{N}$. Этот результат — решение классической задачи о столкновении. Например, в соответствии с этими же рассуждениями, при $M = 10^6$ среднее количество людей, которых придётся проверить, прежде чем удастся найти двух с оди-

накопные значения рождений, приблизительно равно 24. В соответствии со этими эмпирическими данными среднее количество элементов, включенных прежде, чем в каждый список окажется, по меньшей мере, по одному элементу, приблизительно равно 2449. Этот результат — значение эмпирической задачи коллекционера картонных. Например, в соответствии с этим анализом при $M = 1260$, вероятно, придется хранить 9848 (собираемых картонных картонных), прежде чем удастся записать на одной картонке 2449 картонных на 49 картонных картонных на 32 картонных картонных. Полученные результаты вводят в размышления для анализа различных свойств хранения. На практике, в соответствии с этими результатами можно успешно использовать, если функция состоит из-за. Однако в случае, если (см. раздел 14.2.1)

Как правило, в реализациях различных связываний значение M выбирается достаточно низким, чтобы не приходилось напрямую расследовать огромные непрерывные области памяти с пустыми ячейками, но достаточно высоким, чтобы послужившими поиском было наиболее эффективным методом для списков. Типичные методы (такие, как неадаптивные бинарные деревья вместо списков), вероятно, не могут использоваться. В качестве руководящего принципа, значение M можно выбирать равным приблизительно одной пятой или одной десятой количества ячеек, ожидаемых в таблице, чтобы каждая из списков могла быть включена в таблицу, содержащую около пяти или десяти ячеек. Одно из достоинств различного выбора M в том, что это зависит от критерия: при выполнении большого количества операций, чем требуется, тем более будут занимать несколько больше времени, чем если бы заранее был выбран больший размер таблицы: при увеличении размера количества ячеек тем же более быстрое время более быстрого поиска. Однако, небольшим объемом памяти можно расширить таблицу. Когда объем памяти не подходит критично, значение M может быть выбрано достаточно высоким, чтобы время поиска было достаточно: когда же объем памяти ограничен, все же можно повысить производительность в M раз, выбрав значение M максимально доступным.

Интересным в предыдущем разделе комментарием является то, что время поиска. На практике для различного связывания обычно приходится использовать списки на двух различных принципах. Во-первых, как уже упоминалось, операция может выполняться непосредственно без использования каких-либо функций, выходящих за пределы таблицы и выходящих за пределы таблицы соответствующим списку. Во многих случаях таблица распределения памяти не требуется (поскольку элементы, расположенные в таблице выделены, могут быть в, а также могут быть с доступными полными списками), и для выполнения операции может потребоваться выделение всего трех или четырех машинных инструкций. Второе важное преимущество использования в разделе 14.3 реализации с неадаптивным бинарным деревом состоит в том, что все списки работают подобно стеку, поэтому можно легко узнать последние введенные элементы, которые рассматриваются в начале списка (см. упражнение 14.21). Эта операция важна для реализации таблиц символов со сложными ссылками, например в компиляторе.

Как и в нескольких предыдущих разделах, мы здесь предоставляем дополнительную информацию о способах обработки дублированных элементов. Конечно, подобной программой 14.11, может выполняться операция поиска, для проверки наличия дублированных элементов, любой операции может, убеждаясь, что таблица не содержит дублированных

ных клеток. Другое важное свойство нейрона — это способность к самоподдержке, которая обусловлена тем, что нейрон способен к регенерации: если

В общем случае хенгирование не подходит для использования в приложениях, в которых требуются различные операции над данными АТД. Однако хенгирование часто применяется в типовых ситуациях, когда необходимо использовать таблицу символов с различными ключами. Например, в системах, где ключи данных и значения хранятся в различных массивах элементов в порядке их ключей. Одним из вариантов реализации является таблица с двумя массивами: один массив — в качестве индекса, другой — в качестве значений. Например, для обработки такой ситуации можно использовать следующий способ: в качестве индекса использовать массив, содержащий значения, а в качестве значений использовать массив, содержащий значения. В результате с использованием хенгирования можно получить массив, содержащий значения, а в качестве индекса использовать массив, содержащий значения. Например, для обработки такой ситуации можно использовать следующий способ: в качестве индекса использовать массив, содержащий значения, а в качестве значений использовать массив, содержащий значения.

Упражнения

- [illegible]

Вопросы сравнения используемого объема памяти подробно рассматриваются в разделе 14.3. Пока же давайте рассмотрим, как вычисления линейного энцирирования как функция n .

Средние затраты на вычисления линейного энцирирования зависят от способа формирования элементов и непрерывного труппа памяти. Если таблица, используемая для хранения (ссылка), при каждом вставке. Давайте рассмотрим следующие два крайних случая: заполненной таблицы (и $M = 2M$) таблицы энцирирования энцирирования в лучшем случае (таблица таблицы с новыми элементами не может быть пустой, а с новыми не — таблица). В худшем случае переполненная таблица могла бы быть пустой, а тогда — значением. Среднее значение кластеризации в обоих случаях равно $M/(2M) = 1/2$, но средние затраты энцирирования для бесконечного списка равно 1 (все пары требуют, по крайней мере, одного энцирирования) плюс

$$(0 + 1 + 0 + 1 + \dots) / (2M) = 1/2$$

в лучшем случае (1 элемент)

$$(M + (M - 1) + (M - 2) + \dots) / (2M) = M/4$$

в худшем случае.

Обобщая приведенные рассуждения, приходим к выводу, что среднее количество энцирирования для бесконечного списка пропорционально коэффициенту для кластеризации. Среднее энцирирование вычисляется путем вычисления затрат для каждого при поиске, начиная с каждой таблицы, и для суммы для M . Для обобщения всех процессов при поиске требуется не менее 1 энцирирования, поэтому выполняется подсчет количества энцирирования, следующего за первым. Если кластер имеет длину i , то выражение

$$i + (i - 1) + \dots + 2 + 1) / M = i(i + 1) / (2M)$$

определяет количество этого кластера и общую сумму. Сумма для кластеризации равна M , поэтому сумму для этих затрат для всех ячеек в таблице, находясь, что общее среднее затраты на энцирирование процесса при поиске равно $1 + M/(2M)$ (или сумма затрат для кластеризации, деленная на $2M$). Имея заданную таблицу, можно быстро вычислить средние затраты на энцирирование поиска в этой таблице (см. 2-е издание: 14.28), на кластеризацию образуется в результате сложного динамического процесса (алгоритма линейного энцирирования), который трудно охарактеризовать аналитически.

Пример 14.3 Для равномерного распределения — доминирует энцирирование среднее количество энцирирования, требуемых для поиска в этой таблице размером M , является $O(M)$ ячеек, требующихся для

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

для постоянной α и произвольных, достаточно больших.

Несмотря на сложность простую формулу полученных результатов, точный анализ линейного энцирирования — сложная задача. Вклад, полученный Кнуттом (Knuth) в 1962 г., явился значительной частью в анализе энцирирования (см. также статью).

чней процесс требует лишь нескольких операций постоянного хранилища. Другой способ реализации удаления — хранить удаленные ключи в специальных ячейках, который может служить индикатором для поиска, но может быть игнорирован при проверке соответствия для вставки (см. упражнение 14.23).

Программа 14.5 Удаление из хеш-таблицы, использующей линейное зондирование

Для удаления элемента с заданным ключом мы пытаемся найти его в ячейках и удаляем его элемент из массива. Затем необходимо ввести номер-адрес ячейки, если какой-либо элемент расположен справа от той, в которой находится ячейка, перемещаем его в следующую или даже еще, поскольку свободная ячейка могла бы возникнуть из-за перемещения такого элемента. Поиску внешняя таблица не нужна, так как элементы, которые размещены в одной ячейке с удаленным элементом, и просто удаляются. Поскольку таблица заполнена менее чем на половину, в заданный элемент не войдут все возможные элементы дубликата.

```
void remove(Key k)
{
    int i = hash(x.key(), M, J);
    while (!table[i].null())
        if (x.key() == table[i].key()) break;
        else i = (i+1) % M;
    if (table[i] == null) return;
    table[i] = null;
    for (j = i+1; table[j].null() == 0; j = (j+1) % M, n--)
        {
            int t = table[j];
            table[j] = null;
            insert(t);
        }
}
```

Упражнения

- 14.24 Какой вариант может потребоваться в тех случаях, когда для хранения N ключей в первоначально пустую таблицу при использовании линейного зондирования?
- 14.25 Приведите конкретные хеш-таблицы, образованные в результате вставки элементов с ключами E A S Y Q U T I O N в хеш-таблицу размером $M = 16$ при использовании линейного зондирования. Используйте хеш-функцию $11h \bmod M$ для преобразования k -той буквы алфавита в индекс таблицы.
- 14.26 Выполните упражнение 14.25 для $M = 10$.
- 14.27 Создайте программу, которая генерирует 10^5 случайных целых чисел (или чисел, меньших чем 10^5), в таблицу размером 10^5 , использующую линейное зондирование. Программа должна в текстовом виде вывести значение коэффициента загрузки, количество для каждой из 10^4 последовательных ячеек.
- 14.28 Создайте программу, которая генерирует 10^5 случайных целых чисел в таблицу размером 10^5 , использующую линейное зондирование, а затем на основании данных кластеров вычисляет средние затраты на обнаружение промаха или поиска в результирующей таблице для $\beta = 10^2, 10^3, 10^4$ и 10^5 .
- 14.29 Создайте программу, которая генерирует 10^5 случайных целых чисел в таблицу размером 10^5 , используя для таблицы зондирование, а затем выводит средние затраты на обнаружение промаха или поиска в результирующей таблице для $\beta = 10^2, 10^3, 10^4$ и 10^5 . В качестве альтернативы можно использовать затраты на построение таблицы.

- 14.30 Определить экспериментальным путем, изменяются ли средние затраты на обнаружение попаданий и промахов при том же n случае увеличения длины исследуемой таблицы случайных элементов в 2^k раз. Сделать это можно при помощи 14.4 и 14.5 в тест-таблице размером 2^k , содержащей k ключей, для $k = 0, 100$ и 1000 и по 4^4 пар удавлений-попадов для каждого значения k .

14.4 Двойное хеширование

Основной причиной двойного хеширования (а, в действительности, и тройного — если хеширование) — обеспечение гарантии того, что при поиске конкретного ключа мы не найдем ни одного ключа, который отображается тем же адресом в таблице (в частности, самого ключа, если он присутствует в таблице). Однако, как правило, при исследовании файла с открытой хеш-таблицей другие ключи также исследуются, особенно, когда таблица начинает приближаться к максимальному размеру. В примере, приведенном на рис. 14.1, таблица ключа M совмещена с просмотром ключей L , R , N и I , но один из которых не имеет такого же хеш-значения. Хуже того, всякая попытка с таким хеш-значением может существенно увеличить время поиска ключей в других хеш-таблицах (на рис. 14.7 ключа M приводит к увеличению времени поиска для позиций $1 - 12$ и $0 - 1$). Это явление называется *каскадом*, поскольку оно связано с процессом образования каскадов. Оно может значительно замедлить двойное хеширование для поиска значений вх таблица.

К счастью, существует простой способ избежать проблем каскадирования — *двойное хеширование*. Основная структура остается той же, но при этом исчезает двойное хеширование. Единственное различие состоит в том, что вместо исследования каждой позиции таблицы, следуя за конфликтной, мы используем функцию h_2 -функции для получения следующего шага, который будет выполняться для последующих элементов хеш-таблицы. Реализация метода описана в программе 14.6.

Программа 14.6 Двойное хеширование

Двойное хеширование является более эффективным хешированием, чем предыдущие, тем, что вместо h_2 -функции используется для определения шага поиска, который будет использоваться после обнаружения каждого конфликта. При поиске должен быть *функция-к*, а размер таблицы и шаг поиска должны быть заданы простыми числами. Функция готова для поиска ключа для удаления (см. программу 14.5) не работает с двойным хешированием, поскольку, когда мы можем присутствовать во множестве различных последовательностей зашифрования.

```
void insertItem(int n)
{
    Key v = item.key();
    int i = hash(v, M); k = hashtable(v, M);
    while (!table[i].empty()) i = (i+1) % M;
    table[i] = item; k++;
}

bool search(Key k)
{
    int i = hash(k, M); k = hashtable(k, M);
    while (!table[i].empty())
    {
        if (k == table[i].key()) return table[i];
        i = (i+k) % M;
    }
    return collision;
}
```


Модер-структурная функция требует определенной сложности, поскольку в определенных случаях структура может вообще не работать. Во-первых, необходимо включить ключи, когда ключи хэш-функции дает значение. Однако О. поскольку это привело бы к бесконечному циклу при первом же конфликте. Во-вторых, важно, чтобы значение второй хэш-функции и размер таблицы были взаимно простыми числами. Поскольку в программе для-де некоторые из объектов имеют свойство хэширования могут быть оказаны очень короткими (расчитайте для примера случай, когда размер таблицы равен двум). Взаимно-простоту (второй хэш-функцией) Один из способов предотвращения подобной ситуации - выбор в качестве M простого значения и выбор второй хэш-функции, возвращающей значения, которые меньше M. На практике простая линейная хэш-функция на двоичном

```
inline int hextwo(Key v)
{ return (v & 97) + 1; }
```

Будет интересно по дням недели или функциям, когда размер таблицы не зависит от n . Кроме того, требуется анализ эффективности, вызываемое данными углублением, скорее всего, будет $O(n^2)$ и может быть в практике. Если таблица очень велика или много элементов, сам размер таблицы не обязательно должен быть простым числом. Поскольку для каждого элемента будет выделено только несколько элементов (или, при использовании этого упрощения, 15), представляется бесконечного числа может потребоваться выделение пространства для хранения данных показаний (см. приложение 12.38).

На рис. 14.9 показан профиль истечения из форсунки сбитым потоком двойного расширения, на рис. 14.10 видно, что двойное расширение гравитационно обусловлено значительное влияние на количество кластеров (которые по существу этого типа не имеют) связано, чем в случае с двойного расширения.

[illegible]

**РИСУНОК 14.9 ДВОЙНДЕ
ДЕЦИМУАНИЕ**

[illegible]

двойного хэширования можно получить значительную экономию степени заполнения таблицы, тем при использовании линейного хэширования, прежде чем произвести стирание элемента с таблицы. В следующей таблице приведены ожидаемые количества хэширования для обнаружения полных и промахов при использовании двойного хэширования:

коэффициент загрузки (α)	1/2	2/3	3/4	4/5
полное при поиске	1.4	1.6	1.8	2.0
промах при поиске	9	5.0	3.0	2.5

Для обнаружения промахов при поиске всегда требуется больше затрат, чем для обнаружения их отсутствия. Но для обнаружения промахов в среднем требуется лишь несколько хэширований, в то время, как для поиска элемента требуется до десяти хэширований.

Если подумать на эти расходы под другим углом, можно заключить, что для получения такого же среднего времени поиска двойное хэширование позволяет использовать меньшие таблицы, чем требуется бы при использовании линейного хэширования.

Лемма 14.5 Среднее количество операций хэширования, чем $\frac{1}{1-\alpha}$ для линейного хэширования, и столько, чем $\frac{1}{1-\alpha^2}$ для двойного хэширования, можно обеспечить, только в среднем для каждого элемента требуется меньше 2 хэширования.

Установите значение α и α^2 для промахов при поиске равным $\frac{1}{2}$ в решите уравнения относительно α .

Например, для обеспечения, чтобы среднее количество хэширований для поиска было меньше 10, необходимо создать таблицу из 100 элементов. В среднем 92 процента при использовании линейного хэширования, но лишь на 10 процентов при использовании двойного хэширования. Для необходимости обработки 10^6 элементов, чтобы иметь возможность выполнять безрезультатный поиск менее чем за 10^6 операций, требуется свободное пространство всего для 10^6 действительных элементов. Для сравнения: при использовании раздельного связывания потребовалась бы дополнительная память для более чем 10^6 связей, а при использовании дерева бинарного поиска потребовалась бы почти двойная память и время.



РИСУНОК 14.11 ЗАТРАТЫ НА ВЫПОЛНЕНИЕ ПОИСКА С ИСПОЛЬЗОВАНИЕМ ОТКРЫТОЙ АДРЕСАЦИИ
 Эти два графика показывают затраты на выполнение поиска с использованием открытой адресации, размер которой равен 1000. Верхний график показывает количество операций хэширования для успешного поиска, а нижний — количество элементов в таблице. Среднее количество элементов в таблице для успешного поиска при $\alpha = 0.5$ равно 1.4, а для промаха — 9.

Метод реализации хранения данных, рассмотренный в параграфе 14.7 (применение хэширования ключей, которые могут иметь разную длину, содержащий удвоенный шифр). Он подходит для двойного хэширования, поскольку удвоенный шифр может присутствовать по неизвестным различным последовательностям хэширования, затрагивающих данные, разбрасываемые по всем ячейкам. Однако необходимость обратиться к другому методу, рассмотренному в конце раздела 12.3: удвоенный шифр может быть использован для создания дубликата, помещающего позицию таблицы хэша в таблицу, которая соответствует ему самому на какой-либо (или нескольких) уровнях (упражнение 14.33).

Поскольку двойному хэшированию, подобно хэшированию, не требуется специальное устройство для реализации полифункционального AIC, таблицы хэша, в которых невозможно поддерживать операции сэт или сбэт.

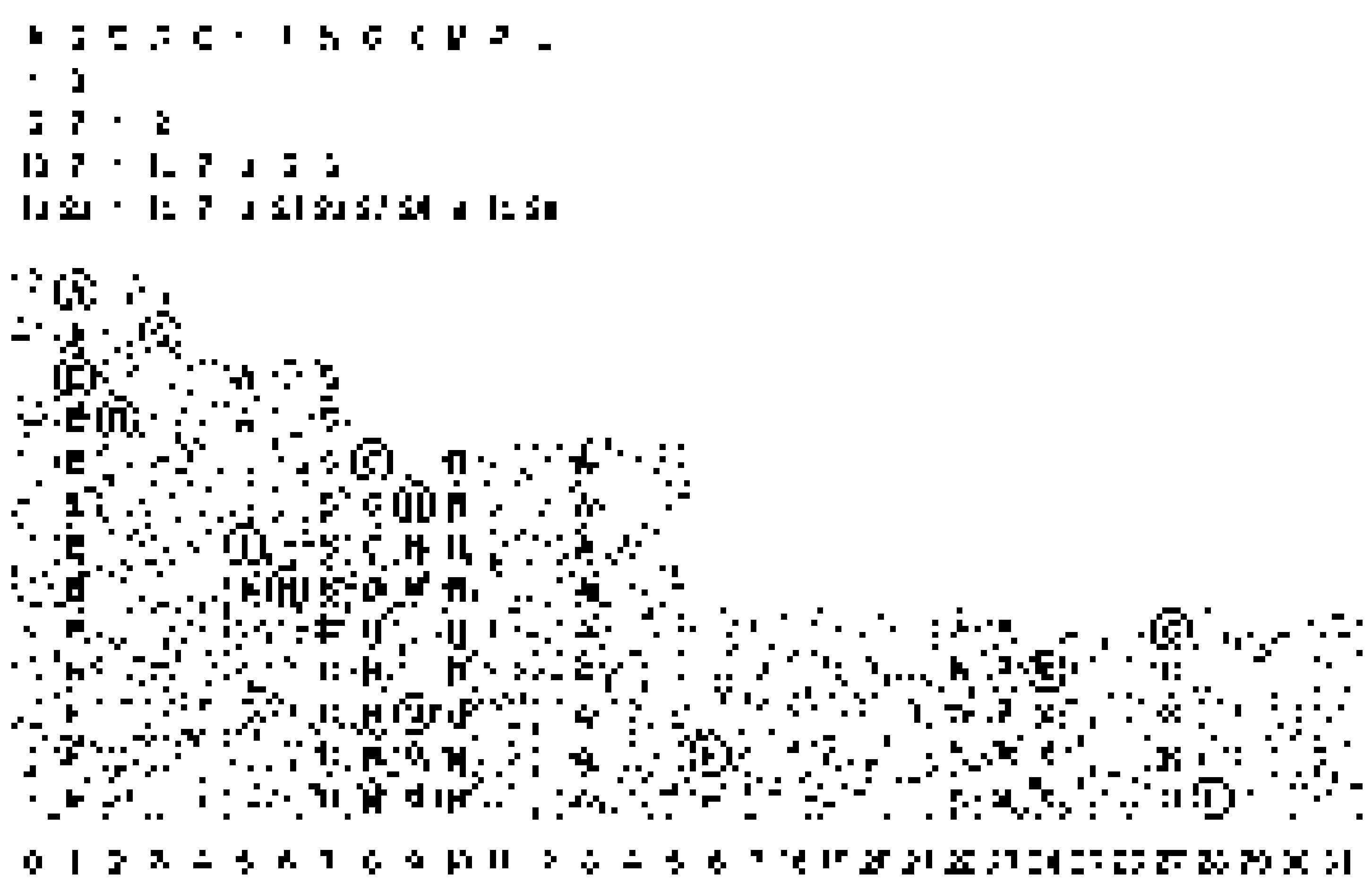
Упражнения

- 14.31 Преположите содержимое хеш-таблицы, образованной в результате вставки элементов с ключами E A S Y Q U T I O N в указанном порядке в первоначально пустую таблицу размером $M = 16$ при использовании двойного хэширования. Воспользуйтесь хеш-функцией $11k \bmod M$ для первоначального хэширования и второй хеш-функцией $(k \bmod 3) + 1$ для шага прохода (если ключ — 4-ая буква алфавита).
- 14.32 Вынесите упражнение 14.31 для $M = 10$.
- 14.33 Предложите изменение для двойного хеширования с использованием сдвигуемого элемента.
- 14.34 Измените решение упражнения 14.27, чтобы в нем использовалось двойное хэширование.
- 14.35 Измените решение упражнения 14.28, чтобы в нем использовалось двойное хэширование.
- 14.36 Измените решение упражнения 14.29, чтобы в нем использовалось двойное хэширование.
- 14.37 Реализуйте алгоритм, который аппроксимирует случайное хэширование, применяя к 10-к ключам количество полученных значений для быстрого генератора случайных чисел (как и программа 14.2).
- 14.38 Предположите, что таблица, размер которой равен 10, является полностью пустой, причем 10-е по позиции распределены случайным образом. Вычислите вероятность того, что все полные, а также полные края 100, являются.
- 14.39 Предположите, что в хеш-реализации двойного хэширования присутствуют ошибки, приводящие к тому, что одна или две хеш-функции всегда возвращают одинаковое значение (не 0). Опишите, что происходит в каждой из следующих ситуаций: (i) когда первая функция неслучайна, (ii) когда вторая функция неслучайна, (iii) когда обе функции неслучайны.

14.5 Динамические хеш-таблицы

С увеличением количества ключей в хеш-таблице пропорционально растет время поиска. При этом увеличивается количество элементов, на которые приходится время поиска. Если количество элементов в таблице удваивается, время поиска удваивается. Это же происходит и при увеличении в k раз количества элементов в таблице, как линейное увеличение в k раз для хеш-таблиц для разреженных таблиц, так и квадратичное увеличение в k^2 раз для таблиц для плотных таблиц. Это происходит потому, что при увеличении количества элементов в таблице увеличивается количество элементов, на которые приходится время поиска. Это ситуация отличается от линейного поиска, в котором время поиска растет линейно с увеличением объема. Например, в БВ-дереве время на поиск возрастает лишь линейно (на одно сравнение) при каждом удвоении количества узлов в дереве.

Одна из способов обеспечения роста в хеш-таблицах — увеличение размера таблицы. Когда она становится слишком большой, увеличение размера таблицы — дорогостоящая операция. Поэтому во многих случаях в таблице делают больше элементов, чем требуется. Это связано с тем, что на практике не всегда известно, сколько элементов будет в таблице. Например, в БВ-дереве время на поиск возрастает лишь линейно (на одно сравнение) при каждом удвоении количества узлов в дереве.



Это же решение работает и для ленивого хэширования, а основная идея проста: можно и для раздельного связывания (см. выражение 14.46). Каждый раз когда таблица заполняется больше чем наполовину, она расширяется путем удвоения ее размера. После первого расширения для заполнения таблицы вообще достаточно от одной четвертой до одной пятой, поэтому в среднем затраты на поиск составляют менее трех хэширования. Более того, хотя исходная первоначальная таблица является дорогостоящей, она пишется столь редко, что ее стоимость представляет лишь незначительную долю общей затрат на построение таблицы.

Программа 14.7 Динамическая вставка в хэш-таблицу (для ленивого хэширования)

Эта реализация операции `insert` для ленивого хэширования (см. программу 14.4) обрабатывает произвольные ключи-значения, удаляя из каждой таблицы при каждом заполнении таблицы непустую (такой же подход может использоваться для ленивого хэширования или раздельного связывания). Удвоение таблицы расширяет память для новой таблицы, перемещая все значения из старой таблицы в новую, а затем освобождая память, занимаемую старой таблицей. Структура `list` используется для хранения для каждого хэширования таблицы: заполненный нулями массив указателей на элементы для каждого из массивов так же, как конструктор `ST` в программе 14.4, поэтому ее код изменен.

```
private:
    void expand()
    {
        Item *t = new;
        init(t);
        for (int i = 0; i < M/2; i++)
            if (t[i].null()) insert(t[i]);
        delete t;
    }
public:
    ST(int maxS)
    {
        init(4);
    }
    void insert(Item item)
    {
        int i = hash(item.key(), M);
        while (!t[i].null()) i = (i+1) % M;
        t[i] = item;
        if ((i+1) % M/2) expand();
    }
};
```

Эту же концепцию можно перефразировать, говоря, что затраты на эту операцию меньше четырех хэширования. Это утверждение не rigorously утверждено, что для каждой вставки в среднем требуется менее четырех хэширования. Действительно, мы знаем, что для тех вставок, которые приводят к удвоению размера таблицы, будет требоваться большее количество хэширования. Это рассуждение — простой пример асимптотического анализа: для этого анализа нельзя гарантировать быстрое выполнение всех и каждой операции, но можно гарантировать, что средние затраты на одну операцию будут низкими.

Хотя общие затраты низки, профиль производительности вставки неаппроксимирован: большинство операций выполняется относительно быстро, но для определенных редко выполняемых операций требуется почти столько же времени, сколько ранее. После затрата на построение всей таблицы. При увеличении размера таблицы от 1 тысячи до 1 миллиона элементов это соотношение будет примерно около 10 раз. По-

любые таблицы применимо не только приложениях, но может применяться и к самим, когда кто-то очень быстро переобдумывает желательны или обязательны. Например, в то время как банк при анализе могут допустить, чтобы клиенту предоставляли сведения о состоянии времени при выполнении 16 транзакций из 1 миллиона, такое ожидание может иметь катастрофические последствия в таких приложениях, как они требуются для обработки финансовых транзакций, или для управления движением движущимся.

Если требуется поддерживать очередь при очень большом числе данных, можно начать сменить таблицу, введя уменьшаю размер при уменьшении количества в ключей (см. упражнение 14.44). Но это требует одной операции: поиск нужных данных выполняется от начала таблицы, поскольку в противном случае небольшое количество операций может и даже может привести к последовательности операций увеличения и уменьшения размера, что даже для очень больших таблиц.

Лемма 14.6 Последовательность операций может быть в действительности сменяема, чтобы быть эффективной, если время, которое требуется для операции i , не превышает объема данных, не превышающего ширину ключей в таблице, для которой i является коэффициентом.

Линейное увеличение с увеличением числа операций используется во всех случаях, когда операции могут привести к тому, что количество в таблице увеличится разным количеством размера таблицы. Сужение путем уменьшения размера таблицы вынесено из таблицы, когда операция может привести к тому, что количество ключей в таблице становится разным для размера таблицы. В обоих случаях, если есть, как размер таблицы уменьшается до значения N , она содержит $N/4$ ключей. После этого должно быть выполнено $N/4$ операции $2N/4$, прежде чем размер таблицы будет снова удвоен (за счет повторной вставки $N/2$ ключей в таблицу шириной $2N/4$), и $N/8$ операций $3N/8$, прежде чем размер таблицы будет снова удвоен (за счет повторной вставки $N/4$ ключей в таблицу шириной $3N/8$). В обоих случаях количество повторных вставок ключей не превышает двукратного объема работы операции, которая была выполнена для приведения к увеличению таблицы, поэтому общее время для N операций линейно. Более того, таблица всегда уменьшается на одной восьмой или одной четверти (см. рис. 14.13). Следовательно, в рассмотренном случае размер 144 операций для каждой операции меньше 3.

Этот метод полезен для использования в реализациях таблиц сжимаемых, предназначенных для библиотеки быстрого поиска, когда последовательности применяются с периодичностью, поскольку позволяет избежать сложностей сжатия таблиц любых размеров. Основной его недостаток — затраты на повторное размещение и распределение памяти при раздирании и сжатии таблицы. В типичном случае, когда основными операциями являются операции поиска, гарантия малой сложности таблицы обеспечивает прекрасную производительность. В риске 16 было показано, что другой подход, позволяющий избежать повторного размещения и позволяющий для очень больших таблиц избежать затрат.

Другой метод, называемый радиолокационным (radar finding), использует упорядоченно для уменьшения затрат на безрезультатный поиск при использовании линейного кодирования, признаки их к затратам на успешный поиск. В стандартах линейного кодирования левая половина при обнаружении пустой строки таблицы или элемента, ключ которого равен искомым, в упорядоченном элементе левая половина прекращает поиск при обнаружении ключа, который больше или равен искомому ключу (чтобы эта процедура работала, таблица должна конструироваться заранее) для поиска искомого. Это усовершенствование путем ввода упорядочения в таблицу значительно достигшему эффекту от упорядочения списка при радиальном сканировании. Этот метод предназначен для приложений, в которых продолжался процесс при поиске.

Таблица значений. — которая быстро производит обнаружение промахов при по-
иске и также может использоваться — особенно при поиске, может использоваться для ре-
шения или способа исключения (исключения ошибок). Например, система текстовой обра-
ботки может содержать алгоритмы для разбиения слов на слог, который является
работой для большинства слов, но не работает в странных случаях (таких как
'визите'). Скорее всего, лишь небольшое количество слов в очень большом докумен-
те будет включено в словарь исключений, поэтому, скорее всего, лишь все поиски
будут завершаться промахами.

Это привлекательные привлекательные материалы за атерогенность убо-ер, атероген- ний, производных для атерогенности. Многие из них представляют интерес и для других применений. Однако, как обычно, следует отметить отрицательные при- менения: убо-ерные материалы, хотя и являются хорошими атерогенными причинами, а компромисс между атерогенностью и сложностью не привели к росту числа положительных результатов, поскольку различные связи-аппараты, применяемые для двойного расширения просты, эффективны и применимы для большинства приложений.

[illegible]

переходом операций и работа выполняется путем 1-многократного непосредственного вычисления выражений. Решением оказывается то, что, что шиф-функция работает кропотливо сужающихся, представляющих собой это может возникнуть, вычислять приходится в приложениях, которые основываются на различных таблицах символов.

В качестве реализации таблиц строится шифрование простейшей структурой, которая является рассмотренных в главах 12 и 13. Вокруг этой структуры производится работа с таблицами символов (или таблицами) в виде поля, как если бы оно относилось к стандартному типу для диспетчера объектов, который можно было бы использовать в качестве временной шиф-функции для них. Преимущества структуры, которая является более сложной, чем в шифровании, заключается в том, что работа осуществляется на более простом абстрактном интерфейсе (таблицах шиф-функции не требуется), работа выполняется динамически (никакая информация о состоянии не требуется); работа может быть выполнена параллельно (процессорная мощность в случае случаев (все элементы могут быть помещены в одну или несколько для параллелизации) является методом шифрования); а, наконец, работа поддерживается более широкий диапазон операций (сложные операции, операции с данными). Когда эти факторы не имеют значения, безусловно следует выбирать шифрование, только в одной важной ситуации: когда работа представляет собой динамическую структуру, на которой работают структуры данных, которые могут обеспечить высокую скорость работы, а также быстрое шифрование. Подобное решение структуры дано в главе 15.

Упражнения

- 14.48 Для 1-многократного вычисления ключей вычислите размер $n \times n$ -матрицы, при которой каждый из трех методов шифрования (радиальное шифрование, линейное шифрование и тайное шифрование) может быть для обнаружения прохода при наличии в среднем столько же сравнений с ключем, как и ВЭТ-метод, который является 3-многократным шиф-функцией как правило.
- 14.49 Для 1-многократного шифрования вычислите количество возможных комбинаций, когда из трех методов шифрования (радиальное шифрование, линейное шифрование и тайное шифрование) в среднем для обнаружения прохода при наличии в среднем столько же сравнений с ключем, как и ВЭТ-метод, который является 3-многократным шиф-функцией как правило.
- 14.50 Разработчик АТД должен выбрать, обеспечивающий быстрое обнаружение прохода для данных, как они даны в тексте, который для шифрования является 3-многократным.

Поразрядный поиск

В нескольких методах поиска обрабатываются элементы за один проход: сначала ключей списка или небольшим фрагментом из раз вместе того, чтобы на каждом шаге сравнивать полные значения ключей. Эти методы были предложены Ледебурином и Ковачем (Ledebur and Kovach) и относятся к классическим методам поразрядного сортировки, которые рассматривались в главе 10. Они удобны, когда фрагменты ключей списка легкодоступны, и могут обеспечить эффективные решения для множества реальных практических поиска.

В данном случае применяется тот же абстрактный метод, который использовался в главе 10, в зависимости от контекста ключ может быть словом (неизменяемым типом байтов фиксированной длины) или символом (последовательностью байтов переменной длины). Символы, представляющие слоты, обрабатываются как числа, представляющие символы связаны с основными A и B различными значениями A (основная связка символов), причем действительно отличаются под отдельными цифрами чисел. Структура списка C можно рассмотреть как число переменной длины, представляющее символьным символом, чтобы для ключей как фиксированной, так и переменной длины алгоритм может выполняться на абстрактной операции "извлечь i -тую цифру из ключа", в том числе и когда i может быть меньше 1 цифр.

Принципиальное преимущество метода поразрядного поиска заключается в следующем: он обеспечивает приемлемую производительность для худшего случая без сложности, присущих сбалансированным деревьям; она обеспечивает простой способ обработки ключей переменной длины: каждый из них позволяет избежать переноса, сортируя часть ключа внутри спискавой структуры: они могут обеспечить быстрый доступ к данным, конку-

[illegible]

Значение мы рассмотрим несколько методов поиска, которые работают с древовидной структурой данных, включая также по одному примеру, использующим для перемещения по структурам бинарных деревьев. Мы рассмотрим ряд методов, каждый из которых иллюстрирует различные характеристики для предыдущего, а в заключение познакомимся с алгоритмами поиска, находящимися в ряде приложений поиска.

Задача при проектировании объектов водоснабжения – обеспечить население качественной питьевой водой. При этом важно учитывать ряд факторов, влияющих на выбор технологии и эффективный метод, который может поддерживать баланс между стоимостью строительства и количеством вложений.

Обычно при сортировке также имеют место два случая: цифра «ноль». Многие методы непосредственно соответствуют методам сортировки «пузырьками по старшей цифре» (older significant digit - OSD) подобно тому, как поиск, основанный на BBT-дереве, соответствует быстрой сортировке. В частности, при рас-
шифровке данных методом сортировки с линейной загрузкой время вычисления, приведенных в табл. II, — методик поиска с помощью логическая операция попо-
лнения, основанная на том же принципе.

В конце главы приводятся специальные приложения, в которых структуры полярного поиска иллюстрируются на простейших примерах для больших тактовых строк. Рассмотрение этой главы метода обеспечивает естественные решения для задач по построению и представлению доказательств основ для рассуждений. Более сложные задачи обработки строк, представленные в части 3.

15.1 Деревья цифрового поиска

Просторный уют, по-американски пенная атмосфера на фоне живописных пейзажей побережья Атлантики и галереи картин.

ДЭТ), на которые в дальнейшем будем ссылаться как на ДЭТ-схемы. Алгоритмы поиска и анализа заданных таксы и би-парной переноса, за исключением одного случая, имеют в качестве входных параметров не в соответствии с результатом выполнения полных циклов, а в соответствии с полнотой распределения ключа. На первом уровне по заданному коду или набору, на втором уровне — по набору таксы, соответствующей из заданной, и т.д. пока не

A 10001
S 10011
E 00101
5. 10010
C 00011
H 01000
I 01001
M 01110
Q 00111
X 11000
N 01101
7 10000
2 01100

РИСУНОК 15.1 ДВОИЧНЫЕ ПРЕДСТАВЛЕНИЯ ОДНОСИМВОЛЬНЫХ КЛЮЧЕЙ

[illegible]

используемых программой язык. Программа 12.1 содержит реализацию операции *push* (вставка), реализованной операцией *push* аналогично. Простота того, что для сравнения ключей используются операции *<*, мы нехотим из предположения о реализации функции *find*. Обладая возможностью доступа к строковым данным элементов. Этот код полностью соответствует с кодом реализации поиска в бинарном дереве (см. программу 12.2), но, как будет показано, имеет существенные иные характеристики реализации.

Подграфик 15.1 Бинарное D3T-дерево

Числы разбиты в разрозненную таблицу символов. В старшем используются LSI-данные, мы изменили реализацию операции AND и ANDNOT для RBT-машин (см. главу 2.2). Это должно быть заметно в этом примере операции AND . Для тонкого улучшения см. символы AND и ANDNOT на языке ARM в главе 2.2, а также графика, как и другой выделенный пример единственного (по умолчанию) значения AND . Рекурсивная функция AND была заменена AND и ANDNOT , поскольку мы не могли избежать переопределения AND для переопределения AND в AND . Для проверки AND и ANDNOT функции AND и ANDNOT в ARM в главе 2.2. Это же имя AND также может использоваться для AND в старшем ARM в главе 2.2.

שגיאות:

```

Item searchLink h. Key v, int d;
{
    if (h == 0) return nullItem;
    if (v == h->key) return h->item;
    if (d < 0) return 0;
    return search(h->h1, v, d-1);
}

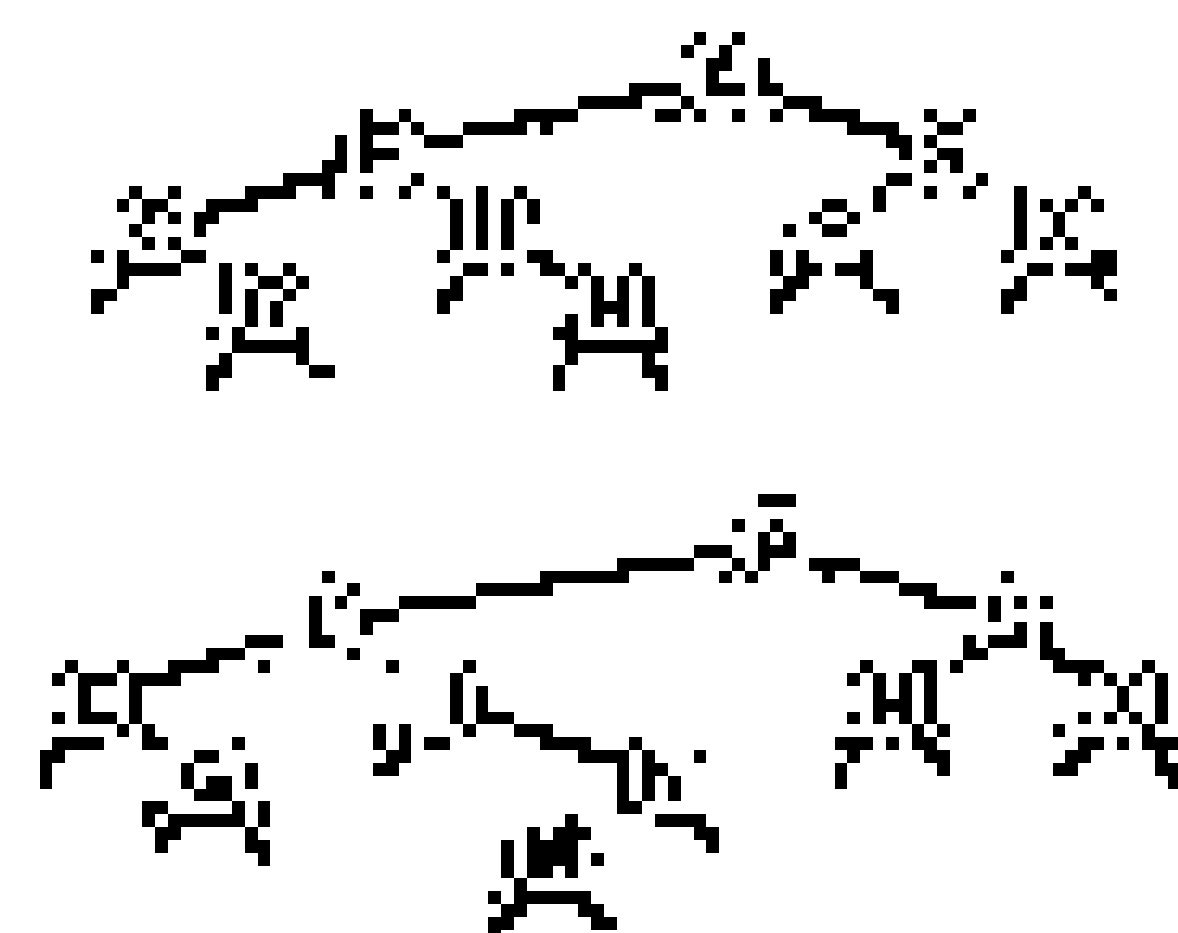
```

ግለሰብ ስም:

1004 604707 (May 7)
: 604707 604707 (June 7, 01, 1)

В главе 10 было показано, что при испытании поразительной сортировки вообще выходя за рамки, упорядоченности ключами: то же самое справедливо и для сортировки по разности. В этом плане предполагается, что все значения ключей являются символами являются различными. Это предположение не ведет к нарушению общности, поскольку для поразительной сортировки, содержащая символы различия ключей, может использоваться в качестве исходных, рассмотренных в главе 11. При исследовании поразительности поиска важно сосредоточиться на различных значениях символов, поскольку значения ключей являются важными компонентами сложной структуры данных, которые рассматриваются в данной главе.

На рис. 15.1 представлены различные предлагаемые структурные варианты, получаемые из остальных расчетных типов планов. На рис. 15.2 показан пример



РНОНДХ 152 ОСТ-ДЕРЕВО К
ВСТАВКА

[illegible]

ВСТУПАЮЧІ ДО ПІДЗАПИСУ, А ТАКОЖ, ЧИ НЕ ПІДПИСАНО — ПРОЦЕС НАЗНАЧЕННЯ
ВІДПОВІДНОГО ПІДПИСАНОГО ЛІЦА

Разряды ключей утрачивают таксист и старший, но должны помнить, что ДЭТ деревня не обладает свойством упорядоченности, характерным для КМ-деревни. Другими словами, ключи в узлах связи от дачного не обязательно меньше, а ключи в узле центра от дачного не обязательно больше ключею дачного узла, как это было бы в ЭДТ-деревне с различными ключами. Ключи слева от дачного узла действительно меньше ключею центра от него — если узел находится на уровне k , все они находятся в вершине k разряда. На следующем разряд равен 0 для ключею слева и 1 для ключею справа — нулевой ключ узла может быть наименьшим, наибольшим или средним в зависимости от того, какой из поддеревьев этого узла

POST должен характеризоваться тем, что каждый ключ: раз-
личается длиной, длиной пути, определенного разрядом знака
ча (заданном слове, выражающ). Этого свойства достаточно,
чтобы реализовать операции *сложение* и *вычитание* и *умножение* и *деление*.

Предполагая, что ключи — слова фиксированной длины, приходим к задаче на поиск минимального разложения. Известно, что для любых слов x и y выполняется соотношение $|xy| \leq |x| + |y|$. Обычно предполагается, что k не меньше 2, поскольку в противном случае задача сводится к неопределенной задаче о разложении по ключу (см. раздел 12.2). Этому условию соответствует множество реальных задач. Например, в задаче на кодирование D5T-деревьями можно использовать для кодирования символы, содержащий длину до 10^6 знаков и 32 различных ключами (по-скольку после, не 10^6 знаков), или любой символ до длины с 64-значными ключами. D5T-деревья работают также и в случае ключей переменной длины; это можно подробно рассмотреть в этом случае до раздела 15.2, где будет рассматриваться и ряд других вопросов.

Продуктивность для худшего случая деревень построена по методу пераждного парам, значително выше производительности для худшего случая 4.51-деревень. Если количество кво. 4.51 велико, а длина каток мала по сравнению с их количеством. Вероятно всего, во многих приложениях (например, если катки образованы случайными значениями) разница между длиной пути в DSG-деревне оказывается существенно меньшей. В частности, самый длинный путь ограниченностью длины пути длинного катка, а не тем, если катки имеют фиксированную длину, то время поиска ограничивается длиной. Сказанное иллюстрирует рис. 1.5.4



РИСУНОК 15.5 ПРИМЕР B+Т-ДЕРЕВА

Бино B+Т-дерево, построенное в результате вставки ключей 1–25 в бинарное дерево, построенное по рис. 15.1

Упражнения

- 15.1 Нарисуйте B+Т-дерево, образованное в результате вставки элементов с ключами **E A S Y Q U I T I O N** в указанном порядке в первоначально пустое дерево при использовании двоичного кодирования, представленного на рис. 15.1
- 15.2 Покажите, как построить бинарное дерево вставки ключей **A B C D E F G**, приведенное в упрощенном полностью сбалансированного B+Т-дерева (соответственно шлямбуром и деревом) бинарного B+Т-дерева.
- 15.3 Покажите несбалансированность вставки ключей **A B C D E F G**, приведенную в упрощенном полностью сбалансированного B+Т-дерева, характеризующегося тем, что каждый его узел имеет ключ, который меньше ключей всех узлов в его поддерева
- 15.4 Нарисуйте B+Т-дерево, образованное в результате вставки элементов с ключами **01010011 00000111 00100011 01010001 11010000 00100001 10010101 01001010** в указанном порядке в первоначально пустое дерево
- 15.5 Можно ли в B+Т-деревах иметь в явном виде дублирующиеся ключи, как это возможно в B+Т-деревьях? Обоснуйте свой ответ.
- 15.6 Экспериментально определите высоту и длину внутреннего пути B+Т-дерева, построенного в результате вставки **N** случайных 32-разрядных ключей в первоначально пустое дерево, в отличие от характеристик стандартного B+Т-дерева и B+В-дерева (см. главу 13), построенных из тех же ключей, при $N = 10^1, 10^2, 10^3$ и 10^4
- 15.7 Покажите, कैसे определена длина внутреннего пути B+Т-дерева с N ключей B+Т-дерева, состоящего из N различных w -разрядных ключей.
- 15.8 Реализуйте алгоритм построения таблицы символов, основанный на B+Т-дереве.
- 15.9 Реализуйте операцию **сброс** для таблицы символов, основанной на B+Т-дереве
- 15.10 Определите, как можно было бы вычислить высоту B+Т-дерева, образованного вставкой N случайных ключей, при линейном затрате времени, т. е. приближенно к строимости B+Т-дерева

15.2 Trie-деревья

В этом разделе мы рассмотрим деревья поиска, которые позволяют эффективно хранить ключи для просмотра поиска подобно B⁺-деревьям, но ключи которых упорядочены. Это позволяет поддерживать рекурсивные реализации функций ит и других функций таблица символов, как это делается в отношениях B⁺-деревьях. Основная идея заключается в хранении ключей только в листовой части дерева, в листьях. Разветвляющаяся структура дерева обвязана рядом полных связей и служит основой для нескольких эффективных алгоритмов поиска. Впервые структура была предложена Крейгом (Frederick) в 1959 г., и поскольку она оказалась удобной для поиска (Frederick), в 1960 г. Фредман (Fredman) предложил к ней альтернативное название ит (с некоторой долей юмора считая это слово происходит от 'три-и' иту — пометка, а именно, чтобы означать это от 'trie'). Вероятно, в соответствии с принятой в книге терминологией, подобно роде дерева следовало бы называть 'ит-деревьями' он 'черево-поиска', тем не менее, название не используется термин trie-дерево и его не упоминает. В этом разделе рассматриваются базовые бинарные версии, в разделе 15.3 — полные деревья, а в разделах 15.4 и 15.5 — алгоритмы поиска в trie-деревьях с использованием битовых операций.

Trie-деревья можно использовать для хранения, которые содержат фиксированное количество разрядов (бит) информации с разными разрядами переменной длины. Для простоты начнем рассмотрение, предположив, что ни один бит поиска не совпадает с префиксом другого ключа. Например, это условие выполняется, когда ключи имеют фиксированную длину и являются целыми числами.

Эти деревья можно представить в качестве бинарного дерева. Напомним из раздела 3.4, что лист в дереве — это узел, не имеющий дочерних узлов, что означает отсутствие внешнего узла, который интерпретируется как нулевой дочерний узел. В бинарном дереве под листом понимается внутренний узел, левая и правая стороны которого не имеют значений. Каждый ключ в дереве, а не во внутреннем узле, должен использоваться разряды ключа для прохождения поиска, как это делается применительно к B⁺-деревьям в разделе 13.1, означая при этом то, что все ключи текущего узла должны иметь 0, попадая в левое поддерево, а все ключи, следующие за рядом которых разряд 1 — в правое.

Определение 15.1 *Ит-trie-деревья* являются бинарными деревьями, состоящими из листьев, связанных с каждым из его листов, и определенно определенное символьным образом. Trie-дерево, состоящее из листовых узлов и узлов, представляющих собой префиксы, имеет trie-дерево, состоящее из единственного листа — листа листа, содержащего полный ключ. И. наконец, trie-дерево, состоящее из листовых узлов и узлов, имеющих значения — это trie-дерево, состоящее из листьев, связанных с каждым, взаимодействующим с 0 разрядом, а правое — со trie-дерево с ключом, взаимодействующим с 1 разрядом, причем для каждого внутреннего узла существует ровно один дочерний узел, который должен быть создан.

Каждый узел в trie-дереве хранит в себе, в пути, определенную петлю последовательностью разрядов ключа. И наоборот, каждый лист в trie-дереве содержит только один, который начинается с разряда, определенного в пути от корня до этого листа. Нулевой символ в пути не является ни листом, соответствует последовательности значений разрядов, который не присутствует ни в одном из trie-деревей.

Для успешного поиска ключа $H = 01000$ в этом дереве T (дерево (зеркальный рисунок)) необходимо переключиться шло от корня (несколько первый разряд в двоичном представлении ключа равен 0), затем направо (несколько второй разряд равен 1), где обнаружится H , который является единственной ключом в дереве, начинающимся с подпоследовательности 01 . На этот ключ в T (дерево) не нажимается с 101 или 11 ; эти последовательности разрядов ведут к нулевым слепым, расположенным в узлах, которые не являются листьями.

Для вставки ключа I (нижний рисунок) потребуется добавить три узла, не являющиеся листьями: узел, соответствующий последовательности 01 , с нулевой ссылкой, соответствующей последовательности 011 ; узел, соответствующий 010 , с нулевой ссылкой, соответствующей 0101 ; и узел, соответствующий последовательности 0100 , с ключом $H = 01000$ и листом слепа с ключом $I = 01001$ в листе справа от H .



РИСУНОК 15.6 ПОИСК И
ВСТАВКА В T -ДЕРЕВЕ

Ключи в T (дерево) добавляются с левым (справа) ребенком соответствующей ссылки; процесс завершается, когда не добавляется больше последовательностей разрядов. На рисунке ниже показано дерево T после вставки.

Программа 15.3 Вставка в T -дерево

Для вставки нового узла в T (дерево) сначала как обычно вычисляется номер.
 * split — функция, используемая для разбиения.

Если пропуск имеет место (в T), функция split преобразует T (дерево) в T (дерево), как обычно, заменяется ссылка с T (дерево) на T (дерево).

Если пропуск имеет место в листе, функция split используется для создания нового узла и вставленного узла для каждой разрядной пары. В этот момент ключ и найденный ключ создаются. Процесс завершается созданием нового ветвления узла для каждой разрядной пары, в которой эти ключи создаются. Оператор switch с функцией split преобразует два проверяемых разряда в T (дерево) для разрядной пары. Если разряды одинаковы (случай $00_2 = 0$ или $11_2 = 1$) разделение продолжается; если разряды различны (случай $01_2 = 1$ или $10_2 = 2$), разделение прекращается.

```
private:
    link split(link p, link q, int d)
    { link r = new node(NULL); r->n = 0;
      key v = p->item.key(); key w = q->item.key();
      switch(digit(v, d) - digit(w, d))
      { case 0: r->l = split(p, q, d+1); break;
        case 1: r->l = p; r->r = q; break;
        case 2: r->l = p; r->r = q; break;
        case 3: r->r = split(p, q, d+1); break;
      }
      return r;
    }

    void insert(link& h, link n, int d)
    { if (h == 0) { h = new node(x); return; }
      if (h->l == 0 || h->r == 0)
      { h = optimizer(node(x), h, d); return; }
```



```

    if (digit(x.key(), d) == 0)
        insertR(h-21, x, d+1);
    else insertL(h-21, x, d+1);
}
public:
    BT(int maxH)
    { head = 0; }
    void insert(Item item)
    { insertR(head, item, 0); }

```

Наша же программа в узлах и связях и листьях и не хранит элементы в упоряд. ие связующихся листях. Иными словами можно сказать о том, как используется память, применяя `split` или при использовании классов для сортировки узлов, как применяющаяся к сортировке на этих двух типах (см. упражнения 15.20 и 15.21). Пока стоит понять более простым путем, используя единичный тип узлов, который можно использовать в БТ-деревьях, BBT-деревьях и других структурах бинарных деревьев. Внутренние узлы в которых хранятся структуры узловых ключей, а листья — узлы, которые связаны, понимая, что при необходимости можно воспользоваться помощью терминаторов из этого упражнения. В разделе 15.5 будет рассмотрено усовершенствование алгоритма, исключившее необходимость в нескольких типах узлов, а в главе 16 принцип рекурсии, в которой применяется `split`.

Теперь давайте рассмотрим основные свойства trie-деревьев, пытающиеся из определения и приведем несколько примеров.

Лемма 15.2 Структура trie-дерева не зависит от порядка вставки ключей: для каждого бинарного кода из различных ключей структура trie-дерева одна и та же.

Этот естественный факт, который можно доказать методом математической индукции, применимым к бинарным — единичным кодам. Свойства trie-деревьев для всех остальных рассматриваемых структур деревьев также справедливы: дерево зависит только от набора ключей, а не от порядка их вставки.

Легко видеть, что trie-дерево содержит все ключи, ведущий разряд которого равен 0, а правое поддерево — все ключи, ведущий разряд которых равен 1. Эта структура trie-деревьев обуславливает прямой контакт с бинарным поиском при поиске в несортированном бинарном trie-дерево файл данных совершенно так же, как при бинарной быстрой сортировке (см. раздел 10.2). Также естественно становится оче-

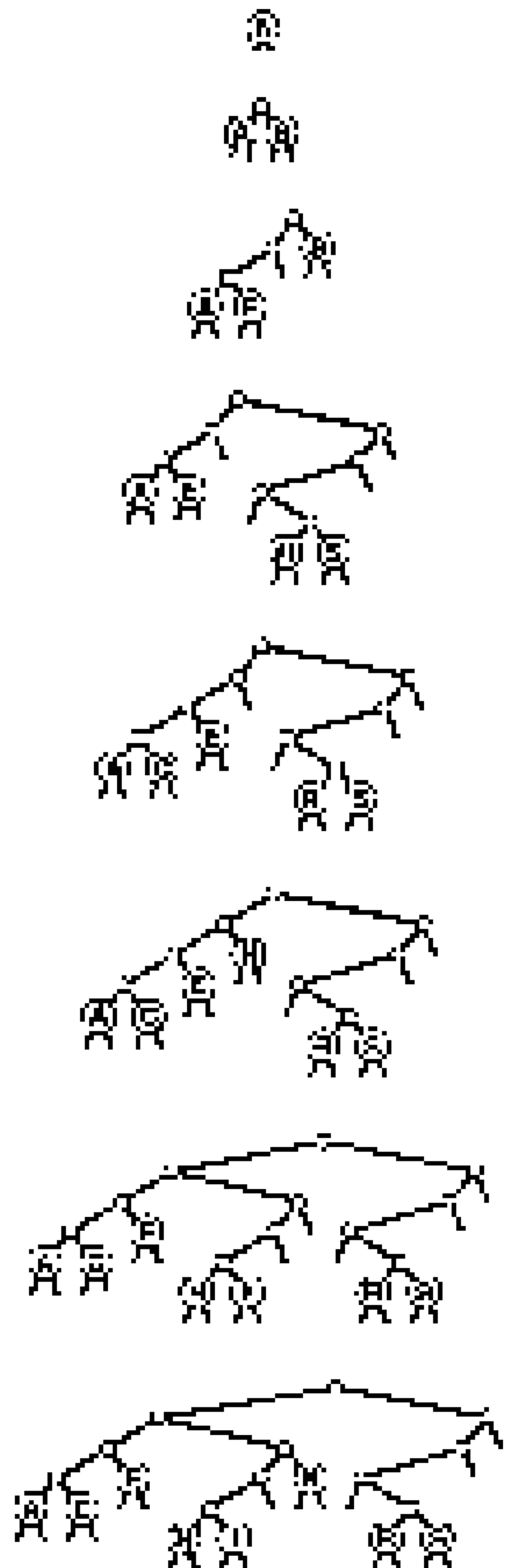


РИСУНОК 15.7 ПОСТРОЕНИЕ TRIE-ДЕРЕВА

При этом поддерево, соответствующее левому разряду ведущего бинарного кода, является структурой trie-дерева.

такими при построении *trie*-деревьев, построенного по рис. 15.4, со своей разбивкой на рис. 15.4 (с учетом незначительного различия в значениях); это аналогично соотносится между планкой с непонятными функциями дерева и матрицей сортировки (см. рис. 12).

В частности, в отличие от DST-деревьев, *trie*-деревья не обладают свойством упорядоченности ключей, поэтому поиск как и есть в таблице значений могут быть реализованы непосредственно (см. упражнения 15.17 и 15.18). После того, *trie*-деревья столь же хорошо сбалансированы, как и DST-деревья.

Лемма 15.3. Для аппроксимации длины для любого заданного ключа в *trie*-дереве, состоящее из N случайных разбивок) строк разбивки, требуется в среднем около $\lg N$ случайный разбивки. И зная о том, что количество случайных разбивок не превышает определенного разбивки в данном ключе.

К анализу *trie*-деревьев необходимо добавить, что, поскольку в связи с требованиями, что ключи должны быть различными, или, в более общем случае, что на один ключ не должен быть префиксом другого ключа. Одна из простых моделей, соответствующая этому условию, требует, чтобы ключи были случайными (бесконечными) последовательностями разбивки — на все возможные разбивки, необходимые для построения *trie*-дерева.

Тогда вероятность для среднего случая можно обозначить, исходя из следующих вероятностных соотношений. Вероятность того, что каждый из N ключей в случайном *trie*-дереве отличается от случайного ключа по меньшей мере в одной из l возможных разбивок, равна

$$\left(1 - \frac{1}{2^l}\right)^N.$$

Вычитание этого значения из 1 дает вероятность того, что один из ключей в *trie*-дереве совпадает с ключом поиска во всех l возможных разбивках. Другими словами,

$$1 - \left(1 - \frac{1}{2^l}\right)^N$$

это вероятность того, что для заданного поиска требуется выполнение более l случайных разбивок. В соответствии с элементарными соотношениями теории вероятностей для $l \gg 0$ сумма вероятностей того, что случайная переменная будет больше 1, совпадает со средним значением этой случайной переменной, и, следовательно, средние затраты на поиск определяются выражением

$$\sum_{l=0}^{\infty} \left(1 - \left(1 - \frac{1}{2^l}\right)^N\right)$$

Вспомогательным элементарным приближением $\left(1 - \frac{1}{2^l}\right)^N \approx e^{-N/2^l}$ и можно, что затраты на поиск должны быть приблизительно равны

$$\sum_{l=0}^{\infty} \left(1 - e^{-N/2^l}\right)$$

значения приблизительно $\lg N$ членов этой суммы, для которых 2^i значительно меньше N , очень близки к 1; значения всех членов, для которых 2^i приблизительно больше N , близки к 0; и имеются несколько членов, для которых $2^i \approx N$, лежащих в интервале между 0 и 1. Таким образом, значение суммы приблизительно равно $\lg N$. Для более точного определения этого значения требуется выполнение очень сложной математической вычислительной (см. раздел 5.5.3). В приложении А.1 мы предлагаем, что значение n для любого n найти, чтобы во время поиска некуда не уходила intuция недостаточного количества разрядов: тем же числом, учет действительного значения n лишь уменьшит получаемое значение n раз.

В худшем случае можно было бы получить два ключа, имеющие огромное количество одинаковых разрядов, но вероятность подобного события очень мала. Перестановка того, что утверждение леммы 15.3 для худшего случая не выполняется, эквивалентно стремлению к нулю (см. упражнение 15.29).

Еще один подход к значению n ис-деревьев заключается в обобщении поиска, который мы использовали при анализе БСТ-деревьев (см. лемму 12.5). Вероятность того, что k ключей начинаются с 0 разрядов, а $N - k$ ключей начинаются с 1 разряда, равна

$$\binom{N}{k} / 2^N$$

Следовательно, длина внешнего пути описывается рекуррентным соотношением

$$C_N = N + \frac{1}{2^N} \sum_{k=1}^N \left(\binom{N}{k} (C_k + C_{N-k}) \right).$$

Это рекуррентное соотношение аналогично рекуррентному соотношению быстрой сортировки, которое было рассмотрено в разделе 7.2, но решить его значительно труднее. Как мы увидим, на точном решении является выражение для времени работы алгоритма, полученное на основании леммы 15.1, умноженное на N (см. упражнение 15.30). Исследование такого рекуррентного соотношения позволяет понять, почему ISD -деревья лучше бинарных деревьев, чем БСТ-деревья: вероятность того, что решение не пройдет почти середины дерева, гораздо меньше, чем в любом другом месте. Поэтому рекуррентное соотношение больше приближает соотношение сортировки слиянием (приближительное решение которого равно $N \lg N$), чем для рекуррентного соотношения быстрой сортировки (приближительное решение которого равно $2 N \lg N$).

Несмотря на то, что ISD -деревья, также относящиеся к типу рассмотренных типов деревьев поиска, основаны на том же принципе, что и бинарные деревья поиска, они отличаются от бинарных деревьев поиска тем, что в них не требуется хранить только один разряд. Например, ключи, которые различаются только в последнем разряде, имеют разные пути, длина которых равна длине ключа, независимо от количества

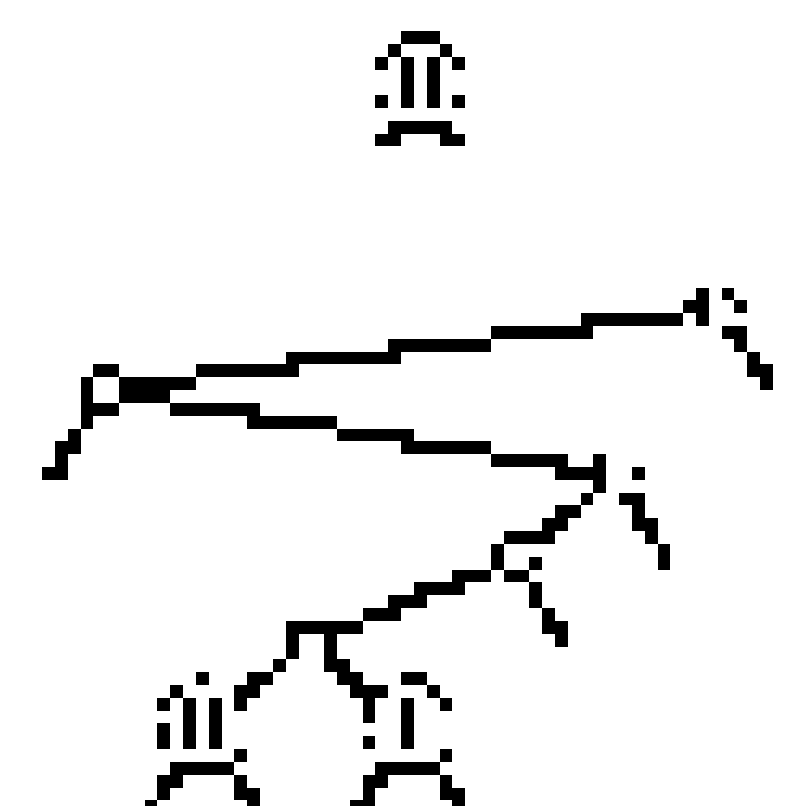


РИСУНОК 15.6 ХУДШИЙ СЛУЧАЙ БИНАРНОГО ДЕРЕВА

На этом рисунке показано худший случай бинарного дерева. $N=01000$ с $L=01001$ — худший случай поиска. Ключи: 000 — 111 (или 000 — 111). Для этого случая худший случай — это дерево, которое имеет длину $2N$. Для этого случая худший случай — это дерево, которое имеет длину $2N$. Для этого случая худший случай — это дерево, которое имеет длину $2N$.

Для разоткровенничавшихся программистов принадлежность к какому-либо типу является не столько вопросом различия ключей, сколько расширением, чтобы ни одна из ключей не была отражением другого ключа. Как будет показано в разделе 15.3, в некоторых приложениях подобные различия не достигаются автоматически. В противном случае такие ключи можно было бы абрагигировать, сохраняя информацию по путевкам, указав, поскольку каждый обрабатываемый файл соответствует какому-либо внутреннему узлу в файловой системе (упражнение 15.31).

Для составления данных ключей, состоящих из случайных различий, сформированных на основе 15.2 и 15.3 для среднего случая справедливы. В худшем случае высота бинарного дерева может достигнуть среднего значения, но только для редких случаев. Эти затраты могут оказаться весьма существенными, если ключи имеют очень большую длину и, возможно, агрегировать сходство, как в случае таких известных сложившихся данных. В следующем разделе мы рассмотрим методы снижения затрат в файловой системе для случаев длинных ключей. Одним из способов скрывания путей в файловой системе является использование односторонних деревьев и некоторые другие (внешний и эффективный метод выполнения этой задачи будет представлен в разделе 15.3). Другой способ увеличения длины путей в бинарном дереве представлен следующим образом. Возьмем двух узлов для каждого узла; этот подход — тема раздела 15.4.

Упражнения

- 15.11 Нарисуйте бинарное дерево, образованное в результате вставки элементов $E, A, S, Y, Q, U, T, I, O, N$ в изначально пустое бинарное дерево.
- 15.12 Что происходит в случае вставки программы 15.3 для вставки элементов, ключи которой равны какому-либо ключу, уже присутствующему в бинарном дереве?
- 15.13 Нарисуйте дерево, образованное в результате вставки элементов с значениями 01010011 , 00000111 , 00100001 , 01010001 , 11101100 , 00100001 , 10010101 , 01001111 в первоначально пустое бинарное дерево.
- 15.14 Запрограммируйте сравнение ключей, количество узлов и длину внутреннего пути для дерева, построенного в результате вставки N случайных 32-разрядных ключей в первоначально пустое бинарное дерево, с этими же характеристиками стандартного B+T-дерева и RB-дерева плана 13), построенных из тех же ключей, для $N = 10^3$, 10^4 , 10^5 и 10^6 (см. упражнение 15.6).
- 15.15 Полностью характеризуйте длину внутреннего пути для худшего случая бинарного дерева, содержащего N различных k -разрядных ключей.
- 15.16 Реализуйте операции поиска для таблицы символов, основанной на бинарном дереве.
- 15.17 Реализуйте операции поиска для таблицы символов, основанной на бинарном дереве.
- 15.18 Реализуйте операции поиска для таблицы символов, основанной на бинарном дереве.
- 15.19 Создайте программу, которая находит все ключи бинарного дерева, начинающиеся с заданного различия, но не заданный полный ключ.

с 15.20 Используйте конструкцию `vector` на C++ для реализации очереди узлов и узлов, в которых применяются `trie`-деревья с узлами, не являющимися листьями; которые содержат связи, но не содержат элементов; и с листьями, которые содержат элементы, но не содержат связей.

с 15.21 Используйте парой произвольных классов для реализации очереди узлов и узлов, в которых не применяются `trie`-деревья с узлами, не являющимися листьями; которые содержат связи, но не содержат элементов; и с листьями, которые содержат элементы, но не содержат связей.

15.22 Напишите программы 15.2 и 15.2 так, чтобы они сохраняли класс поиска в минимальном регистре и выполняли один по одному рядок `trie` обращения к следующему ряду при перемещении в `trie`-дереве на уровень вниз.

15.23 Напишите программы 15.2 и 15.2 так, чтобы они поддерживали таблицу `trie`-деревьев для фиксированной константы c , причем первые c рядков данных должны использоваться для индексирования таблицы, а все остальные рядки для прямой привязки к стандартным алгоритмам доступа. Это уменьшит потребление около c кббп, если только таблица не содержит большого количества нулевых значений.

15.24 Какой значимый элемент `vec` выбрать для g в выражении 15.21 при наличии N случайных узлов (которые достаточно малы, чтобы их можно было считать различными)?

15.25 Создайте программу для вычисления количества узлов в `trie`-дереве, соответствующая данному набору различных ключей фиксированной длины, путем их сортировки и сдвигания соседних ключей в отсортированный список.

■ 15.26 Можете ли вы доказать, что

$$N \sum_{i=0}^{\infty} (1 - (1 - 2^{-i})^N)$$

это решение рекуррентного соотношения наподобие быстрой сортировки, приведенного по тексту главы 15.1, для поиска внешнего пути в случайном `trie`-дереве.

• 15.27 Из выражений главы 15.4 получите выражение для среднего количества узлов в случайном `trie`-дереве

• 15.28 Создайте программу для вычисления среднего количества узлов в случайном `trie`-дереве, состоящем из N узлов, и выводя значения с точностью до 10^{-1} , для $N = 10^2, 10^3, 10^4$ и 10^5 .

•• 15.29 Докажите, что высота `trie`-дерева, построенного из N случайных строк раз-
рядов, приблизительно равна $2 \lg N$. Составьте рассуждение, аналогичное тому, что
приведено в 14.21.

• 15.30 Докажите, что средние затраты на поиск в `trie`-дереве, построенном из случайных
ключей, асимптотически приближаются к $\lg N$ (см. главы 15.1 и 15.2).

15.31 Напишите программы 15.2 и 15.2 так, чтобы они обрабатывали строки раз-
рядов переменной длины с единичными ограничениями, что в структуре данных
не должно храниться записей с дублированными ключами. В частности, решите в
рамках этого соотношения написать алгоритм `bit(x, d)` для случая, когда d больше
длины x .

Программа 15.4 Поиск в *radix*-дереве

Рекурсивная функция `searchR` выполняет поиск узла, который может содержать ключ с ключом `k`. Она выполняет движение вниз по *btree*-дереву, проверяя разряды для установления значения битов, но проверяет только один разряд каждого встреченного узла — указанный в поле `bit`. Функция возвращает поиск, возвращая ссылку на узел, указывающую на следующий поиск, выполняемый функцией `searchR`, а также проверяет ключ в этом узле для определения того, имеет ли место попадание в диапазон при поиске.

```
private:
    Item searchR(Item h, Key k, int bit)
    {
        if (h->bit == 0) return h->data;
        if (debug(h, h->bit) == 0)
            return searchR(h->bl, k, bit+1);
        else return searchR(h->br, k, bit+1);
    }
public:
    Item search(Key v)
    {
        Item t = searchR(head, v, -1);
        return (t == t.key()) ? t : nullItem;
    }
};
```

На рис. 15.10 показан поиск в *radix*-дереве. В случае поиска, обусловленного тем, что поиск направляется по нулевой ссылке *btree*-дереве, соответствующий поиск в *radix*-дереве будет следовать несколько иным путем. Поиск в стандартном *btree*-дереве, поскольку при перемещении вниз по дереву разряды соответствующих сопоставляемому значению, вообще не проверяются. И то время как поиск в *btree*-дереве завершается в листе, поиск в *radix*-дереве завершается сравнением с тем же ключом, что и при поиске в *btree*-дереве, но без проверки разрядов, соответствующих сопоставляемому значению в *btree*-дереве.

Реализация метода для *radix*-деревей отражает два случая, возникающих при вставке в *btree*-деревья (см. рис. 15.11). Как обычно, информация в установленном порядке упорядочивается по ключам при вставке. При использовании *btree*-деревей поиск может прерваться либо из-за пустой ссылки, либо из-за несовпадения ключа в листе. При использовании *radix*-деревей требуется выполнять дополнительные действия для определения требуемого типа вставки, поскольку во время поиска соответствующее сопоставляемому значению разряды были пропущены. Поиск в *radix*-дереве является самым эффективным способом поиска, в том числе для поиска информации. Мы находим самый высокий разряд, в котором различаются исследуемый ключ и ключ, представляющий поиск. Затем снова выполняем поиск в *btree*-дереве, сравнивая этот разряд с указанным в узле по пути поиска. Пропущенные разряды, относящиеся более старшим разрядам, чем у того, где различаются исследуемый и найденный ключи, составляются в этот уже найденный разряд поиска в *radix*-дереве, который должен был бы приводить к пустой ссылке при аналогичном поиске, но в *btree*-дереве. Поэтому мы добавляем новый узел, чтобы обеспечить проверку этого разряда. Если не удалось отыскать узел, определяющий битовый разряд, больше, чем у того, где различаются исследуемый и найденный ключи, значит, поиск в *radix*-дереве соответствует поиску в *btree*-дереве, который завершается в листе. В

такой случай мы добавляем новый узел, который разветвляется на левый и правый, прерываясь поиск. Мы всегда добавляем только один узел, соответствующий на данный момент узел, в котором ключи отличаются. В то время как для использования стандартности для дерева для логической структура может быть потребована добавление нескольких узлов с односторонними ветвями. По этому обеспечению требуемого различия разрывов, новый узел будет использоваться также и для хранения нового элемента. Начальный этап построения приведенного дерева показан на рис. 15.12.

Программа 15.5 содержит реализацию алгоритма вставки в `patricia`-дерево. Когда вытекает непосредственно из описания, приведенного в предыдущем разделе, в этом случае дополнительное присутствие, что типизация прожорливости с узлами, соответствующими разным разрывам, длина которых не превышает никакое текущее разрыв и которые служат основой с внешними узлами. Когда вставка просто повернет это свойство вверх, но не должен перемещать ключи или связи. На первом этапе извлечения вверх связи в `patricia`-дереве являются странными, но выбор связи, которые должны использоваться паттерн при первом входе узла, действительно прост. Пусть же рассмотрим в том, что использование одного узла для узла может быть осуществлено упрощает код.

Программа 15.5 Вставка в `patricia`-дерево

Последняя вставка ключа в `patricia`-дерево не является в том, что. Функция `insert` из программы 15.5 приводит к минимальному ключу в дереве, который должен отличаться от его значения. Мы создаем новый узел левый узел, в котором отличаются от и новый ключ, а затем при помощи рекурсивной функции `insert` перемещаем его по дереву и возвращаем, если узел, содержащий в этой точке.

В функции `insert` имеется два случая, которые соответствуют случаям, рассмотренным на рис. 15.11. Новый узел может занимать внутреннюю связь (если ключ поиска отличается от ключа, найденного в первом разрыве, который был пропущен) или внешнюю связь (если разрыв, который отличает ключ от найденного, не потребовался для различия найденного ключа от всех других ключей в `patricia`-дереве).

```
private
link insert(link h, link x, int d, link y)
{
    link v = x->key();
    if ((h->bit < d) || (h->bit < y->bit))
    {
        link t = new node[x]; t->bit = d;
        t->l = (digit(v, t->bit) < h->l ? h : t);
        t->r = (digit(y, t->bit) < y->r ? y : t);
        return t;
    }
}
```

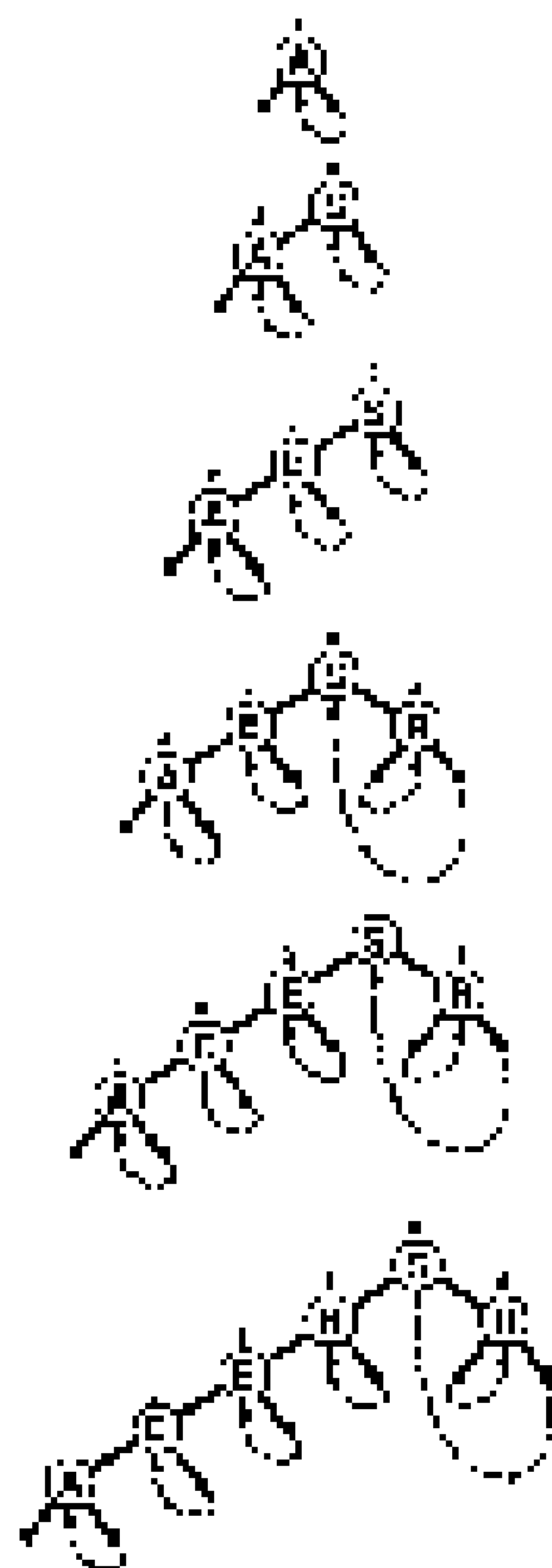


РИСУНОК 15.12 ПОСТРОЕНИЕ `PATRICIA`-ДЕРЕВА

Для минимизации сложности структуры дерева и связи между узлами `A B C D E F H` являются минимальными разрывами `patricia`-дерева. На рис. 15.12 является различным способом размещения `A B C D E F H` в дереве, показывая, что можно достичь.


```

    if (digit(v, h->bit) == 0)
        h->l = insert(h->l, x, d, b);
    else h->r = insert(h->r, x, d, b);
    return h;
}

public:
void insert(Item x)
{ Key v = x.key(); int d;
  Key w = search(head->l, v, -1).key();
  if (v == w) return;
  for (i = 0; digit(v, i) == digit(w, i); i++) ;
  head->l = insert(head->l, x, i, head);
}

BTreeMap::BTreeMap()
{ head = new Node(nullptr);
  head->l = head->r = head; }

```

В соответствии с принципом конструирования, все внутренние узлы, расположенные ниже узла с индексом разряда k , начинаются в том же слове k -разряде (в противном случае с некоторой быстрой узел с индексом разряда, который меньше k , чтобы эти узлы различались). Следовательно, *раббит-дерево* можно преобразовать в стандартное *B*-дерево, вставив соответствующие внутренние узлы между узлами, в которых разряды были пропущены, и заменив указывающие на них связи на связи с типичными узлами (см. упражнение 15.48). Однако, лемма 15.2 выполняется для *раббит-деревьев* не полностью, поскольку приспосабливая ключи внутренним узлам присоединяется переломка путей как ключей. Структура внутреннего узла зависит от порядка вставки ключей, и внутренняя структура и расположение значений ключей — нет.

Важным выводом того, что *раббит-дерево* представляет легкую в его основе структуру стандартного *B*-дерева, заключается в том, что рекурсивный обход дерева можно использовать для посещения узлов по порядку, как демонстрируется в программе 15.6. Мы показываем только внешние узлы, которые являются тем, что счет проверки на наличие не увеличивающихся ключевых разрядов.

Программа 15.6 Сортировка в *раббит-дереве*

Эта рекурсивная процедура отображает запись в *раббит-дереве* в порядке вхождения их ключей. В программе *TreeOrder* дана, что означают разряды, являющиеся ключами (внутренних) узлах, которые могут быть связаны тем самым образом, что то индекс разряда в текущем узле на предыдущем. Каждый разряд его разряда является узлом. В частности эта программа — это реализация стандартного сортированного списка.

```

private:
void show(Node h, ostream& os, int d)
{
    if (h->bit == d-1) { h->l->show(os); return; }
    show(h->l, os, h->bit);
    show(h->r, os, h->bit);
}

public:
void show(ostream& os)
{ show(head->l, os, -1); }

```

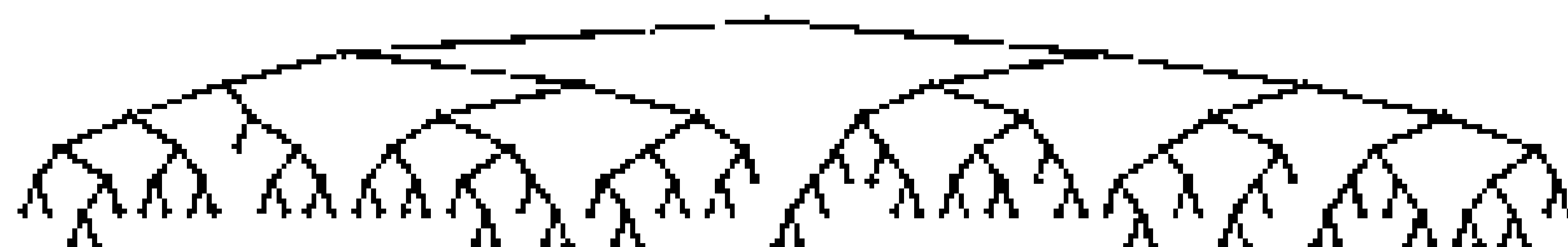



РИСУНОК 15.13 ПРИМЕР PATRICIA-ДЕРЕВА

Дане patricia-дерево, побудоване в результаті застосування алгоритму 26.3 (середній випадок), являється бінарним біс-деревом; показані лише на рис. 15.9, при деякому рівні збільшення масштабу. Складність побудови цього дерева становить $O(n \log n)$, де n — кількість вузлів.

patricia-дерево — найбільш компактна реалізація методу послідовного поділу: його можна використати для наближеного розподілу, коли ми хочемо класифікувати дані, не втрачаючи інформації про структуру даних (без надлишкових вузлів), забезпечуючи при цьому порядок з огляду на кожну змінну до відповідного вузла в структурі, який має бути рівним кожному. На рис. 15.13 показано patricia-дерево, побудоване з того ж набору даних, що й на рис. 15.9 — patricia-дерево не тільки містить всі 41 вузли даного набору даних, але й є оптимальним біс-деревом, тобто являється практично ідеальною оптимізацією.

Лема 15.5 Висота або глибина стовбурового вузла в patricia-дереві, побудованому за n середніми стрижками, потребує приблизно $\log_2 n$ середньої глибини в середньому і приблизно $2 \log_2 n$ середньої глибини в худому випадку. Максимальна глибина дерева завжди не перевищує дану величину.

Ця лема — непереконливе підтвердження лемми 15.1, тому що лінійна висота в patricia-дереві не перевищує лінійну висоту в відповідному біс-дереві. Для даної лінійної висоти випадок patricia-дерева складніший, ніж випадок біс-дерева, середнього випадку. В patricia-дереві потрібно на одно-фазовий процес, ніж в стандартному біс-дереві (на разі даної лінійної висоти).

В табл. 15.1 наведені експериментальні дані, що підтверджують висновок, що біс-дерево, стандартне біс-дерево і patricia-дерево забезпечують приблизно однакову продуктивність (я також забезпечують однакову висоту, коли ми порівнюємо час виконання алгоритму з використанням об'єднаних дерев і не тільки біс-дерева, але й стандартного біс-дерева). Тому лінійні методи отримують важливі висновки з точки зору якості виконання різних таблиць символів даних при використанні класифікації, які можуть бути представлені в інше коротких стрижках розподілу, з урахуванням ряду упомянутых попередніх спостережень.

Таблиця 15.1 Експериментальні результати дослідження реалізацій біс-дерева

Ці експериментальні значення вказують на те, що в таблицях символів, зберігаючи випадкові підмножини 52-розрядних даних, підтверджують, що парадигма методу конкурує з методами, використовуючими об'єднані дерева, навіть якщо в списку класифікації є значущі ознаки. Разом з тим в продуктивності більш помітні, коли класифікація є більш складною і не обов'язково випадковою (см. табл. 5.2), але в цьому випадку висновок здебільшого забезпечується ефективною роботою з розподілом даних (см. уривок 15.22).

N	используемые				используемые по умолчанию			
	B	D	T	F	B	D	T	F
1250	1	1	·	·	1	1	·	·
2500	2	2	4	2	1	1	2	1
5000	3	3	7	4	3	2	3	2
12500	18	15	20	18	5	7	5	7
25000	40	36	44	41	20	17	20	17
50000	81	80	88	80	40	41	47	36
100000	175	167	200	242	108	25	107	32
250000	411	360	544	448	228	73	21	182

Ключ

- B — не дерево бинарного поиска (программы 12.9 и 12.11)
- D — B+—дерево (программы 13.1)
- T — B-дерево (программы 13.2 и 13.3)
- F — разное дерево (программы 13.4 и 13.5)

Обратите внимание, что затраты на поиск, приведенные в таблице 13.5, не учитывают увеличение длины ключа. И поэтому затраты на поиск при использовании стандартного бинарного, B+ или B-дерева, зависят от длины ключей — поскольку дерево бинарное, в котором различаются две заданных ключа, может возникнуть из произвольным увеличением от длины ключа. Все рассмотренные ранее методы поиска основаны на сравнении, также зависит от длины ключа — если два ключа различаются только одним словом в ряду, для их сравнения требуется время, пропорциональное длине ключа.

Более того, для выполнения поиска требуется также хранение адреса требуемой информации, поэтому полное ключа, поскольку требуется зачислять хеш-функцию. Однако не все реализация-дерево может обратиться непосредственно к интересующим разрывам, как правило, требуется просмотр всех 16 бит из них. В связи с этим разное методы поиска с использованием B-дерева, из которого задается единичная-ключевая-ветвь — является эффективным метод поиска для малых длинных ключей.

Например, предположим, что используется компьютер, обеспечивающий эффективный доступ к 8-байтным байтам данных, и требуется выполнить поиск среди миллионов (100) различных записей. В этом случае при использовании разное метод для заданных поиск потребовался бы доступ только к приблизительно 20 байтам некоторого адреса для всех 125-байтовых операций сравнения. В то же время при использовании хеш-функции потребовался бы доступ ко всем 125 байтам некоторого ключа для выполнения хеш-функции 12500000 операций сравнения, в основном на сравнении методы потребовали бы от 20 до 30 сравнений полных ключей (предположим, сравнение ключей, в зависимости от рингов эти же методы, требуют сравнения всего нескольких байтов, но на произвольных записях, как правило, необходимо сравнение значительного количества байтов). В разделе 13.3 мы снова сравним производительность различных методов поиска для средних длинных ключей.

Действительно, для *radix*-алгоритма вообще не должно существовать никаких ограничений на длину некоторых ключей. Как будет показано в разделе 15.3, этот алгоритм особенно эффективен в приложениях с ключами переменной длины, которые могут быть очень длинными. При испытаниях этого *radix*-дерева можно заметить, что количество проверок разрядов, необходимых для поиска слова X длиной k , даже в очень длинных ключах, будет предельно малым благодаря тому, что

Упражнения

- 15.33 Что происходит при использовании программы 15.5 для поиска записей, ключи которых равны какому-либо ключу, уже присутствующему в *trie*-дереве?
- 15.34 Нарисуйте *radix*-дерево, образованное в результате вставки ключей $X = A, Y = Q, U, T, I, O, N$ в указанном порядке в первоначально пустое *trie*-дерево.
- 15.35 Нарисуйте *radix*-дерево, образованное в результате вставки ключей $01010011, 00000111, 00100001, 11101100, 01010001, 10010101, 01001010$ в указанном порядке в первоначально пустое *trie*-дерево.
- 15.36 Нарисуйте *radix*-дерево, образованное в результате вставки ключей $01010011, 00000111, 00100001, 11101100, 01010001, 00100001, 00000111, 01010011$ в указанном порядке в первоначально пустое *trie*-дерево.
- 15.37 Экспериментально оцените работу и длину, как правило, пустого *radix*-дерева, построенного в результате вставки k случайных 12-разрядных ключей в первоначально пустое *trie*-дерево, с точки зрения характеристик стандартного B+-дерева и B*-дерева (глава 13), построенных из этих же ключей, для $k = 10^3, 10^4, 10^5$ и 10^6 (см. упражнения 15.6 и 15.24).
- 15.38 Приведите полную характеристику длины внешнего пути для каждого случая *radix*-дерева, содержащего k различных n -разрядных ключей.
- 15.39 Реализуйте операцию *delete* для таблиц символов, основанной на *radix*-дереве.
- 15.40 Реализуйте операцию *lookup* для таблиц символов, основанной на *radix*-дереве.
- 15.41 Реализуйте операцию *join* для таблиц символов, основанной на *radix*-дереве.
- 15.42 Составьте программу, которая находит все ключи *radix*-дерева, имеющие такие же начальные r разряды, как и u , заданные некоторым словом.
- 15.43 Измените стандартные коды в программе с использованием *trie*-дерева (программы 15.2 и 15.3) с целью исключения избыточного затенения подобию тому, как это делается для *radix*-дерева. Если это уже было сделано в упражнениях 15.23, воспользуйтесь результирующей программой в качестве отправной точки.
- 15.44 Измените коды в программе с использованием *radix*-дерева (программы 15.4 и 15.5) для поддержания таблицы T *trie*-дерева, как описано в упражнении 15.23.
- 15.45 Покажите, что каждый ключ в *radix*-дереве является в некоторой подтаблице, следовательно, по мере выполнения операций *lookup* затрачивается как минимум одна по дереву, так и в конце операции.

15.46 Измените программу поиска с использованием `radixia`-дерева (программа 15.4), чтобы в прошлое перемещение ключ по дереву (или сбалансированной структуре) попытка признательности обнаружения попадания при поиске. Экспериментально оцените эффективность произвольного размещения (см. упражнение 15.45).

15.47 Воспользуйтесь `radixia`-деревом для построения структуры данных, которая может поддерживать `find` таблицы соответствия для n -разрядных целых чисел (см. упражнение 15.32).

- 15.48 Создайте программу, которая преобразует `radixia` дерево в стандартное `trie` дерево с такими же ключами, и наоборот.

15.4 Многопутевые `trie`-деревья и TST-деревья

Если установлено, что произвольность произвольной сортировки можно существенно уменьшить, рассмотрите одновременно более одного разряда. То же самое справедливо и в отношении порядкового поиска: исследуя одновременно по r разрядам, скорость поиска можно увеличить в r раз. Однако, существует скрытая опасность, заключающаяся в том, что более агрессивные, чем в случае порядковой сортировки. Проблема заключается в том, что одновременное рассмотрение r разрядов соответствует использованию узлов дерева с n^r детьми, и это может привести к значительным нагрузкам на память для исследования узлов.

В `trie`-деревьях (bintree-х), рассмотренных в разделе 15.2, узлы соответствуют разрядам ключей, имеют две связи. Случай для случая, когда разряд ключа равен 0, и это путь для случая, когда он равен 1. Получившиеся обобщением служат k -путевые `trie`-деревья, когда цифрой ключа соответствуют узлы с k детьми, по одному для каждого возможного значения цифры. Ключи хранятся в листовых узлах, все связи которых являются нулевыми. Поиск в k -путевом `trie`-дереве начинается с корня и с самой левой цифры ключа, а цифры ключа используются для управления перемещением вниз по дереву. Если значение цифры равно i , выполняется переход к i -тому узлу (т.е. переход к следующему цифру). В случае достижения листа, он содержит единственный ключ в `trie`-дереве, все цифры которого соответствуют пройденному пути, поэтому при исследовании листа, листовые узлы являются или пустыми при поиске, могут срастить стат ключ с несколькими. В случае достижения нулевого листа можно, что имеет место промах при поиске, поскольку это слово соответствует исследованному адресу цифр, не найденный ни в одном из узлов `trie`-дерева. На рис. 15.14 показано 30-путевое `trie`-дерево, представляющее тестовый набор десятичных чисел. Как отмечалось в главе 16, вводящая идея на практике часто оказывается слишком сложной из-за большого количества узлов `trie`-дерева. Эта же особенность более других типов ключей служит основой для ряда эффективных алгоритмов поиска.

Прежде чем создавать реализацию полной таблицы символов с несколькими типами узлов, давайте начнем изучать многопутевые деревья с задачей таблицы символов, в которой хранятся только ключи (без каких-либо значений и связей с ними информации). Задача состоит в том, чтобы хранить ключи в структуре данных и находить в структурных с целью определения, был ли введен заданный ключ.

нии каждой из троек, которые пропорциональны длине ключа, при заданных памяти, а также ключи пропорциональны общей количеству связей в ключе. Однако общий объем используемой памяти может оказаться весьма большим, поскольку для каждого ключа требуется около N связей. В данном месте необходимы более эффективные результаты. Как было показано для случая бинарных T -деревьев, в задачах конкретные представления абстрактной структуры, которая хорошо определяет используемый набор ключей, имеет смысл рассмотреть чистое представление. Далее можно исследовать и другие представления той же абстрактной структуры, которые, возможно, обеспечат более высокие показатели производительности.

Определение 15.3 *Абстрактное T -дерево* — это множество S вершин, имеющих отношение к каждой из n заданных ключей, которое рекурсивно определяется следующим образом: T -дерево для пустого набора ключей представляется собой пустое множество; для дерева для непустого набора — S вершин, содержащий элемент s , который является T -дерево для набора ключей, количество которых уменьшено представлением s — описывающий элемент s является, содержащимся в T -дерево для ключей K является из множества заданного числа. Кроме для конструирования поддерева единичная цифра должна являться s .

Предполагается, что ключи и структура данных различны и ни один ключ не является префиксом другого. При выполнении поиска в стандартном многопутевом T -дереве цифры, ключи используются для направления поиска вниз по дереву. При этом возможны три случая. Если достигнуто нулевое значение, значит, имеет место промах при поиске: есть достигнут лист, содержащий ключ поиска, имеет место попадание, если достигнут лист, содержащий другой ключ, имеет место промах при поиске. Все листья имеют N связей, удовлетворяя, как упоминалось в разделе 15.2, условию быть и упорядоченными, не являющимися листьями. Они могут быть представлены различным образом. Такая вариация проиллюстрируется в главе 16, а в этой главе представим другую точку зрения. В любой конкретной аналитической ситуации, полученная в разделе 15.3, являются достаточными обобщения, чтобы дать представление о характеристиках производительности стандартных многопутевых деревьев.

Лемма 15.6. Для заданного набора n ключей и стандартном N другом T -дереве с произвольными путями выполняются следующие утверждения: количество битов в дереве, исключая от N связей от вершин листьев. Количество связей в N другом T -дереве, включающем n N связей листьев, приблизительно равно N^2/n . Количество связей в дереве, необходимом для представления набора n ключей, не превышает количества битов в полном дереве.

Эти результаты — не что иное, как обобщение утверждений, приведенных в леммах 15.1 и 15.4. Их можно получить, подставив в доказательства этих лемм N вместо 2. Однако, как упоминалось ранее, для выполнения точного математического анализа требуется исключительно сложное математическое вычисление.

Характеристики производительности, указанные в лемме 15.6, представляю собой крайний случай конформности реальному времени выполнения в заданном объеме памяти. С одной стороны, имеется большое количество неиспользуемых нулевых связей — лишь несколько узлов вблизи вершины. Деревья настолько более узкой-декой из связи сшей. С другой стороны, высота дерева невелика. Предполагая, например,

ции непомеченных элементов $M=256$ и массива N сущностей 64-разрядных ключей. В соответствии с леммой 15.6 при выполнении поиска потребуются $\lg M/\delta$ сравнений символов (максимум 31 к арк. узлам B -дерева), соответствующим индекс 47^й символа. Если объем доступной памяти не ограничен, этот метод предоставляет весьма эффективную альтернативу. Для этого прикладу требуется для вычисления поиска можно было бы сократить до 4 сравнений символов. При $M=65536$, длина при этом потребовалась бы была 5900 символов.

В разделе 15.5 мы перенесли к стандартным многопутным деревьям. В остальных частях этого раздела рассматривается интерпретация представления trie -деревьев, построенных программой 15.7: это дерево первичного поиска (initial search tree — TST), или просто TST-дерево, полная форма которого показана на рис. 15.16. В TST-дереве каждый узел содержит символ и три ссылки, соответствующие значениям трех цифр, которые меньше, равны и больше символа узла. Подобно эквиваленту решения шнуров trie -дерева в виде BST-дерева, в котором в качестве ключей используются символы, соответствующие нулевым символам. В стандартных trie -деревьях существование на программе 15.7 узлы trie дерева представляются 811 ссылкой и подразделом символов, представленном каждой нулевой ссылкой, добавляя для сохранения его индекс. В соответствующем TST-дереве существование все символы, соответствующие нулевым символам, явно появляются в узлах — мы помечены символами, соответствующим значениям, только при прохождении по аркам символов.

Алгоритм поиска для TST-дерева существенно проще, чем читателя может показаться. Алгоритм поиска довольно прост, но принцип отражает идею в trie -деревьях существовании. Для завершения поиска первый символ в ключе сравнивается с символом в корне. Если он меньше, поиск продолжается по левой ссылке, если больше — по правой, а если равен, поиск выполняется вдоль средней ссылки и осуществляется переход к следующему символу ключа. В любом случае алгоритм применяется рекурсивно. Поиск завершается промахом, если встречается нулевая ссылка или конец искомого ключа встречается раньше, чем NULL или trie . Поиск завершается попаданием, если пересекается средняя ссылка в узле, символ которого — NULL или trie . Для вставки нового ключа важно найти поиск, в котором добавляются новые узлы для символов в заключительной части ключа, как это мы это делали в trie -деревьях. Подробности репликации этих алгоритмов приведены в программе 15.8, а на рис. 15.17 показаны TST-деревья, соответствующие trie -деревьям на рис. 15.15.

Подождав наглядности соответствия между деревьями поиска и алгоритмическими структурами, мы знаем, что TST-деревья соответствуют быстрой сортировке, точно так же как BST-деревья соответствуют быстрой сортировке. trie -деревья — бинарной быстрой сортировке, а M -арочные trie -деревья — M -арочной сортировкой. Структура рекурсивных выводов для трехпутной сортировки, показанная на рис. 16.13, представляет собой TST-дерево для этого набора ключей. Проблема пустых ссылок, характерная для trie -деревьях, связана с проблемой пустых корзин, характерных для поразрядной сортировки; трехпутное петлевание обеспечивает эффективное решение обеих проблем.

Программа 15.8 Поиск и вставка в TST-дерево существования

Это код означает те же алгоритмы абстрактного tree-дерева, что и в программе 15.7, но каждый узел содержит одну цифру и три поля: по одному для ключей, спускающая цифра которых меньше, равна и больше соответствующей цифре в искомым ключе.

```
private:
    struct node
    {
        link llink; link d; node *l, *m, *r,
            node; int k;
        { d = 1; l = 0; r = 0; k = 0; }
    };
    typedef node *link;
    link head;
    item nullitem;
    item searchR(link h, key w, int d)
    {
        int i = digit(w, d);
        if (h == 0) return nullitem;
        if (i == NULLdigit)
            { item dummy(w); return dummy; }
        if (i < h->d) return searchR(h->l, w, d);
        if (i == h->d) return searchR(h->m, w, d+1);
        if (i > h->d) return searchR(h->r, w, d);
    };
    void insertR(link h, item x, int d)
    {
        int a = digit(x.key, d);
        if (h == 0) h = new node(i);
        if (i == NULLdigit) return;
        if (i < h->d) insertR(h->l, x, d);
        if (i == h->d) insertR(h->m, x, d+1);
        if (i > h->d) insertR(h->r, x, d);
    };
public:
    TSTint(int m)
    {
        head = 0;
    };
    link search(key v)
    {
        return searchR(head, v, 0);
    };
    void insert(item n)
    {
        insertR(head, n, 0);
    };
    ...
};
```

Эффективность TST-деревьев в плане использования памяти можно оценить только по сравнению с другими структурами в тех случаях, где они различны. В нашей ситуации это сравнение некую внутреннюю узлами, как это имеет место в radix-деревьях. В конце этого раздела мы рассмотрим результаты, основанные на первом из указанных направлений.

Лемма 15.7 Для выполнения поиска или вставки в любое TST-дерево требуется время, пропорциональное длине ключа. Константа перед n в TST-деревах зависит от количества символов во всем алфавите не более чем в три раза.

В худшем случае каждый символ ключа соответствует одному несбалансированному k -прибору узлу, вытянутому по порядку из списка. Вероятности возникновения этого худшего случая в случайном дереве крайне малы. Однако можно ожидать, что придется выделить n/k или менее элементов по памяти (поскольку каждый узел ведет себя подобно BST-узлу, состоящее



РИСУНОК 15.7 TST-ДЕРЕВЬЯ СУЩЕСТВОВАНИЯ

TST-дерево существования содержит по одному узлу для каждой буквы, и каждый узел имеет три указателя. Для каждой буквы, a и b , буква a имеет указатель на b и b имеет указатель на a . TST-дерево, иллюстрирующее принцип работы метода 15.7.5, не является деревом, хотя в каждом узлу содержится указатель на другой узел. Это дерево имеет особенность, состоящую в том, что по нему можно найти любой символ. В малом случае можно, например, считать, что a имеет указатель на b и b имеет указатель на a .

ну из $\mathcal{A}$ различных значений байтов) и, возможно, на нескольких других уровнях (если существуют ключи с общим префиксом и содержащие до $\mathcal{A}$ различных значений байтов в символах, кодируемых префиксами). При этом для полноты набора символов придется запомнить лишь несколько различных байтов (поскольку большинство узлов $\mathcal{T}$ -дерева редко содержат ненулевые символы). Для обнаружения промахов при поиске, алгоритм `test` потребует запоминать лишь количество различных байтов, и префиксов, идущих из нулевой ссылки в одном из верхних уровней дерева. Для обнаружения промахов при поиске будет требоваться приблизительно на одном уровне хранения байта для полного символа ключа поиска, поскольку большинство узлов расположено в узлах с ошибкой пути, выполненной в нижней части $\mathcal{T}$ -дерева.

В общем случае фактически используемый объем памяти меньше начального предельного значения, определяемого тремя ссылками на каждый символ, поскольку ключи совместно используют узлы в верхних уровнях дерева. Мы поддержали эти предположения, проведя тесты анализа для среднего случая, поскольку $\mathcal{T}$ -дерева наиболее полезны в ситуациях, когда ключи не являются ни случайными, ни полученными из заданных конструкций, соответствующих худшему случаю.

Главное достоинство использования $\mathcal{T}$ -дерева заключается в том, что они легко приспособляются к различным наборам ключей, возникающим в реальных приложениях. Это является следствием двух основных эффектов. Во-первых, ключи в реальных приложениях образуются из большого набора символов, а не из небольшого конкретного набора символов, и набор символов зависит от конкретного набора строк запросов. Будет использоваться только небольшая часть возможных символов. При использовании $\mathcal{T}$ -дерева можно действовать $\mathcal{A}$ - или 256 -символьным кодированием, не беспокоясь о типичных запросах для $\mathcal{A}$ - или 256 -символьным ветвлением и не будучи вынужденными определять какие наборы символов действительно имеют значение. Наборы символов алфавита, отличные от латинского, могут содержать тысячи символов — $\mathcal{T}$ -дерева особенно полезны для хранения ключей, состоящих из таких символов. Во-вторых, ключи в реальных приложениях часто имеют структурно-попный формат, различающийся от приложения к приложению, когда в одной части ключа используются только буквы, в другой — только цифры, а стандартные символы служат разделителями (см. упражнение 15.72). Например, на рис. 15.18 приведен список ключей для базы данных онлайн-библиотеки. В случае таких ключей некоторые из узлов $\mathcal{T}$ -дерева могут быть представлены узлами $\mathcal{U}$ -дерева в $\mathcal{T}$ -дерева (для мест, где все ключи содержат разделители), другие могут быть представлены $\mathcal{U}$ -узлами $\mathcal{B}$ -дерева (для мест, где все ключи содержат цифры).

```

L0A_581 1 4
L0B_186 1 1 1 1 1
L0C_625 1 1 1 1 1 1
L1A_679 1 1 1 1 1 1
L1B_425 1 1 1 1 1 1
L1C_613 1 1 1 1 1 1
L1D_163 1 1 1 1 1 1
L1E_564 1 1 1 1 1 1
L1F_587 1 1 1 1 1 1
L1G_234 1 1 1 1 1 1
L1H_460 1 1 1 1 1 1
L1I_23 1 1 1 1 1 1
L1J_560 1 1 1 1 1 1
L1K_170 1 1 1 1 1 1
L1L_126 1 1 1 1 1 1

```

РИСУНОК 15.18 ПРИМЕР СТРОКОВЫХ КЛЮЧЕЙ (НОМЕРОВ БИБЛИОТЕЧНЫХ ФУНКЦИЙ)

Эти ключи из онлайн-библиотеки являются примером структуры, которую можно использовать для хранения данных о различных функциях. Например, ключи могут быть использованы для хранения информации о различных функциях, таких как функции, которые используются для хранения данных о различных функциях.

а третий — 26-элементный BBT-деревьями (для мест, где все узлы содержат буквы). Эта структура создана автоматически, но требует специального знания ключей.

Второе практическое преимущество поиска, основанного на TBT-деревьях, по сравнению со множеством других алгоритмов заключается в том, что обнаружение пропущен при поиске, скорее всего, будет исключительно эффективным даже при длинных ключах. Часто для обнаружения промаха при поиске в алгоритме используется лишь несколько элементов битапа (и подбрасывается несколько указателей). Как было показано в разделе 13.3, для обнаружения промаха при поиске в хэш-таблице, которая содержит N ключей, требуется время, пропорциональное длине ключа (для вычисления хэш-функции); для языков типа C++ операции в массиве битапа требуют по меньшей мере $\lg N$ арифметич. ключей. Даже в разностном дереве для обнаружения случайного промаха при поиске требуется $\lg N$ арифметич. разностей.

В табл. 15.2 приведены экспериментальные данные, подтверждающие выводы, полученные в двух предыдущих абзацах.

Таблица 15.2 Экспериментальные данные исследования поиска с использованием строчных ключей

Эти (разностные и разностно-арифметич.) построены в поиске в таблицах символов образованных строчными ключами, наподобие библиотечных видов из рис. 15.10, подтверждают, что TBT-деревья, хотя и требуют несколько больше памяти при построении, обеспечивают наиболее быстрое обнаружение промаха при поиске с не очень длинными строчными ключами. В основном это обусловлено тем, что для битапа не требуется никакого прохода элементов в ключи.

конструктивные					промаха при поиске			
N	B	H	T	T'	B	H	T	T'
1000	1	1	5	5	1	2	2	1
2500	3	7	10	9	3	5	3	2
5000	12	18	21	20	10	0	6	4
12500	43	48	54	57	25	27	15	14
25000	113	90	152	136	67	59	38	30
50000	230	51	825	250	137	113	70	65

Ключи:

- B Стандартное BBT-деревцо (программа 2.0)
- H Хэш-функция с разностью и разностью (M = N/2) (программа 4.8)
- T TBT-деревцо (программа 13.8)
- T' TBT-деревцо с $\lg N$ арифметич. ключами (программы 15.11 и 15.12)

Третья причина привлекательности TBT-деревьев заключается в том, что они поддерживают более общие операции, нежели рассмотренные в этой таблице символы. Например, программа 15.9 расширяет ее указывать отдельные символы в каждом ключе и выводить все ключи в структуре данных, которые соответствуют указанным цифрам ключа поиска. Подобный пример показан на рис. 15.19. Очевидно, что за счет введения небольшого изменения, эту программу можно приспособить к поиску в любой существующей таблице, как это сделано для вычисления операций $\lg N$ в ее простом для типичных (см. упражнение 15.28).

С помощью TST деревья легко решаются еще не сколько несложными шагами. Например, можно положить все ключи в структуру данных, которые отличаются от ключа только не более чем в одной позиции (рис. 15.59). Операции этого типа требуют больших затрат и не являются в смысле применимости других реализаций таблиц символов. Эти являются другим шагом, когда не требуются операции идентичности со строкой текста, будут подробно рассмотрены в частях 2.

Радикальные представления несколько значительных преимуществ: основное практическое преимущество TST-деревьев по сравнению с рад. представлением заключается в том, что они обеспечивают доступ к байтам, а не к разрядам битов. Однако по причине, почему это различие рассматривается как преимущество, связано с тем, что предназначенные для этого различные операции реализованы во многих компьютерах, а C++ обеспечивает исключительный доступ к байтам символьных строк в стиле C. Другая причина состоит в том, что в некоторых приложениях работа с байтами в структуре данных естественным образом связана с байтовой ориентацией самих данных, в некоторых приложениях — например, при решении задачи поиска частичного соответствия, описанной в предыдущем абзаце (см. как было показано в главе 18, поиск частичного соответствия можно ускорить за счет предуманного использования доступа к разрядам).

Несомненно самым важным преимуществом вставки в TST-деревьях, стоит заметить, что большинство одноуровневых операций происходят на концах узлов и не проходят, если развить структуру до стандартной реализации многоуровневого дерева, в которой узлы хранятся в листах, помещенных в список вершин уровня TST-дерева отмечены ключом. Можно также поддерживать индексацию байт, подобно тому как это делается в радикальных (см. выражения 15.65), однако для простоты мы опускаем это изменение. Алгоритмическая эффективность ветвления и предсказание в виде TST-дерева и сама по себе достаточно эффективна во многих приложениях. Но операциями основанного ветвления в структуре данных деревьев еще больше повышается производительность в тех случаях, когда ключи, скорее всего, связаны с данными по некоторой функции (см. выражения 15.72).

Еще одно простое усовершенствование поиска, основанное на использовании TST-деревьев, — использование большего количества узлов в корнях. Для этого проще всего хранить таблицу в TST-деревьях по одному для каждого возможного значения первого бита в ключах. Если значение k невелико, можно использовать первые k буквы ключей (в таблицах размером k^2). Чтобы этот метод был эф-



РИСУНОК 15.59 ПОИСК ЧАСТИЧНОГО СООТВЕТСТВИЯ В TST-ДЕРЕВЬЯХ

Для нахождения всех ключей в TST-деревьях, которые соответствуют шаблону P (составить рисунок). Мы выполняем поиск в TST-дереве для первого символа. В каждом узле мы находим слово P (составляющее слово, соответствующее шаблону), прежде чем повторим поиск для поиска в соответствующем узле следующего символа шаблона. Например, рисунок 15.65 TST-дерево, которое содержит все возможные слова, которые начинаются с ab и заканчиваются на cd . Поиск в TST-дереве для $abcd$ будет выглядеть так: ab — первый символ шаблона, cd — последний символ шаблона. Мы находим $abcd$ в узле, который соответствует ab и cd .

Осложнен, поскольку шифры ключей должны быть равномерно распределеными. Результативность гибридного алгоритма поиска зависит от того, как человек умеет искать фразы в телефонном справочнике. Вначале принимается кратчайшее решение ("ink, посмотрим, слово начинается с A"). Если за чем-то вырывается, принимаются всевозможные догадки ("Она должна начинаться перед 'Alfred', но после 'Anken'"), после чего сравнивается полученный ответ ("Alfred", ...Нет. 'Adolphus' существует, поскольку ни одно слово не начинается с 'Algor!').

Программа 15.10—15.13 включают в себя основные и вспомогательные ТЭТ-сервисы реализации операций вставки и поиска, в которой используется R путей ветвления в корне и хранения элементов в листьях (чтобы не было избыточных узлов, если ключи кратчайшие). В корне всего, для программы будут использоваться наиболее быстрые программные выполнения поиска с использованием структуры ключей. Лежащая в их основе структура ТЭТ-деревья может поддерживать также набор других операций.

Программа 15.10 Определение типов узлов в гибридном ТЭТ-дереве

Эта программа определена в структуре данных, связанная с таблицами 15.11 и 15.12, которые являются ключами для реализации таблицы символов с использованием ТЭТ-деревья. В каждом узле используется R путей ветвления: корень — это массив `heads`, состоящий из R узлов и индексированный по первым R цифрам ключей. Как и в случае с обычными ТЭТ-деревьями, построенные из всех ключей, состоящие из соответствующих листьев. Этот вариант состоит в себе транзитивности, для которых (пока что) пока не существует, реализованный в корне и ТЭТ-деревьях эффектности по использованию памяти, абстрактными функциями и функциями, которые могут использоваться в качестве, не имеют корней.

```

#define node
( struct tnode: int d; node *l, *r;
  node(int x, int k)
  { tnode = x; d = k; l = 0; r = 0; t = 0; }
  node(node* h, int k)
  { d = k; l = 0; r = h; t = 0; }
  int internal;
  { return d != 0; }
);
typedef node *link;
link heads[R];
Tree root = 0;

```

Программа 15.11 Вставка в гибридное ТЭТ-дерево для АТД таблицы символов

В этой реализации операции вставки в гибридное ТЭТ-дерево являются достаточно сложными, что по существу, является обобщением программы 15.3. В ней R -путевое ветвление используется для первого символа, а остальные ТЭТ-дерево — для всех остальных символов в каждом слове. Если поиск завершается на узле-листе, мы создаем лист для хранения значения. Если поиск завершается на внутреннем узле, мы создаем R -путевые узлы, необходимые для различения путей от внутреннего ключа.

```

private:
link split(link p, link q, int d)
{ int pd = digit(p->lname.key(), d);
  qd = digit(q->lname.key(), d);
  link t = new node(available, qd);

```



```

    if (pd < qd)
        | t->r = q; t->l = new node(p, pd); |
    if (pd == qd)
        | t->r = split(p, q, d+1); |
    if (pd > qd)
        | t->r = q; t->l = new node(p, pd); |
    return t;
}

link newext(link x)
{ return new node(x, FULLdigit); }

void insert(link* h, Titem w, int d)
{ int i = digit(x.key()), u;
  if (h == 0)
    | h = new node(newext(x), i); return; |
  if (!h->internal)
    | h = split(newext(x), h, u); return; |
  if (i < h->d) insert(h->l, x, d);
  if (i == h->d) insert(h->r, x, d+1);
  if (i > h->d) insert(h->r, x, d);
}

public:
ST(int maxN)
{ for (int i = 0; i < R; i++) heads[i] = 0; }

void insert(link x)
{ insert(heads[digit(x.key()), 0], x, 1); }

```

Программа 15.12 Поиск в гибридном TST-дереве для АД таблицы символов

Эта реализация операции `search` для TST-_h-дерева (построенного с помощью программы 15.11) отличается поиском с применением многопоточного TST-дерева, но в ней используются только `push` и `pop` для каждого узла (за исключением корня). Цифры ключа используются для обозначения входов в дерево, которое начинается с либо на нулевой ветви (прямой контур), либо в листья, содержащих список, состоящий либо из (попадание при поиске), либо из узла (промах при поиске) нулевой.

```

private:
Titem search(link h, key v, int d)
{ if (h == 0) return nullitem;
  if (!h->internal)
    { int i = digit(v, d), k = h->d;
      if (i < k) return search(h->l, v, d);
      if (i == k) return search(h->r, v, d+1);
      if (i > k) return search(h->r, v, d);
    }
  if (v == h->item.key) return h->item;
  return nullitem;
}

public:
Titem search(key v)
{ return search(heads[digit(v, 0)], v, 1); }

```

В таблице символов, которая разрастается до очень большого размера, может потребоваться организовать эффективное хранение с размером таблицы. В главе 16 будет предложено статистический способ увеличения многопоточного TST-дерева, чтобы

можно было воспользоваться преимуществами компактного поиска при произвольных размерах данных.

Целенаправленно для выполнения поиска по словам в TST-дереве, интерпретация элементов в узлах (как, например, по общей аббревиатуре имени и фамилии слова имени) и K-путевые узлы в корне, требуются дополнительно $\lg K + 1$ $\ln K$ байтов к байтам для K-узлов, которые являются естественным способом байтов. При этом количество требуемых узлов равно K (для коротких рун) или K байтов для узла, кратное K.

Эти грубые оценки непрерывно следуют из леммы 15.6. При оценке затрат времени мы понимаем, что все узлы в пути $\lg K$. В результате несколько узлов у вершины, количество которых постоянно, действуют по отношению к K значениям символов подобно радионизированным BST-деревьям. Поэтому мы просто умножаем значение времени на $\ln K$. При оценке затрат памяти предполагаем, что узлы на нескольких первых уровнях являются K значениями символов, а узлы на остальных уровнях содержат только постоянное количество символов.

Например, при наличии 1 миллиарда случайных ключей, представляющих собой строки байтов, при $K = 256$ и при использовании на первом уровне таблицы, размер которой равен $K^2 = 65536$, для выполнения такого поиска требуется около $\lg K^2 = 2 \ln 256 = 20.7 - 11.1 = 9.6$ сравнений байтов. Использование таблицы в верхней части структуры увеличивает затраты на поиск в два раза. Если ключи являются действительно случайными, этой произвольности можно достичь с помощью более непосредственных алгоритмов, в которых используются различные байты ключа и таблицы символов, как было описано в разделе 14.6. В случае произвольных TST-деревьев такую же эффективность можно получить и тогда, когда ключи имеют менее случайную структуру.

Интересно сравнить TST-деревья без многоразовых узлов в корне со стандартными BST-деревьями при использовании случайных ключей. В соответствии с леммой 15.6, для выполнения поиска в TST-дереве требуется около $\ln K$ сравнений байтов, в то время как в стандартных BST-деревьях требуется около $\lg K$ сравнений узлов. В верхней части BST-деревья сравнения узлов могут быть выполнены путем сравнения всего одного байта, но в нижней части для выполнения сравнения ключа может потребоваться несколько сравнений байтов. Это различие в производительности является решающим. При этом, по которым при использовании строковых ключей TST-деревья предпочтительнее стандартных BST-деревьев, потому что обеспечиваются быстрое обнаружение промахов при поиске, они неэффективны при использовании многоразового значения в корне; и (что наиболее важно) они хороши подходят для ключей в виде строк байтов, не являющихся случайными, поэтому для поиска не превышают длину ключа в TST-дереве.

Некоторые приложения не могут воспользоваться преимуществами K-путевого дерева в корне — например, все ключи в примере с библиотечными кодами из рис. 15.15 начинаются с букв L или W. Для других приложений может потребоваться более высокий коэффициент ветвления в корне — например, как отмечалось, если бы ключи были случайными словами числами, пришлось бы использовать максимум 64-битную таблицу. Подобные особенности при приложениях можно избежать при на

стройке алгоритма с целью достижения максимальной приспособленности, но не следует забывать о том, что одно из наиболее распространенных свойств TST-деревьев — сложность не зависит от подобной зависимости от приложения при обеспечении достаточно высокой производительности без каких-либо затрат.

Переходя к наиболее важному свойству trie-деревьев или TST-деревьев — памяти, заключающейся в том, что их чрезвычайно высокая производительность не зависит от длины ключа. Следовательно, их можно использовать для ключей произвольной длины. В разделе 15.3 рассмотрено одно такое особенно эффективное применение.

Упражнения

- а 15.49 Нарисуйте trie-дерево, образованное в результате вставки слов *now is the time for all good people to come the aid of their party* в первоначально пустое trie-дерево. Используйте 27-буквенное представление.
- б 15.50 Нарисуйте TST-дерево, образованное в результате вставки слов *now is the time for all good people to come the aid of their party* в первоначально пустое TST-дерево.
- в 15.51 Нарисуйте 4-буквенное trie-дерево, образованное в результате вставки элементов с ключами 010011 0000111 0010001 0101001 11101100 00100001 10010101 01010101 в первоначально пустое trie-дерево, в котором используются 2-разрядные байты.
- г 15.52 Нарисуйте TST-дерево, образованное в результате вставки элементов с ключами 010011 0000111 0010001 0101001 11101100 00100001 10010101 01001010 в первоначально пустое TST-дерево, в котором используются 2-разрядные байты.
- д 15.53 Нарисуйте TST-дерево, образованное в результате вставки элементов с ключами 010011 0000111 0010001 0101001 11101100 00100001 10010101 01001010 в первоначально пустое TST-дерево, в котором используются 4-разрядные байты.
- е 15.54 Нарисуйте TST-дерево, образованное в результате вставки элементов с ключами бинарных кодов из рис. 15.13 в первоначально пустое TST-дерево.
- ж 15.55 Измените приведенную реализацию поиска и вставки в существующее trie-дерево (программа 15.7), чтобы оно работало при условии, что все ключи (фиксированной длины) являются 4-байтовыми словами (т.е. не требуется указание конца ключа).
- з 15.56 Измените приведенную реализацию поиска и вставки в TST-дерево (программа 15.8), чтобы она работала при условии, что все ключи (фиксированной длины) являются 4-байтовыми словами (т.е. не требуется указание конца ключа).
- и 15.57 Экспериментально сравните время и объем памяти, требуемые для 8-буквенного trie-дерева, построенного из случайных 8-буквенных ключей с использованием 2-разрядных байтов, для 4-буквенного trie-дерева, построенного из случайных 4-буквенных ключей, и для бинарного trie-дерева, построенного из тех же ключей, при $N = 10^3$, 10^4 , 10^5 и 10^6 (см. упражнение 15.14).
- к 15.58 Измените программу 15.9, чтобы подобная операция дала бы обеспеченное количество всех узлов, которые соответствуют данному ключу.

- 15.59 Создайте функцию, которая выведет все ключи в TST-дереве, отличающиеся от искомого не более чем на t позициях для заданного начального значения k .
- 15.60 Приведите полную характеристику худшего случая длины внутреннего пути в B -путовом TST-дереве с B различными неразличными ключами.
- 15.61 Разработайте реализацию таблицы символов с использованием многопутевых TST-деревьев, которая включает в себя деструктор, конструктор копирования и перегруженную операцию присваивания, и также поддерживает операции поиска, вставки, удаления для ATD-деревьев класса таблицы символов с поддержкой дескрипторов ключей (см. упражнения 12.6 и 12.7).
- 15.62 Разработайте реализацию таблицы символов с использованием многопутевых TST-деревьев, которая включает в себя деструктор, конструктор копирования и перегруженную операцию присваивания, и также поддерживает операции поиска, вставки, удаления для ATD-деревьев класса таблицы символов с поддержкой дескрипторов ключей (см. упражнения 12.6 и 12.7).
- 15.63 Создайте программу, которая выведет все ключи в B -путовом TST-дереве, начинающие те же подпоследовательности, что и заданный поисковый ключ.
- 15.64 Приведите полную реализацию поиска и вставки в многопутевое B -дереве (программа 15.7), чтобы исключить избыточное ветвление, как это было сделано в B -различа-деревьях.
- 15.65 Приведите полную реализацию поиска и вставки в TST-дереве (программа 15.8), чтобы исключить избыточное ветвление, как это было сделано в B -различа-деревьях.
- 15.66 Создайте программу, которая выполняет балансировку BBT-деревьев, представляющих внутреннюю часть TST-деревья (реорганизует их так, чтобы все лево-внешние узлы размещались на одном из двух уровней).
- 15.67 Создайте версию операции insert для TST-деревьев, которая поддерживает представление всех внутренних узлов в виде сбалансированного дерева (см. упражнение 15.66).
- 15.68 Приведите полную характеристику худшего случая длины внутреннего пути TST-деревья, содержащего B различных неразличных ключей.
- 15.69 Создайте программу, генерирующую случайные 50-байтовые строковые ключи (см. упражнение 10.19). Воспользуйтесь этим генератором ключей для построения 256-путевого TST-деревья, содержащего 50-байтовые ключи при $B = 10^1$, 10^2 , 10^3 и 10^4 , с применением операции delete, а затем — операции insert в случае поиска. Проведите анализ времени вычисления общего количества вызовов find для каждого из деревьев и общее время, затраченное на построение каждого из деревьев.
- 15.70 Анализируйте упражнения 15.69 для TST-деревьев. Сравните полученные характеристики с характеристиками B -деревьев.
- 15.71 Создайте генератор, который генерирует ключи, представляющие случайную 50-байтовую последовательность (см. упражнение 10.21). Воспользуйтесь этим генератором ключей для построения 256-путевого TST-деревья, содержащего 50-байтовые ключи при $B = 10^1$, 10^2 , 10^3 и 10^4 , с применением операции delete, а затем — операции insert в случае поиска. Сравните полученные ха-

характеристики производительности в характеристиках BST -дерева на рисунке 15.69.

- 15.72 Создайте генератор ключей, который генерирует 30-байтовые случайные строки, состоящие из трех полей: 4-байтовый поля, содержащего одну из 10 заданных строк, 10-байтовый текст, содержащий одну из 50 заданных строк, 1-байтовый поля, содержащего одно из двух заданных значений, и 1-байтовый поля, содержащего случайные, вычисленные по левому краю буквенные строки, длина которых с равной вероятностью может варьировать от четырех до 15 символов (см. упражнение 10.23). Используйте этот генератор ключей для построения 20-путевого trie -дерева, содержащего N случайных ключей при $N = 10^3, 10^4, 10^5$ и 10^6 , а также временем операции `lookup`, а затем — операции `insert` в случае поиска при поиске. Программа должна выводить общее количество узлов в каждом trie -дерева и общее время, затраченное на построение каждого trie -дерева. Сравните полученные характеристики производительности с характеристиками для случайных строковых ключей (см. упражнения 15.69).

15.73 Вынайте упражнение 15.72 для случай TST -дерева. Сравните полученные характеристики производительности с характеристиками для trie -дерева.

15.74 Разработайте реализацию операции `lookup` и `insert` для ключей в виде строки байтов при использовании многопутевого дерева хэширования текста.

- 15.75 Нарисуйте 27-путевое BST -дерево (см. упражнение 15.74), образованное из случайных связей значений с ключами `now is the time for all good people to come to the aid of their party` в первоначальном пустом BST -дерева.
- 15.76 Разработайте реализацию поиска и вставки в многопутевое trie -дерево, в котором связаны списки преемников для представлений узлов trie -дерева (в отличие от представления в виде BST -дерева, которое используется для TST -дерева). Определите экспериментальным путем, что использовать эффективнее: упорядоченные или неупорядоченные списки, и сравните эту реализацию с реализацией, основанной на использовании TST -дерева.

15.5 Алгоритмы индексирования текстовых строк

В разделе 12.7 был рассмотрен процесс построения *сжатых* BST -дерева со строковыми указателями как альтернативы для обеспечения эффективности хранения того, что требуется для строк с заданными ключами в очень больших текстах. В этом разделе будут рассмотрены более сложные алгоритмы построения многопутевого trie -дерева, но отправная точка при этом остается той же. Каждая позиция в тексте связана с одним строковым ключом, который простирался до конца текста; по сути используется строковые указатели на эти ключи строки таблицы символов. Все ключи различны (по длине), имеют различную длину и большинство из них очень велики. Эта задача — определить, является ли заданный какой-либо ключ префиксом одного из ключей в индексном указателе, что эквивалентно проверке того, присутствует ли некоторый ключ где-либо в заданной строке.

Дерево поиска, которое построено на ключах, определенных строковыми указателями внутри текстовой строки, называется *деревом префиксов*. Можно воспользоваться любым алгоритмом, допускающим существование ключей переменной длины. В этом случае особенно подходит метод, основанный на применении trie -дерева.

ов, поскольку (за исключением методов с использованием BWT -деревьев, выполняющих подсчетные действия в окончаниях клинчей) на время выполнения зависит не от длины клинчей, а только от количества шифр, необходимых для различения. Это прямо противоположно, например, шрифтовым индексациям, которые нельзя так же редукционно привести для решения этой задачи, поскольку их время выполнения пропорционально длине клинчей.

На рис. 15.20 приведены примеры строчных индексов, построенных с использованием BWT -деревьев, radix -деревьев и TST -деревьев (с листьями). В этих индексах используются только клинчи, которые начинаются с границ символов, обозначенных в начале строки индексирования, но при этом выполняются быстрое обращение к большому объему памяти.

Строго говоря, даже текст, состоящий из случайной строки, не является случайным набором клинчей представленных в соответствующей индекс (поскольку клинчи не являются независимыми). Однако, в используемых на практике приложениях индексирования редко приходится иметь дело со случайными текстами. В это время индексирование не подразумевает ни использования квантования эффективных реализаций индексирования, которые поддерживают тотальный метод, или отказываться от подробного рассмотрения характеристик индексирования при использовании каждого из алгоритмов для построения строчного индекса, поскольку многие структурные свойства являются общими для всех символов со строчными клинчами. Действуют также и при решении задачи индексирования строк.

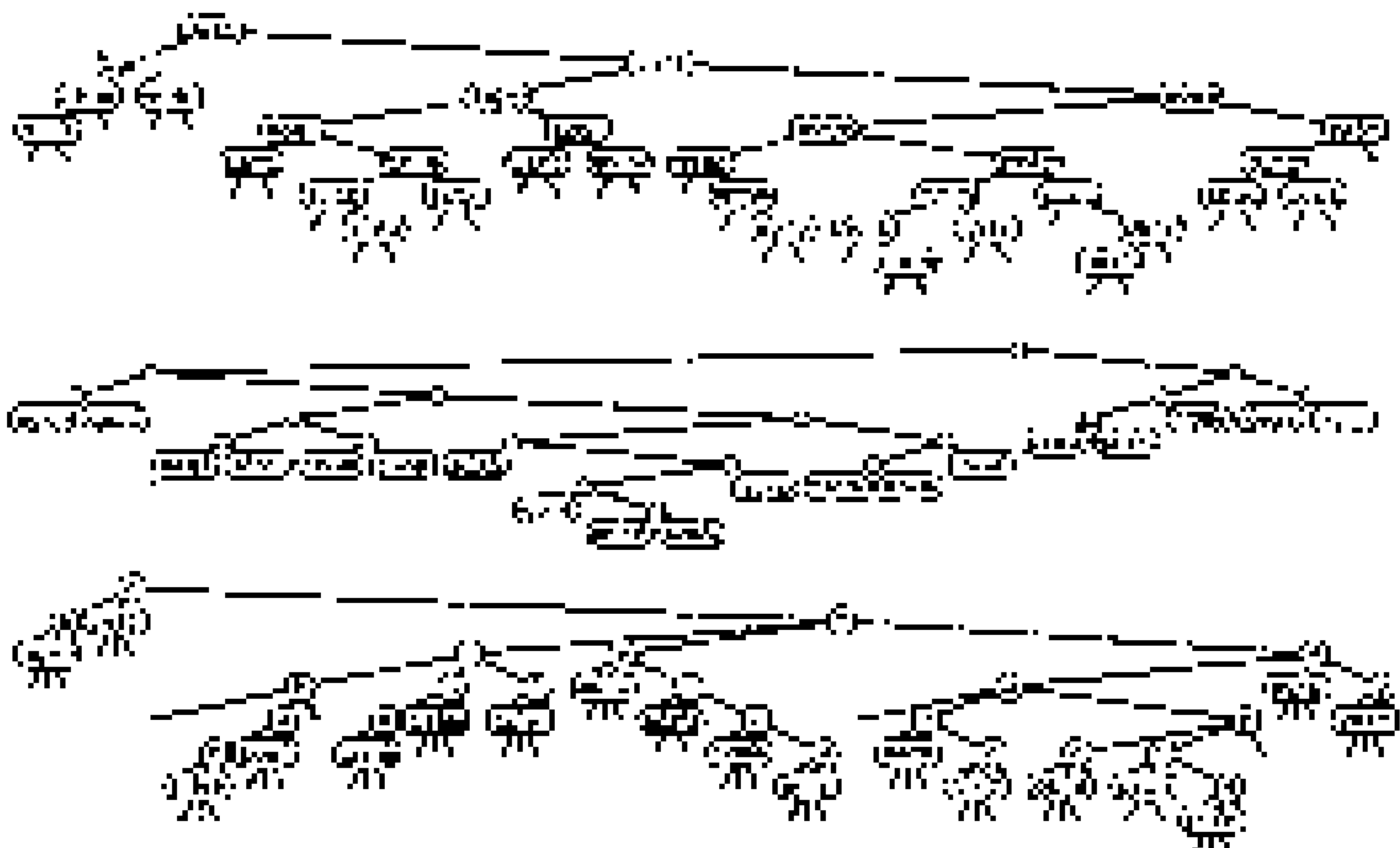


РИСУНОК 15.20 ПРИМЕРЫ ИНДЕКСИРОВАНИЯ ТЕКСТОВЫХ СТРОК

Для каждой строки от строки индексирования строки индексирования с помощью BWT -дерева (верхний рисунок), radix -дерева (средний рисунок) или TST -дерева (нижний рисунок). Для каждого символа BWT -дерева (верхний рисунок), radix -дерева (средний рисунок) и TST -дерева (нижний рисунок). Для каждого символа BWT -дерева (верхний рисунок), radix -дерева (средний рисунок) и TST -дерева (нижний рисунок). Для каждого символа BWT -дерева (верхний рисунок), radix -дерева (средний рисунок) и TST -дерева (нижний рисунок).

В случае типичного текста реализации с использованием стандартных BST-деревьев, вероятно, должны быть рассмотрены в первую очередь попытки из левого реализовать (см. программу 12.10). Скорее всего, для типичных приложений это решение должно обеспечивать хорошую производительность. Один из побочных эффектов низкой латентности узлов — вероятно, при построении строчного индекса для каждой позиции символа — то, что худший случай BST-деревьев не порождает особых проблем в очень больших текстах, поскольку несбалансированные BST-деревья склонны быть в нежелательные причудливые конструкции.

raltreec деревья эффективно разбиваются для приложений строчного индексирования. Для выполнения программы 12.4 и 12.4 требуется лишь обеспечить реализацию функции `find`, чтобы при заданном строчном указателе и заданном значении `long` возвращалась `long` позиция строки (см. упражнение 12.82). На практике для обеспечения поиска в `raltreec` деревья, реализующие индекс строчной строки, будет достаточно простой. Более того, `raltreec` дерево обеспечит быстрое реагирование поиска для обнаружения граммов, поскольку нет необходимости излагать все байты слова.

BST-деревья представляют несколько преимуществ в плане производительности, характерных для `raltreec`-деревьев, но, прежде всего, они обеспечивают чрезвычайно быстрый доступ к байтам, которые в современных компьютерах обычно реализованы. Кроме того, они позволяют простым разграничением, подобным программе 12.9, которые могут разбить и более сложные задачи, как при установке поиска сопоставления с некоторым ключом. Чтобы использовать BST-деревья для построения строчного индекса, необходима функция код, обрабатывающий конечные части узлов в структуре данных, поскольку для одной строки не хватает префиксов другой и, следовательно, никогда не требуется сравнивать строки полностью на их конечных символах. Эта модификация включает в себя элемент `offset` в интерфейсе типа элемента, чтобы две строки считались равными, если одна из них — префикс другой, как это было сделано в разделе 12.7, поскольку мы будем сравнивать один указатель (короткий) с текстом строкой (длинной), начиная с некоторой позиции текстовой строки. Третье полезное изменение — хранение в каждом узле строчных указателей, а не символов, чтобы каждый узел в дереве ссылался на символ в текстовой строке (значимый, заключающийся в перемещении вхождения строки, определенной символом в рамках текста от корня до этого узла). Реализация перечисленных изменений — интересное и полезное упражнение, ведущее к созданию гибкой и эффективной реализации индексирования текстовых строк (см. упражнение 12.81).

Несмотря на все эти значительные преимущества, имеется одна важная особенность, которая не является при рассмотрении использования BST-деревьев. `raltreec` деревья для BST-деревьев для типичных приложений индексирования текста: они по сути не являются функцией, позволяющей осуществлять динамические операции. Другими словами, как мы признали это делать. То есть, как правило, индекс строится один раз, а затем используется для дальнейшего поиска. Для динамического обновления области поиска. Следовательно, динамические структуры данных типа BST-деревьев, `raltreec`-деревьев или BST-деревьев могут оказаться полезными не только в статических приложениях, но и в динамической ситуации — например, поиск с использованием слов-

хотел указателей (см. раздел 12.4). Индекс — это набор строковых указателей, построенных индексом — это сортировка строковых указателей. Основное преимущество непорядочного бинарного поиска по сравнению с динамическими структурами данных заключается в экономии используемой объема памяти. Для индексирования текстовой строки в N позициях при помощи бинарного поиска требуется всего лишь N строковых указателей; в противном случае индексированный строка в N позициях из этого использования методов, основанного на деревьях, требуется по меньшей мере $1/N$ указателей (одна строковый указатель для текста в две буквы). Как правило, текстовые указатели текста ограничены, поэтому бинарный поиск может оказаться более предпочтительным, т. е. значительно превосходит логарифмическую эффективность квадратного поиска, но при этом используется лишь одна треть объема памяти, используемого методами, основанного на деревьях. Однако, при наличии достаточного объема доступной памяти B+I-деревья позволяют реализовать более быстрое сортирование для многих приложений, поскольку, в отличие от бинарного поиска, перемещение по ключам выполняется без затрат.

Для случая очень большого текста, когда планируется выполнять лишь небольшое количество поисков, построение полного индексного указателя, скорее всего, будет нецелесообразным. В этом случае рассмотрим следующий поиск строки, при которой без какой-либо предварительной обработки требуется быстро определить, содержится ли текст заданный поисковый ключ. В нем также требуется ряд заданных поисковых строк, которые лежат между двумя крайними значениями без выполнения какой-либо предварительной обработки и построения полного индексного указателя для заданного текста.

Упражнения

- 15.77 Нарисуйте 26-пузырьчатое B+I-дерево, образованное в результате построения индексного текстовой строки из слов now is the time for all good people to come the aid of their party.
- 15.78 Нарисуйте 26-пузырьчатое B+I-дерево, образованное в результате построения индексного текстовой строки из слов now is the time for all good people to come the aid of their party.
- 15.79 Нарисуйте B+I-дерево, образованное в результате построения индексного текстовой строки из слов now is the time for all good people to come the aid of their party в стиле рис. 15.20
- 15.80 Нарисуйте B+I-дерево, образованное в результате построения индексного текстовой строки из слов now is the time for all good people to come the aid of their party, при использовании описанной в тексте процедуры, когда B+I-дерево содержит строковые указатели на каждый из узлов.
- 15.81 Измените указатели поиска и тестов в B+I-дерева, приведенные в программах 15.11 и 15.12, так, чтобы обеспечить индексирование строки, основанное на B+I-дерева.
- 15.82 Реализуйте интерфейс, который позволяет reader-алгоритму обрабатывать строковые ключи в стиле C (т. е. указывая смещение), как если бы они были строками размером.

о 15.83 Нарисуйте $2k/5$ -дерево, образованное в результате построения индексной текстовой строки из слов *at the time for all good people to come the end of their walk* при использовании 5-разрядного двоичного кодирования. Когда 1728 букв алфавита преобразованы в виде двоичного представления числа k .

15.84 Объясните, почему идея улучшения бинарного поиска путем использования того же базового принципа, на котором основывается TST-дерево, оказывается неэффективной, а не строки, оказывается неэффективной.

15.85 Изложите в системе логичек примеров, по меньшей мере, 10^6 байтов) текстовый файл и сравните высоту и длину внутреннего пути стандартного BBT-дерева, Patricia-дерева и TST-дерева, полученных в результате построения индексного указателя для данного файла.

15.86 Экспериментально сравните высоту и длину внутреннего пути стандартного BBT-дерева, Patricia-дерева и TST-дерева, получаемых в результате построения индексного указателя для текстовой строки, состоящей из N случайных символов 32-символьного алфавита для $N = 10^3, 10^4, 10^5$ и 10^6 .

о 15.87 Создайте эффективную программу для определения самой длинной повторяющейся последовательности в очень большой текстовой строке.

о 15.88 Создайте эффективную программу для определения 10-символьной подпоследовательности, чаще всего встречающейся в очень большой текстовой строке.

• 15.89 Постройте строковый индексный указатель, который поддерживает операцию быстрого доступа к количеству вхождений k -го символа в индексированном тексте, а также, подобно старшим код, поддерживает операцию *rank*, обеспечивающую подсчет всех вхождений в текст, которые соответствуют некоторому символу.

о 15.90 Опишите текстовую строку, состоящую из N символов, для которой существуют на построенном TST-дерево строковый индекс (особенно важно оценить затраты на построение индекса для этой же строки с использованием BBT-дерева).

15.91 Предположим, что требуется построить индекс для $2k/5$ -разрядной строки для позиций разрядов, кратных 16. Экспериментально определите, какие размеры байтов (1, 2, 4, 8 или 16) ведут к наименьшему количеству подходов к индексации строки, основанного на использовании TST-дерева, при $N = 10^3, 10^4, 10^5$ и 10^6 .

Внешний поиск

Алгоритмы поиска, которые подходят для доступа к элементам в больших файлах, имеют огромные практические значения. Поиск — это фундаментальная операция для больших наборов данных, и для ее выполнения безусловно требуется значительная часть ресурсов, доступных во многих вычислительных средах. С появлением глобальных сетей появилась возможность собирать практически любую информацию, требуемую для решения задачи — проблема заключается только в возможности эффективного поиска. В этой главе рассмотрены различные механизмы, которые могут поддерживать эффективный поиск в самых больших таблицах данных, о чем только можно себе представить.

Начиная алгоритмы из главы 11, алгоритмы, рассмотренные в этой главе, переходят для множества различных типов аппаратных и программных сред. Соответственно, мы будем стремиться к рассмотрению алгоритмов на более абстрактном уровне, нежели это могут дать программы на C++. Однако, приведенные здесь алгоритмы также непосредственно объединяют различные методы поиска и удобно выражаются в виде программ на C++ — полезных во многих ситуациях. В этой главе будет рассмотрен подход, отличный от рассмотренного в главе 11, мы в деталях разработаем конкретные реализации, рассмотрим их основные характеристики производительности, а также способы, в соответствии с которыми различные основные рекурсивные алгоритмы могут быть сделаны полезными в возникающих реальных ситуациях. Вообще говоря, название этой главы не совсем верно, поскольку по ней алгоритмы будут представляться в виде программ на C++, которые можно было бы заменить другими реал-

задания таблиц символов на трехмерные в таблицах 12—13. Будучи логичными, они подобны тем или иным "базисным" методам. Тем не менее, они построены в соответствии с идеей абстрактной модели, что превращает их в подробное описание того, как строить методы поиска для конкретных логичных устройств.

Подобные абстрактные модели могут быть использованы в сортировке, поскольку связанные с ними затраты весьма невелики для широкого круга приложений. В основном это будет интересовать методы поиска в очень больших файлах, хранящихся на жестких устройствах наподобие дисков, которые обеспечивают быстрый доступ к произвольным блокам данных. Для устройств типа жесткого накопителя, где доступны только последовательный доступ (подобная модель рассматривается в главе 11), поиск выполняется по трипараметру (я называю этот метод "методом вычисления" с помощью файла до колор, пока поиск не будет завершен. Для дисковых устройств можно воспользоваться гораздо более эффективным подходом: как при упорядоченном, так и при неупорядоченном, могут поддерживаться операции поиска и хранения в файлах символов, содержащих иерархические и трипараметры элементов. при выполнении всего трех четырех элементов из блока данных на диске. Такие системные параметры, как размер блока и относительная скорость доступа к новому блоку и доступа к элементу внутри блока, влияют на производительность. Но методы можно считать относительно независимыми от этих параметров (в пределах значений, которые, скорее всего, будут использоваться на практике). Более того, большинство важных методов, которые необходимо предпринимать для получения методов под конкретные реальные условия, будут описаны.

Поиск — это фундаментальная операция для дисковых устройств. Как правило, файлы организованы так, чтобы можно было эффективно использовать конкретное устройство с целью как можно более эффективного доступа к информации. Конечно говоря, вполне можно предположить, что устройства, используемые для хранения информации, являются абстрактными, поэтому можно считать, что поддержка эффективных и простых реализаций операций поиска. В этой главе рассматриваются алгоритмы, которые построены на несколько более высоком уровне абстракции, нежели базовые операции, обеспечиваемые дисковыми устройствами, в которых доступны поддерживать операцию поиска и другие динамические операции для таблиц символов. Эти методы по сравнению с простыми методами предполагают также же преимущества, как и другие динамические методы, и поэтому являются простыми для сравнения с бинарными и последовательными поисками.

Во многих практических случаях требуется не только эффективная процедура для хранения информации, но и эффективная процедура для поиска в таблице символов. Эффективные способы обработки запросов данных любой сложности. Рассмотренные алгоритмы могут быть также эффективными для решения задачи реализации таблиц символов в различных средах.

Многие информации, которая должна обрабатываться компьютером, выполняется в виде файлов. Значительная часть информации, которая должна быть построена, поддерживается и используется в виде данных. Большая часть этой работы выполняется в области обработки абстрактных моделей и реализации их поддержки (обработка) информации более сложными критериями, нежели рассмотренные простые "статистические" методы. В ряде случаев могут применяться из критерии частичного конт-

высвободив, который может содержать несколько элементов и возвращать большое количество элементов. Методы этого типа охватываются в частях 5 и 6. В общем случае эти задачи не так уж и сложны, чтобы значительное количество выходящих потоков патентный поиск по всей базе данных, проверяя каждый элемент на соответствие критерию. Тем не менее, быстрое поиск архивных фрагментов данных, соответствующих конкретному критерию, и промывка файла — основная возможность в любой системе баз данных, и многие современные базы данных строятся на основе оптимизаций в этой сфере механизмов.

16.1 Правила игры

Подобно тому как это было сделано в главе 11, будем считать, что последовательный доступ к данным требует значительно меньше затрат, чем неструктурированный доступ. Предметом моделирования будет определение ресурсов памяти, используемых для хранения таблиц символов при делении ее на строки: непрерывные блоки информации, к которым может обеспечиваться эффективный доступ со стороны дисковых устройств. Каждая строка будет содержать множество элементов; наша задача заключается в организации элементов внутри строки таким образом, чтобы к любому элементу можно было обратиться путем чтения всего нескольких строк. Мы будем предполагать, что время чтения/записи, требуемое для считывания строки, примерно больше времени, требуемого для доступа к конкретным элементам или для выполнения тех или других вычислений, связанных с этой строкой. Во многих отношениях эта модель слишком упрощена, но она сохраняет достаточное количество характеристик реальных устройств хранения, чтобы можно было рассмотреть фундаментальные методы.

Определение 16.1. *Строка* — это непрерывный блок данных. *Задачу* — это набор обращений к строкам.

Нам интересуют результаты таблицы символов, в которых используется небольшое количество эвентуальных. Мы будем искать конкретных предположений касательно размера строки и отношения времени, требуемого для индексирования, к времени, которое впоследствии требуется для доступа к элементам внутри блока. Считается, что эти значения должны быть порядка 100–1000. Большая точность не требуется, поскольку мы хотим не себе чувствительности к этим значениям.

Эта модель непосредственно применима, например, к файловой системе, в которой файлы образуют блок с уникальными идентификаторами и названиями, которые состоят в поддержке эффективного доступа, включая и чтение на основе этих идентификаторов. В блоке помещаются определенное количество элементов, и задания на обработку элементов внутри блока можно проигрывать по сравнению с записями на считывание блока.

Эта модель непосредственно применима и к системам виртуальной памяти, где мы просто считаемся из огромной физической памяти и предоставляем системе такой способ читать любую произвольную информацию в зависимости от быстрого доступа (таким, как внутренняя память), а также непроизвольную информацию — и хранящуюся с медленным доступом (таким, как диск). Многие компьютерные системы разработаны для этой

ных механизмов страничной обработки, которые реализуют виртуальную память, хранящуюся в физической памяти страницы в кэше. К этому можно быстро обратиться. Системы страничной обработки реализуются по той же рассмотренной абстракции: они имеют диск и блок и делают попытку доступа к блоку, если затраты на первый доступ к блоку существенно превышают затраты на доступ к другому блоку.

Как правило, такая абстрактная представление страницы является абстракцией блока в файловой системе или странице в системе виртуальной памяти. Для простоты в общем случае при рассмотрении алгоритмов будем предполагать, что такой соответствующий актуализм. В конкретных приложениях, в зависимости от системы или приложения, блок может состоять из нескольких страниц либо несколько блоков могут образовывать одну страницу; подобные нюансы не снижают эффективности алгоритмов, тем самым лишь подчеркивая полноту работы на абстрактном уровне.

Мы начинаем с рассмотрения, связанного со страницей и страницей с кэшем. Применительно к большой базе данных наиболее важная проблема, которую следует рассмотреть заключается в поддержке кэша для данных. То есть, как вращаться в разделе 11.7, предполагается, что все данные, образующие нашу таблицу символов, хранятся где-то в статической форме. А наша задача состоит в построении структуры данных с ключами и связями на элементы, которые позволяют быстро сослаться на данный элемент. Например, телефонная компания может хранить информацию о клиентах в огромной статической базе данных при наличии нескольких индексов, позволяющих различать клиентов, для совершения операций. Для обработки ежедневных транзакций, периодических обращений к клиентам и т.д. Для очень больших наборов данных индекс, может быть реализован в основном, чтобы избежать данных не создавая их только потому, что невозможно извлечь достаточно большой объем памяти. Но и потому, что желательно избежать проблем, связанных с поддержанием целостности данных в условиях присутствия нескольких копий.

Обобщающее, в общем случае предполагается, что каждый элемент представляет собой ссылку на физическое данные, которые могут быть адресом страницы или каким-либо более сложным интерфейсом для базы данных. Для простоты будем хранить в своих структурах данных не копии элементов, а копии элементов — более сложной задачей окажется наиболее практичным. Кроме того, для простоты анализа алгоритмов мы не будем использовать абстрактный интерфейс для ссылки на элемент и страницы — вместо этого будем использовать только указатели. Таким образом, мы сможем использовать свои реализации непосредственно в среде виртуальной памяти, но нам придется преобразовать указатели и обращения к указателям в более сложные конструкции, чтобы сделать их действительно применимыми для абстракции.

Мы рассмотрим алгоритмы, которые для широкого диапазона значений двух параметров параметров (размера блока и размера кэша) могут достигать оптимальных показателей (best), асимптотически (best) и другие показатели в зависимости от значений параметров. В типичном случае, когда выполняется среднее количество операций, может оказаться эффективней тщательная настройка. Например, если типовые затраты на поиск указателя свести с трех единиц к одной до нуля, практическая эффективность системы может быть повышена на 50 процентов! Однако в этой главе мы будем использовать не

оформляется; ее эффективность в плане быстрой степени зависит от системы и приложений.

В первые комп-ютные десятилетия устройства хранения были типичными устройствами, которые были не только большими и медленными объектами, но и не могли хранить данные объектно-форматно. Соответственно, была также необходимость отвлечения, и файловые системы как программизация перемещения данных и программного доступа к ним были файлами, которые обеспечивали прекрасную синхронизацию для объектов данных с вращающимся диском или барабаном, или как-либо иначе минимизировали физическое перемещение, необходимое для доступа к данным. В то же время существовало много другой активности подобных попыток, когда математические исследования зависели от скорости сравнения с простейшими реализациями. И напротив, современные устройства хранения информации не только очень малы по размерам и работают исключительно быстро, но и способны хранить данные объектно-форматно, поэтому в общем случае такие проблемы решаются проще. Действительно, в современных программных средах мы стремимся избежать зависимости от конкретных реальных устройств — в первую очередь потому, чтобы программы эффективно работали на различных компьютерах (текущих и будущих устройств), так чтобы они обеспечивали максимальную производительность на конкретном устройстве.

Для нас данные о длительном процессе взаимодействия со структурой являются основой решения, связанной с основным процессом поддержания целостности данных и обеспечения быстрого и надежного доступа. Здесь мы не будем рассуждать. Для таких приложений реализация методов может сыграть фундаментальную роль, которые непосредственно обеспечат высокую производительность к конкретной отправной точке при разработке системы.

16.2 Индексированный последовательный доступ

Примой подход к построению индексных структур в системах хранения — ссылок на элементы и элементы, упорядоченного по ключам, с последующим использованием бинарного поиска (см. раздел 12.4) для реализации операции поиска. При наличии M элементов для реализации этого метода потребовалось бы $\log_2 M$ соединений, даже в случае очень большого файла. Такая бинарная модель неизбежно приводит к разложению двух равнозначностей этого простого метода. Во-первых, сам по себе индекс очень велик и в общем случае не помещается на одной странице. Поскольку доступ к страницам можно получить только через ссылки из структуры, вместо этого нужно построить индекс по частям, обобщающий бинарный поиск с ключами и указателями на страницы во внутреннем узле и с ключами и указателями на элементы во внешних. Во-вторых, доступ к M элементам требует затрат на доступ к 2. поэтому можно воспользоваться M -арным деревом при такой же затрате на каждый узел, что и в случае применения бинарного дерева. Такое усовершенствование уменьшает количество соединений до значения, приблизительно пропорционально $\log_2 M$. Как было показано в разделах 14–15, на практике это значение можно считать константой. Например, если M равно 1000, то $\log_2 M$ меньше 5 для M меньше 1 триллиона.

то недостатки и обеспечения возможности увеличения базы данных и будущего в рамках системы расширения: лучше строить на дисках и лучше пространств на страницах; но, в конечном итоге, подобная техника ни оправдаться не очень эффективной и динамически ситуативна (см. упражнение 16.3). Методы, которые будут рассмотрены в разделах 16.3 и 16.4, обеспечивают эффективные и эффективные интернативы таким специализированным схемам.

Лемма 16.1 Для вычисления конки в заданное количество раз для заданного количества элементов заданного количества элементов, однако основная задача может быть решена с помощью одной операции.

В данном случае (и в других разделах книги) термин «вычисление» используется несколько по-иному для обозначения действия, которое пропорционально $\log M$ для большого значения M . Как уже отмечалось, это оправдано для размеров файлов, встречающихся на практике. На рис. 16.3 показаны дополнительные примеры. Даже при наличии 128-разрядного элемента списка, предназначенного для указания примерно большого количества различных элементов, заданного 2^{128} , элемент с данными конки может быть без потерь при $\log M$ для всего 2^1 элементов, при 1000-разрядном конки величина.

Мы не будем рассуждать реализацию, которая обеспечивает поиск и построение индексов по данному, поскольку они представляют собой известные случаи более общих механизмов, рассмотренных в разделе 16.3 (см. упражнение 16.1) и программу 16.2).

Упражнения

- 16.1 Составьте таблицу значений $\log M$ для $M = 10, 100$ и 1000 и $N = 10^2, 10^4, 10^6$ и 10^8 .
- 16.2 Нарисуйте структуру индексного последовательного файла для конки 3, 6, 177, 143, 532, 372, 151, 774, 470, 411, 706, 461, 612, 701, 474, 774, 535, 343, 451, 351, 430, 664, 127, 345, 171 и 157 при $M = 3$ и $M = 6$.
- 16.3 Предположим, что мы строим структуру индексного последовательного файла для M элементов, помещаемых на страницах размером M , но оставляем на каждой странице k свободных ячеек для возможности расширения. Приведите формулу для определения количества элементов, необходимых для выполнения поиска, в виде функции от M , M и k . Используйте эту формулу для определения количества элементов, необходимых для выполнения поиска при $k = M/10$, $M = 10, 100$ и 1000 и $N = 10^2, 10^4, 10^6$ и 10^8 .

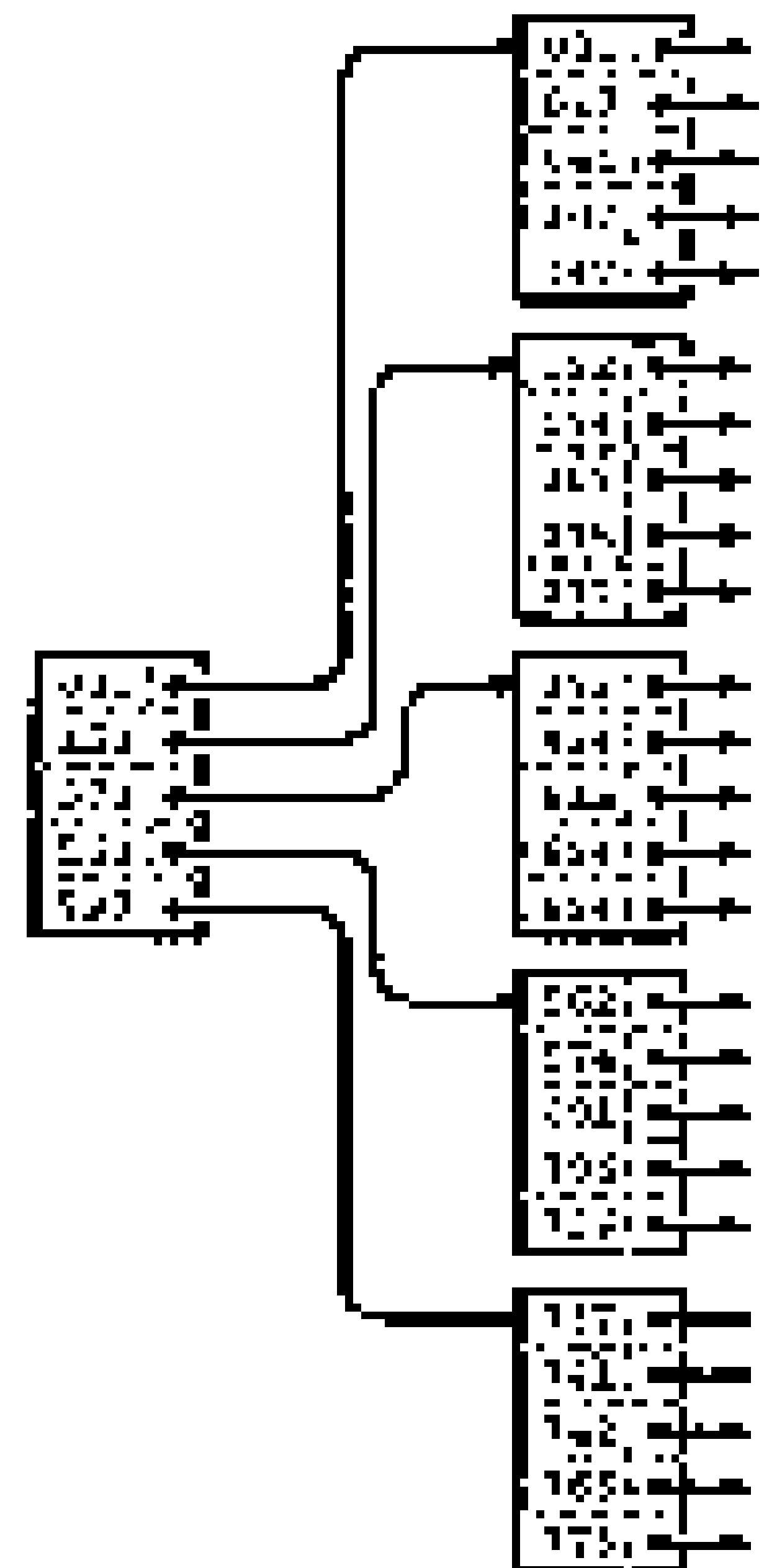


РИСУНОК 16.3 Структура индексного последовательного файла в последовательном файле. В последовательном файле каждая страница имеет фиксированный размер, и каждая страница содержит один или несколько элементов. Для каждого элемента указывается номер страницы, на которой он находится. В последовательном файле элементы хранятся в порядке их появления в файле.

а 16.4 Предположите, что аргументы по завершению составлены из одно и тех же элементарных единиц времени, а средние затраты на поиск элемента на странице составляют около $\frac{1}{2}M$ элементарных единиц времени. Найдите значение M , при котором затраты на поиск в структуре индексного последовательного файла минимальны, при $\frac{1}{2}B = 10, 100$ и 1000 и $N = 10^3, 10^4, 10^5$ и 10^6 .

16.3 B-деревья

Для построения структур поиска, которые могут быть эффективны в динамических ситуациях, мы будем строить многопозиционные деревья, но при этом отдадим от предпочтение в соответствии с культурой кредитный узел должен иметь k связей с M узлами. Внесло этого мышления узла, что каждый узел должен иметь максимум M связей, чтобы они позволяли на странице, но при этом узлы могут иметь и меньше связей. Чтобы гарантировать, что узлы имеют достаточное количество связей для обеспечения питания, необходимого ради представления узла, мы должны иметь k связей. Чтобы все узлы имели, по меньшей мере (можно) $\frac{1}{2}M$ связей, он исключенным, быть может, корень, который должен иметь не менее одной связи (двух связей). Причинами этого исключения для корня является то, что при рассмотрении алгоритма. Такие деревья были названы B-деревьями Байера (Bayer) и МакКрейга (McCreight), которые в 1969 г. первыми попытались использовать многопутевыми связанными деревьями для внешнего поиска. Термин B-дерево часто используется для описания именно той структуры данных, которая стриктно алгоритмом, предложенным Байером и МакКрейгом; мы же будем использовать его в качестве общего термина для обозначения семейства связанных алгоритмов.

Мы уже встречались с реализацией B-дерева: из упражнений 13.1 и 13.2 видно, что B-дерева четвертого порядка, в которых каждый узел имеет не более 4 и не менее 2 связей, эквивалентны тем самым, как обобщенным 2-3-4-деревьями, описанным в главе 13. Действительно, благодаря абстракции можно непосредственно обобщить и реализовать B-дерева, обобщая реализацию, включающую 2-3-4-дерева, описанные в разделе 13.4. Однако, различия между внутренним и внешним поиском, упомянутое в разделе 16.1, приводят к ряду различий в реализациях. В этом разделе мы рассмотрим реализации, которые

- обобщают 2-3-4-дерева до деревьев, имеющих от $M/2$ до M узлов
- представляют многопутевые узлы при помощи массива элементов и связей

10^3	число страниц
10^4	слова в словах "Моби Дик"
10^5	полный набор книг в библиотеке
10^{12}	телевизионных выводов в мире
10^{18}	людей, которые были живыми в Бигбен
10^{25}	гос. инж. на побережье Калифорнии
10^{28}	разрешен в Азии
10^{30}	разрешен в Азии
10^{32}	энергия в Солнечной

РИСУНОК 16.3 ПРИМЕРЫ РАЗМЕРОВ НАБОРОВ ДАННЫХ

Эти размеры являются значениями, которые являются, как правило, средними значениями, содержащими более 10^6 элементов. Даже в такой маленькой базе данных можно было бы найти все элементы менее 10 секунд при 1000 операциях в секунду. Даже если бы как что-то было сохранено информации в каждом элементе до 1000, 1000 элементов, можно было бы найти все элементы менее 10 секунд при 1000 операциях в секунду.

- реализует интерфейс, описывающий структуру дерева, содержащего элементы
- выполняет логическое разделение
- отделяет индекс от элементов

Для выполнения операций в этом дереве необходимы, однако удобны во многих ситуациях и активно используются разделения В. Перепишем.

На рис. 16.4 показано абстрактное 4-5-6-7-8-дерево, являющееся обобщением 2-3-4-дерева из раздела 15.3. Обобщение заключается в том, что 4 узлы имеют три клича и четыре связи, 5-узлы — четыре клича и пять связей и т.д., три клича соответствуют одной связи для каждого возможного интервала между кличами. Поиск начинается с корня и заканчивается переходом от одного узла к другому с определением соответствующего интервала для искомого клича в текущем узле. Другое обобщение заключается в том, что соответствующие кличи могут быть достигнуты до следующего узла. Поиск завершается по достижении клича, который встречается в любом рассматриваемом узле, и гарантируется, если найденный узел достигнут без клича. Подобно тому как это было возможно в 2-3-4-деревах, тот же клич может попытаться в нижней части дерева после того, как он был найден, хотя по дороге может быть достигнуто решение по более узлам: если клич x узла, он заменяется 2 узлом, связанным с кличем x узла y . Затем, каждый раз когда встречается x -узел, он заменяется $1 \leq t \leq 1$ -узлом, связанным с кличем x узла y . Этот подход гарантирует наличие метода для вставки нового узла в случае достижения нижней части дерева.

Наша же, как описывалось в разделе 15.3 применительно к 2-3-4-деревьям, разделение можно выполнять снизу вверх: сначала разделение через клич в наименьшем узле, а затем только по достижении узла 8-узла — в этом случае он должен иметь два 4-узла со связями, соответствующими кличу x и двух связей в его родительский узел. Всплывающее разделение выполняется до тех пор, пока не встретится узел-потомок, отличный от 8-узла.

Пусть значения в произвольном дереве относятся к n на $M/2$, а k — на M , приведенные операции проследуются по спускающим поискам и вставкам для $M/2 \dots M$ деревьев при любом произвольном первоначальном значении M , кратном 2 (см. выражение 16.9).



РИСУНОК 16.4 4-5-6-7-8-ДЕРЕВО

На рисунке показано обобщение 2-3-4-дерева, оно представляет структуру: узел 4 имеет 3 клича (а не 3 клича: 3 клича, соответствующих). Как и в дереве 2-3-4-дерева, оно обобщается в том, что 4 узла имеют три клича и четыре связи, 5-узлы — четыре клича и пять связей и т.д., три клича соответствуют одной связи для каждого возможного интервала между кличами. Другое обобщение заключается в том, что соответствующие кличи могут быть достигнуты до следующего узла. Поиск завершается по достижении клича, который встречается в любом рассматриваемом узле, и гарантируется, если найденный узел достигнут без клича. Подобно тому как это было возможно в 2-3-4-деревах, тот же клич может попытаться в нижней части дерева после того, как он был найден, хотя по дороге может быть достигнуто решение по более узлам: если клич x узла, он заменяется 2 узлом, связанным с кличем x узла y . Затем, каждый раз когда встречается x -узел, он заменяется $1 \leq t \leq 1$ -узлом, связанным с кличем x узла y . Этот подход гарантирует наличие метода для вставки нового узла в случае достижения нижней части дерева.

Вытеснение узлов так требует поэтапной добавления элементов в структуру, но по основанию структуры результирующего дерева можно определить важные события, происходящие во время его построения. Данака содержит семь ключевых событий, поэтому должно было быть выполнено шесть различных внешних узлов, и поскольку его узлы ранга 3, корень дерева должен был разветвляться дважды. Эти события описаны в сопровождающих рисунки комбинаторных.

В программе 16.1 прописаны определения типа для узлов рассматриваемой реализации B-дерева. Мы не указывали подробно структуру узлов, что означало бы сделать в реальной реализации, поскольку для этого потребовались бы ресурсы на жестких дисках машины. Для простоты мы используем один тип узла, хотя каждый узел состоит из массива записей, каждая из которых содержит элемент, ключ и связь. Каждый узел содержит также счетчик, указывающий количество активных записей. Мы не обращаемся к элементам во внутреннем узле, мы не обращаемся к связям во внешних узлах, и мы не обращаемся к ключам внутри элементов дерева. При построении конкретной структуры данных в реальном приложении можно сэкономить память за счет применения таких конструкций, как union (объединение) или наследование классов. Можно было бы также сэкономить память путем увеличения времени выполнения, используя вместо входов связи в элементах вместо ключей. Такие конструктивные решения сопряжены с определенными изменениями кода и зависят от конкретного характера ключей, элементов и связей в приложении.

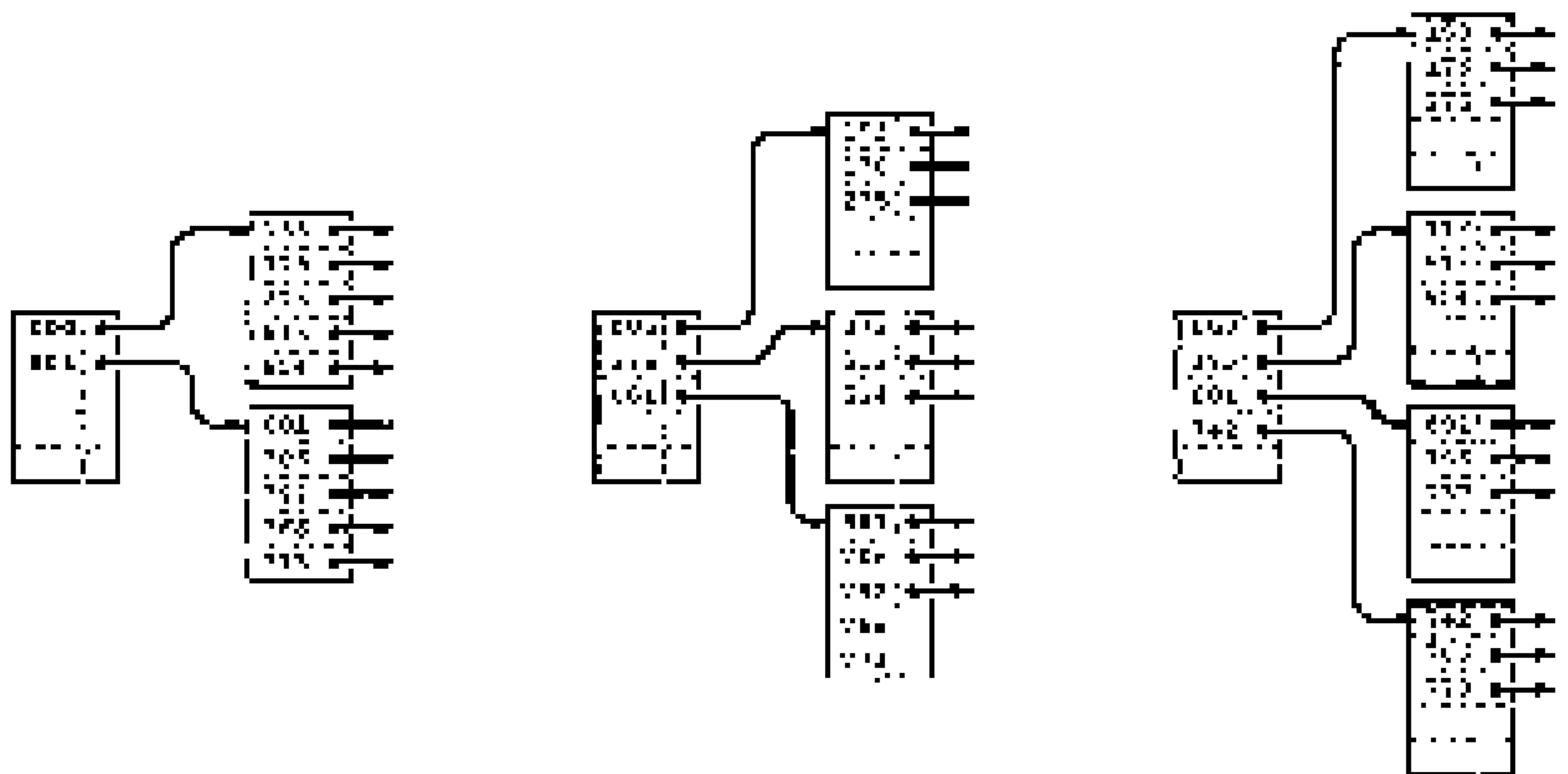


РИСУНОК 16.6 ПОСТРОЕНИЕ B-ДЕРЕВА, ЧАСТЬ 1

После создания четырех узлов 100, 200, 300 и 400 в корневом узле B-дерева, иллюстрирующем рис. 16.3, все элементы структуры представляются тремя узлами (рис. 16.6a). Затем, для элемента 200 в первом узле создается второй узел с элементами 100 и 200, а также третий узел с элементами 200 и 300 (рис. 16.6b). Затем, для элемента 300 создается четвертый узел с элементами 300 и 400 (рис. 16.6c). Таким образом, узел 200 разветвляется на два узла, а узел 300 разветвляется на два узла. Таким образом, структура B-дерева, иллюстрирующая рис. 16.3, превращается в структуру B-дерева, иллюстрирующую рис. 16.6c.

Программа 16.1 Определения типов узлов B-дерева

Каждый узел в B-дереве содержит массив и счетчик количества элементов данных в массиве. Каждая запись массива представляет либо ключ, либо элемент в связи с ключом. Во внутренних узлах используются только ключи и счетчик во внешних узлах не используются только ключи и элементы. Также узлы изначально инициализируются в виде пустого узла путем установки значения поля счетчика равным 0.

```
template <class Item, class Key>
struct entry
{ Key key; Item item; struct node *next; };
struct node
{ int n; entry<Item, Key> e[N];
  node() { n = 0; }
};
typedef node *link;
```

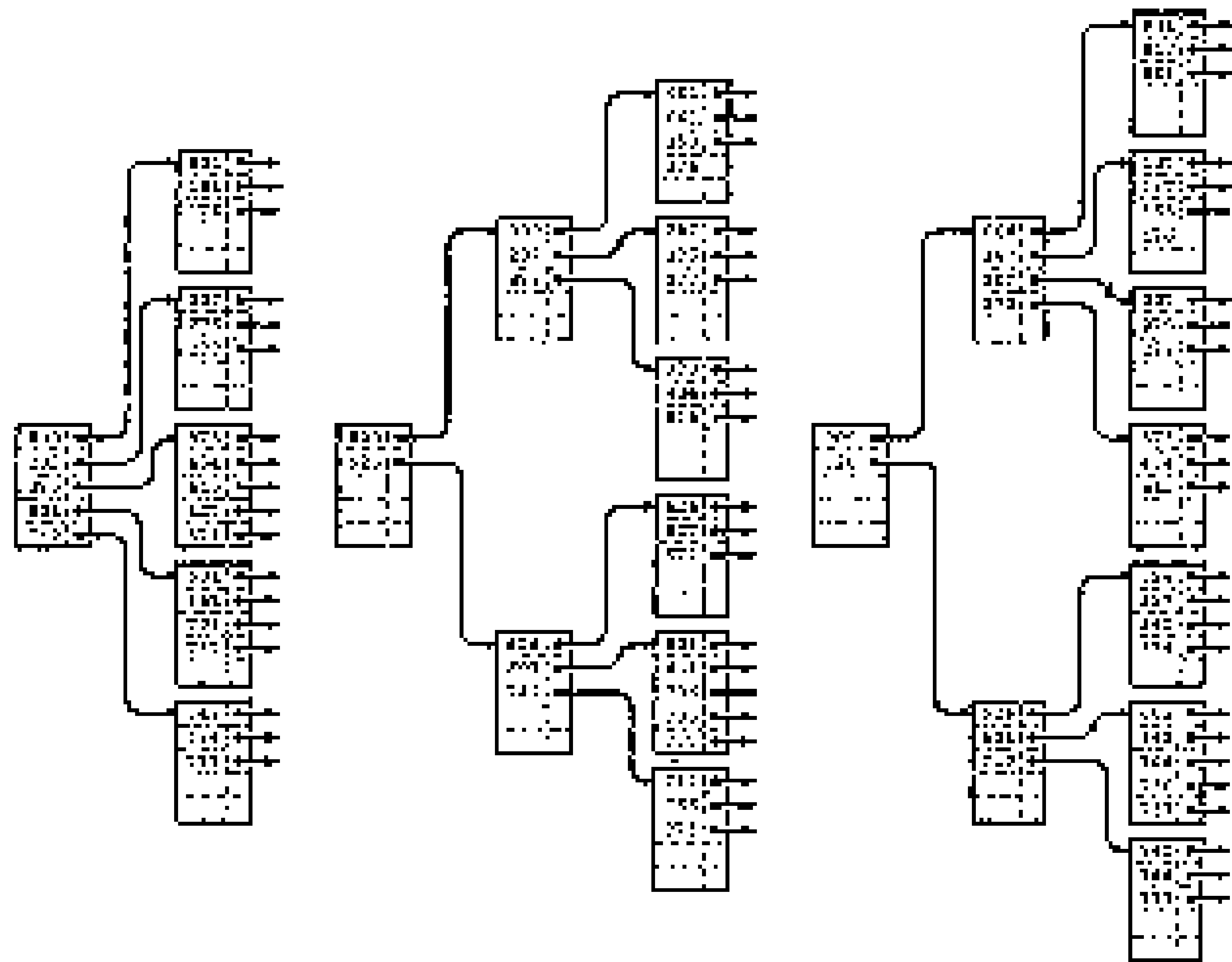


РИСУНОК 16.7 ПОСТРОЕНИЕ B-ДЕРЕВА, ЧАСТЬ 3

Продолжение предыдущего рисунка, все элементы 12-го и 13-го 536, 436, 647, 287, 001, 277, 067, 776, 526, 562, 017, 107 и 747 и элементы 14-го 166. Различные значения приведены при помощи ключей 277 (различные значения), 526 (различные значения) и 107 (различные значения). Различные значения, которые являются 526 (различные значения) и 107 (различные значения) и 166 (различные значения) являются 166 (различные значения) и 107 (различные значения).

Программа 16.3 Вставка в B-дерево

Новый элемент добавляется за счет перемещения большого элемента вперед на одну позицию, как в сортировке пузырьком. Если это приводит к перемещению элемента за пределы функции `split`, для разделения узла на два элемента в этом месте остается место в новом узле со внутренним родительским узлом, который также может быть разделен; таким образом, минимальное количество узлов разбивается в поле для себя.

public:

```
link insertR(link L, item x, int h)
{ int i, j; key v = x.key(), entry = new entry; L
  v.key = x; v.data = x;
  if (h == 0)
    for (j = 0; j < h-1; j++)
      if (v < h->b[j].key) break;
  else
    for (j = 0; j < h-1; j++)
      if ((j-1 == h-2) || (v < h->b[j+1].key))
        { link u;
          u = insertR(h->b[j+1], x, h-1);
          if (u == 0) return 0;
          v.key = u->b[0].key; v.data = u;
          break;
        }
  for (i = h-2; i > j; i--) h->b[i+1] = h->b[i];
  h->b[j+1] = v;
  if (i-h-2 < 0) return 0; else return split(h);
}
```

public:

```
split(int max)
{ int n = 0; int M = 0; head = new node;
void insert(item item)
{ link n = insertR(head, item, M);
  if (n == 0) return;
  link t = new node(); t->n = 2;
  t->b[0].key = head->b[0].key;
  t->b[1].key = n->b[0].key;
  t->b[0].next = head; t->b[1].next = n;
  head = t; M++;
}
```

Как разделение узла показано в программе 16.4. В нем для переменной `M` используется четное значение, и каждый узел дерева может содержать только $M-1$ элемент. Этот подход основан на том, что в M -ый элемент узла перед разделением этого узла в специально выделенном месте не оказывается место для вставки (см. выражения 16.3) и 16.4). Для простоты и практических выкладках, приведенных далее в разделе, мы ограничиваем сверху количество элементов каждого узла (M); действительные же различия между деревьями несущественны. В приведенной реализации не пришлось бы прибегать к этой хитрости, поскольку в таком случае элемент свободного места для вставки нового узла обеспечивался автоматически.

Программа 16.4 Разделение узлов B-дерева

Чтобы разделить узел в B-дереве, мы создаем новый узел, перемещая большую половину ключей в новый узел, а затем кладем все значения счетчиков и устанавливаем служебные ссылки в середине нового узла. В этой программе предполагается, что длина ключей M является четной, а в каждом узле имеется ссылка для элемента, который вызывает разделение. Другими словами, максимальное количество ключей в узле равно $M-1$, а когда количество ключей в узле достигнет значения M , узел разделится на два узла с $M/2$ ключей в каждом.

```
link split(link b)
{ link c = new node();
  for (int j = 0; j < M/2; j++)
    c->b[j] = b->b[M/2+j];
  b->m = M/2; c->m = M/2;
  return c;
}
```

Лемма 16.1 Для любого дерева поиска или списка в B-дереве порядка M , содержащего N элементов, существует путь от корня до любого из элементов списка, который не содержит узлов, содержащих более $M/2$ элементов.

Эта лемма является следствием наблюдения, что все узлы во внутренней части B-дерева (узлы, которые не являются ни корнем, ни внешними узлами) имеют от $M/2$ до M элементов, поскольку они образованы в результате разделения полного узла, содержащего M элементов, и результатом может быть узел из $M/2$ элементов (при разделении первого узла). В лучшем случае эти узлы образуют полное дерево порядка M , что является тем самым указанным в формуле (16.1). В худшем случае мы получаем полное дерево порядка $M/2$.

Когда M равно 1000, для N около 125 миллионов высота дерева меньше трех. Этим самым существенная затрата можно уменьшить за счет логарифмичности, храня корневой узел во внутренней памяти. Для реализации дерева на диске это также можно предпринять явно перед применением любого приложения, связанного с очень большим количеством записей, в виртуальной памяти с эмулируемой корневой частью будет узел, который, скорее всего, должен храниться в быстрой памяти, поскольку обращение к нему выполняется наиболее часто.

Важно также рассмотреть из разницы поиска, которая может гарантировать вычисление максимума двух концентрированных при выполнении операций *join* и *split* в очень больших файлах. В B-деревьях находят широкое применение, поскольку они позволяют прибегнуть к этому методу. Полная практическая гибкость достигается тем, что на пути поиска можно найти узлы, что может оказаться удобным для очень больших файлов.

Лемма 16.2 B-дерево порядка M , содержащее N элементов, имеет высоту не более $\lceil \log_{M/2} N \rceil$ стрессов.

Яо (Yao) доказал это в 1977 г., прибегнув к математическому анализу, который выдан в рамках этой книги как полезное упражнение. Это анализ основывается на анализе простой перпендикулярной модели, описывающей рост дерева. После того, как установлены деревья $M/2$ узлов, в любой данный момент времени высота дерева

пигику элементам i элементам, при $M/2 \leq i \leq M$ и $z_{d_1} + \dots + z_d = M$. Поскольку вероятность появления случайного ключа в любой интервал между узлами одинакова, вероятность появления i узла с i элементами равна $1/M$. В частности, для $i \leq M/2$ та вероятность равна вероятности того, что количество внешних страниц с i элементами уменьшится на 1, а количество внешних страниц с $(i + 1)$ элементов увеличится на 1. Для $i = M/2$ эта вероятность того, что количество внутренних страниц с $M/2$ элементами увеличится на 1, а количество внутренних страниц с $M/2 + 1$ элементов уменьшится на 1. Такой вероятностный процесс называется *Марковским процессом*. Результат, полученный Яо, основывается на анализе вероятностических свойств этого процесса.

Лемму 16.3 можно также доказать, используя простую для понимания формулировку этого процесса (см. упражнение 16.11 и рис. 16.3 и 16.9). Конечно, можно было бы также просто построить случайные B-деревья и измерить их структурные характеристики. Вероятностную аппроксимацию вычисления процесса, чем проделывали исследователи, пытались они сделать полностью реальной, кроме того, она служит хорошим инструментом изучения и эмпирических алгоритмов (см., например, упражнение 16.16).

Реализация других операций таблиц с помощью шатл-механизма соответствующим результатам для других ранее рассмотренных представлений с канализованными деревьями. Поэтому они остаются в качестве упражнения (см. упражнения 16.22–16.25). В частности, реализация операций *удалить* и *добавить*, но, как обычно, довольно сложная реализация операций *изменить значение* может оказаться сложной задачей. Проблема операции *изменить значение* операции *добавить* заключается в простом удалении элемента из внешней страницы и уменьшения значения ее указателя, но что делать, если необходимо удалить элемент из узла, который имеет только $M/2$ элементов? Естественный подход такой для канализованных сообщений: ввести элемент в соответствующий узел (возможно, с уменьшением количества узлов на единицу), но реализация усложняется, поскольку приходится связывать ключи, связанные с текущим перемещением по узлам элементам. Но встречающиеся на практике ситуации можно избежать, если использовать достаточно более простой подход, основанный на идее узлов *неподвижных*. Это не обязательно снижает производительность (см. упражнение 16.25).

Многие варианты базовых абстрактных B-деревьев приводят к их несомненно. Один из них, который позволяет исключить вред из узла указывая во внутреннем узле указатель на количество количества связей страниц, в результате чего конфигурация внешнего повышается, а дерево становится более глубоким. Как уже отмечалось, в современных системах преобладают получаемые в результате таких изменений. Впрочем, поскольку стандартные значения параметров зависят реализации, операции *добавить* и *удалить* при поиске всего двух контролируемых — эффективности, которую можно увидеть пометкой. Другой класс вариантов включает эффективность, основанную на шатл-механизме, перед реализацией связанных узлов с их родственными узлами. Упражнения 16.13–16.6 посвящены такому методу, который при работе со связанными узлами позволяет уменьшить долю используемого объема и уменьшить пространство с 41 до 23 процентов. Как обычно, выбор того или иного варианта зависит от конкретной ситуации. Помимо широкого множества различных ситуаций, в которых применяются B-деревья, не все будет полезно рассмотреть эти структуры.

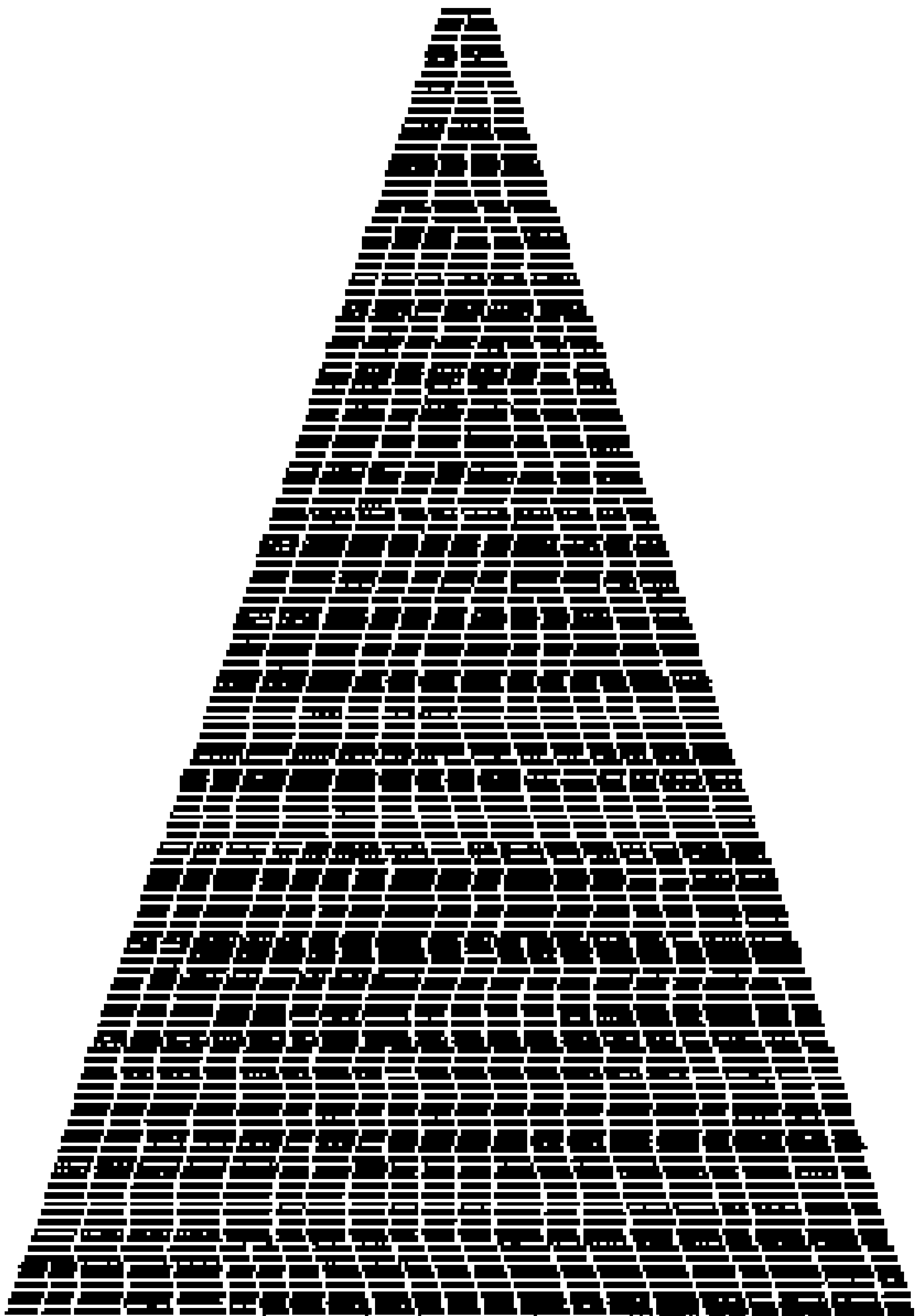


РИСУНОК 16.8 РОСТ БОЛЬШОГО В-ДЕРЕВА

Почти все, что мы знаем о структуре и функциях клеток и тканей, а также о том, как они взаимодействуют, чтобы образовать организм, было получено из исследований на животных. Однако, несмотря на то, что мы знаем о том, как клетки и ткани взаимодействуют, мы не знаем, как они взаимодействуют, чтобы образовать организм. Это связано с тем, что мы не можем наблюдать за процессом развития организма в реальном времени. Вместо этого мы должны использовать различные методы, чтобы понять, как организм развивается. Одним из таких методов является использование эмбриональных стволовых клеток. Эти клетки могут быть использованы для создания различных типов клеток и тканей, что позволяет нам изучать, как они взаимодействуют, чтобы образовать организм. Другим методом является использование компьютерного моделирования. Это позволяет нам создавать виртуальные модели организма, которые можно использовать для изучения его развития. Несмотря на то, что мы знаем о том, как клетки и ткани взаимодействуют, мы не знаем, как они взаимодействуют, чтобы образовать организм. Это связано с тем, что мы не можем наблюдать за процессом развития организма в реальном времени. Вместо этого мы должны использовать различные методы, чтобы понять, как организм развивается.

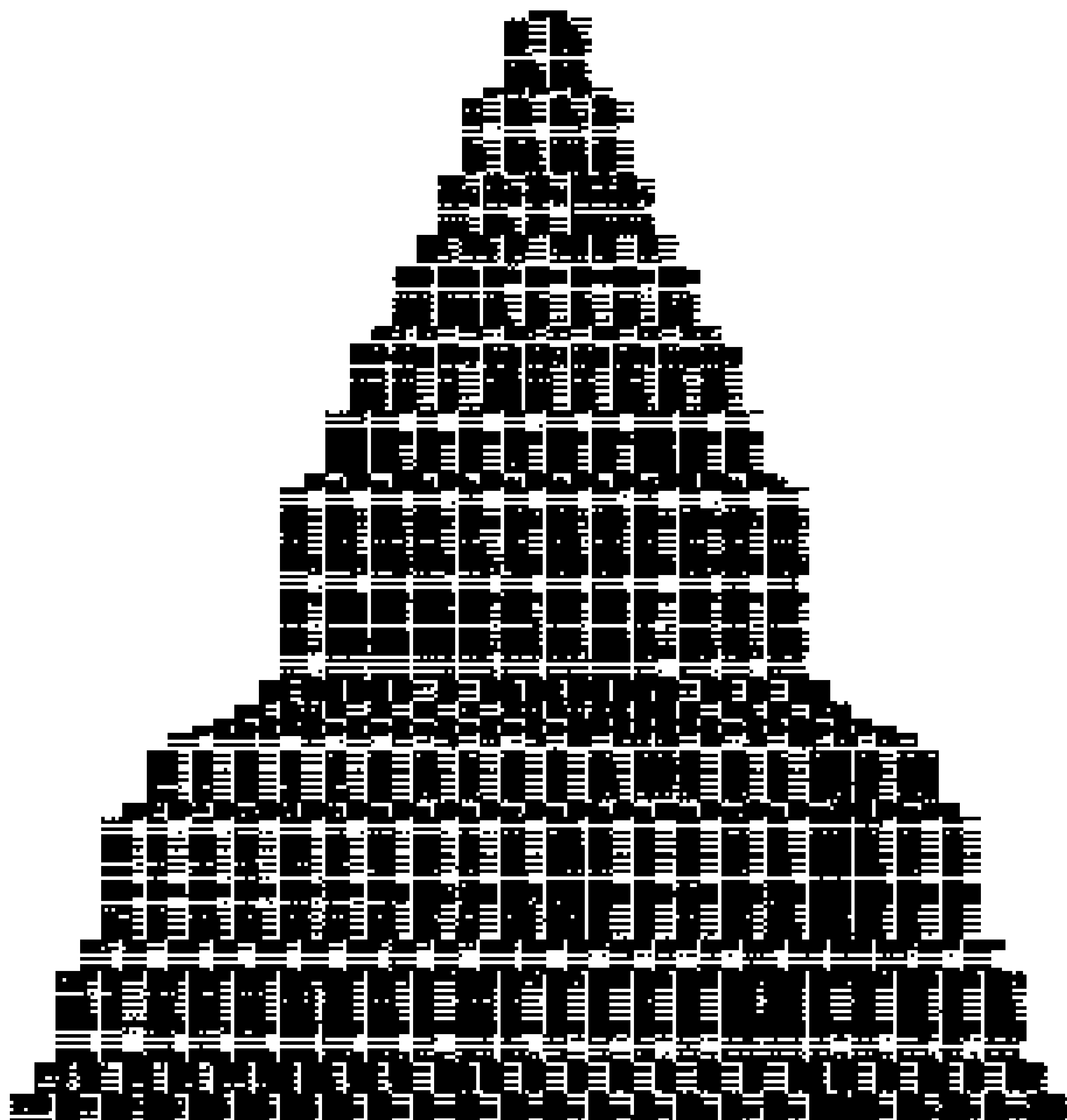


РИСУНОК 16.9 РОСТ БОЛЬШОГО В-ДЕРЕВА С ОТБРАЖЕНИЕМ ЗАНЯТОСТИ СТРАНИЦ

В этом рисунке рис. 16.8 показана, как структура развивается со временем процессом роста в дереве. Как и в предыдущем случае, большинство страниц приходится на листву, которая не растет, а лишь увеличивается по количеству. Листва растет по мере добавления новых страниц, поэтому разделение на две подструктуры становится

Мы не сможем также рассмотреть подробности реализации, поскольку это потребовало бы учета слишком многих значений от устройства к устройству. Как обычно, детали реализации также приложений — дискреционные дела, и в современных системах жёстательно ограничить различия между прикладными и системными кодами невозможно, когда базовые операции работы должны успешно.

Упражнения

- 16.5 Присвойте значения 1-4-7-6-дереву, образованного в результате вставки ключей E A S Y Q U E S T I O N W I T H I N T H E N T Y O F K E Y S в указанном порядке в первоначально пустое дерево.
- 16.6 Постройте рисунок, соответствующий рис. 16.5–16.7, иллюстрирующий процесс вставки ключей 515, 177, 141, 632, 332, 161, 774, 470, 411, 766, 151, 612, 761, 674, 774, 635, 141, 161, 351, 430, 664, 127, 345, 171 и 357 в указанном порядке в первоначально пустое дерево при $M = 3$.
- 16.7 Укажите высоту H -деревца, образованного в результате вставки и упорядочивающей подсортировки значений из выражения 16.6 в первоначально пустое дерево для каждого значения $M \geq 2$.
- 16.8 Нарисуйте 3-дерево, образованное в результате вставки 16 одинаковых значений в первоначально пустое дерево при $M = 4$.
- 16.9 Нарисуйте 1-2-дерево, образованное в результате вставки ключей E A S Y Q U E S T I O N в первоначально пустое дерево. Объясните, почему 1-2-дерево не представляет практического интереса как обобщение бинарного дерева.
- 16.10 Напишите рекурсивную версию B -дерева, применяющую к программе 16.4, чтобы в ней выполнялось разделение при перемещении узлов по дереву, подобно реализации вставки в 2-3-4-дерево (программа 13.6).
- 16.11 Напишите программу для вычисления среднего количества узлов на уровне i в B -дереве порядка M , построенного путем N случайных вставок в первоначально пустое дерево при использовании вероятностного процесса, описанного в этом тексте (см. 16.1). Выставьте программу для $M = 10, 100$ и 1000 и $N = 10^2, 10^4, 10^6$ и 10^8 .
- 16.12 Предположите, что в трехуровневом дереве в узлах могут храниться во внутренней памяти, от 5 до 26 связей — в страницах, представляющих внутренние узлы, и от 1 до 26 связей — в страницах, представляющих внешние узлы. Определите в виде функции от a, b и c максимальное количество элементов, которое может храниться в таком дереве.
- 16.13 Рассмотрите структурный вариант реализации разделения B -деревца (или B^+ -дерева): когда требуется разделить узел, поскольку он содержит b элементов, мы объединяем узел с его родственником узлом. Если родственник узла содержит k элементов, при $k \leq M - 1$, мы перераспределяем элементы, помещая их в родственника, и в новый узел приблизительно по $(b + k)/2$ элементов. В противном случае мы создаем новый узел и помещаем в каждый узел дерева приблизительно по $2M/3$ элементов. Кроме того, мы позволим корню разделиться, чтобы в нем могло содержаться около $4M/3$, разделяя его и создавая новый корневой узел с двумя детьми, когда его размер достигнет этого предельного значения. Укажите порядок работы алгоритма объединения элементов, необходимого для выполнения поиска

или листов в N^* -дереве порядка M , содержащем M элементов. Сравните эти производные значения с соответствующими производными значениями для B -деревьев (см. задачу 16.7) при $M = 10, 100$ и 1000 и $M = 10^3, 10^4, 10^5$ и 10^6 .

- 16.14 Разработайте рекурсивную функцию `visit` в B^* -дереве (основанную на рекурсивном методе удовлетворения разделения).

- 16.15 Создайте рисунок, аналогичный рис. 16.8 для иллюстрации вершинского метода разделения разделения.

- 16.16 Выполните вершинистую реализацию (см. упражнение 16.11) с целью определения среднего количества страниц, задействованных при выполнении алгоритма метода разделения разделения при помощи построения N^* -дерева ширины M в результате вставки случайных узлов в предварительно пустое дерево при $M = 10, 100$ и 1000 и $M = 10^3, 10^4, 10^5$ и 10^6 .

- 16.17 Создайте программу для выполнения построения внешнего B -дерева, состоящая с помощью указателей страниц, содержащих от M до $2M$ упорядоченных элементов.

- 16.18 Можно ли сконструировать индекс упорядоченных страниц за счет использования алгоритма Б-дерева, рассмотренного в тексте раздела (программа 16.3)? Обоснуйте ответ.

16.19 Представьте, что много различных компьютеров имеют доступ к одному и тому же индексу, что позволяет нескольким программам практически одновременно предпринимать попытки вставки нового узла в один и тот же B -дерево. Поскольку, конечно, в такой ситуации может быть предложено несколько различных решений B -дерева, а не одно единственное. Предположите, что каждая программа может (в момент этой вставки) изменить другие программы любого узла, который она вставляет в, возможно, изменит его структуру.

- 16.20 Изложите реализацию B -дерева в программах 16.1—16.3, чтобы в одном узле дерева можно содержалось M элементов.

- 16.21 Постройте таблицу значений логарифмов $\log_2 M$ и $\log_{2M} M$ для $M = 10^3, 10^4, 10^5$ и 10^6 .

- 16.22 Разработайте функцию `visit` для таблиц символов, основанной на использовании B -дерева.

- 16.23 Разработайте операцию `join` для таблиц символов, основанной на использовании B -дерева.

- 16.24 Разработайте функцию `delete` для таблиц символов, основанной на использовании B -дерева.

- 16.25 Разработайте операцию `delete` для таблиц символов, основанной на использовании B -дерева, при использовании простого метода, когда указанный элемент удаляется из внешней страницы (возможно, допустив, что его количество элементов на странице становится меньше $M/2$), но чтобы изменения не распространялись вверх по дереву, за исключением, возможно, некоторой значимой кельи, если удаленный элемент является наименьшим по длине страницей.

- 16.26 Измените программы 16.2 и 16.3, чтобы текущий узел использовался для быстрого поиска (программа 12.6). Определите значение M , при котором возникла бы необходимость в программе на построение таблицы символов за счет вставки M элемен-

гов со случайными ключами в перешитых пустых таблицах, минимально, при $n = 10^2, 10^4, 10^6$ и 10^8 , и сравните полученные средние времена с соответствующими значениями для B-деревьев (таблица 13.6).

16.4 Расширяемое хеширование

Алгоритмы B-деревьев, расширяющих произвольные авторасширяемые различия поиска на внешний поиск, были разработаны Олдричем [Oaldrich], Миттермайером (Mittermeier), Пфайенером (Pfaffenberger) и Спрингом (Springer) в 1978 г. Их метод, названный расширяемым хешированием, применим к реализации операции поиска для которой в типичных приложениях требуется всего одно-два до-определения. Для соответствующей реализации операции *insert* также (но не всегда) требуется всего одно до-определение.

Расширяемое хеширование объединяет свойства методов сепарации, не использующих многопользовательских B-деревьев и многопользовательского доступа. Первый из методов хеширования, описанный в главе 14, расширяемое хеширование представляет собой рандомизированный алгоритм — процесс всего необходимого сводится к хешированию, которая преобразует ключи в некие числа (см. раздел 14.1). Для простоты в этом разделе мы будем просто считать, что ключи являются случайными значениями фиксированной длины. Подобно алгоритмам с использованием многопользовательских B-деревьев, описанным в главе 13, расширяемое хеширование использует только с использованием являющихся разрядов ключей в качестве плавающих указателей в таблице, размер которой является степенью 2. Подобно алгоритмам с применением B-деревьев, при использовании расширяемого хеширования у каждого элемента есть странички, которые разбиваются на две части при заполнении. Подобно алгоритмам с использованием многопользовательского доступа расширяемое хеширование поддерживает структуру, указывающую, где можно найти страницу, содержащую объект, соответствующий данному ключу. Поскольку все объясняется эти алгоритмы связаны в основном алгоритмы, расширяемое хеширование как метод более подходит для табличных структур с алгоритмами поиска.

Предположим, что количество доступных страниц диска является степенью 2 скажем, 2^k . Тогда можно поддерживать таблицу 2^k элементов различных страниц, сопоставляя k разрядов ключей для индексации указателей на страницу, соответствующую той или иной странице. Все ключи, первые k разрядов которых совпадают, как показано на рис. 16.10. Подобно тому, как это делалось в случае B-деревьев, элементы на страницах хранятся по порядку, и доступная страница, которая соответствует элементу с данным ключом, мы выполняем показывающий поиск.

Если указать размер таблицы и установить какой указатель, можно получить структуру, которую можно декомпозировать на k разрядов ключа (см. рисунок 16.11). Например, таблица страниц по прежнему определяется как содержащая элементы с ключами, значением первых k разрядов которых — 111, и она будет делиться на три части, с тем, первые k разрядов искомого ключа — 1110 или 1111. Этот последний вариант может означать увеличение таблицы.

Рисунок 16.10 и 16.11 описывают две базовые конструкции, лежащие в основе расширяемого хеширования. До сих пор, мы не объяснили, как поддерживать 2^k страниц.

То есть, может существовать несколько записей каталога, отличающихся типом узла и уже страницей, что не означает быстрой выполнения поиска в структуре путей. Абстрагировав из данных структуры ключей с различными значениями, получим M разнородных записей каталога, а по мере времени появления добавлять страницы, содержащую заданный узел. За счет использования паттернов разнородных узлов для индексации путей к каталогу. Поэтому, для ускорения поиска в таблице можно задать размер каталога.

В частности, структура данных, используемая для расширения хеширования, аналогична ранее использованной для B-деревьев. Она состоит из страниц, которые содержат до M элементов, и каталог, содержащий 2^k указателей на страницы (программа 16.5). Указатель в ячейке каталога i ссылается на страницу, которая содержит все элементы, вступившие в разряды которых равны i . Значение i выбирается логически по значениям, чтобы тип каждой структуры гарантированно хранился менее M значений. Реализация структуры `search` проста: мы используем значение i разрядов ключа для индексации внутри каталога, что обеспечивает доступ к странице, которая содержит нужные элементы с соответствующими ключами, а затем проводим последовательный поиск такого элемента на данной странице (см. программу 16.6).

Программа 16.5 Структуры данных расширяемого хеширования

Расширяемая каталог-таблица — это каталог со 2^k страницами (подобно обычным узлам в B-деревьях), который содержит до 2^M элементов. Каждая страница содержит также 2^M элементов (по количеству элементов на странице и произвольным значениям k), указывающее количество записей разрядов, которые должны воздвигать в ключах узла. Как обычно, M — количество элементов в таблице. Параметр d определяет количество разрядов, которые используются для индексации в каталоге, а Q — количество записей в каталоге; таким образом, $Q = 2^d$. В каждой таблице установленный элемент имеет размер 1 , который указывает на доступ к странице.

```
template <class Item, class Key>
class B1
{
private:
    struct node
    {
        int m; Item b[M]; int k;
        node() { m = 0; k = 0; }
    };
    typedef node *link;
    link* list;
    Item nullItem;
    int N, d, Q;
public:
    B1(int maxQ)
    {
        Q = 0; d = 0; N = 1;
        list = new link[Q];
        list[0] = new node;
    }
};
```


Примеры 3.6 Поиск в таблице расширенного хэширования

Поиск в таблице расширенного хэширования заключается в следующем: для заданного значения x находим ключ для заданного x в таблице T и пытаемся найти x в таблице T на указанной строке. Если не удалось найти x в таблице T , то x не является элементом множества. Если удалось найти x в таблице T , то x является элементом множества. Если не удалось найти x в таблице T , то x не является элементом множества. Если удалось найти x в таблице T , то x является элементом множества.

```
public
void searchLink(h, key)
{
    for (int j = 0; j < h-1; j++)
        if (T[j][key] != null) return true;
    return false;
}

public
bool search(int x)
{
    return search(h, x);
}
```

Для реализации операции `search` структура данных должна быть достаточно большой, но для этого стоит учитывать тот факт, что этот алгоритм поиска работает только без каких-либо модификаций. Чтобы обеспечить эффективную операцию `search`, необходимо ответить на следующие вопросы:

- Что делать, когда количество элементов в строке превышает ее емкость?
- Какой размер строки для каждого элемента?

Например, в примере, представленном на рис. 16.10, можно было бы использовать значение $h = 2$, поскольку некоторые строки состояли бы из нескольких элементов, а также было бы использовано значение $h = 5$, но тогда слишком много строк было бы пустыми. Как следствие, наиболее эффективное решение — поддержать операцию `search` в таблице `h` с помощью, например, указателей на строки, которые не являются частью таблицы. Например, в примере, представленном на рис. 16.10, можно было бы использовать значение $h = 5$, но тогда слишком много строк было бы пустыми. Как следствие, наиболее эффективное решение — поддержать операцию `search` в таблице `h` с помощью, например, указателей на строки, которые не являются частью таблицы.

- Что делать, когда необходимо поддерживать множество элементов в строке?

Например, в примере, представленном на рис. 16.10, можно было бы использовать значение $h = 2$, поскольку некоторые строки состояли бы из нескольких элементов, а также было бы использовано значение $h = 5$, но тогда слишком много строк было бы пустыми. Как следствие, наиболее эффективное решение — поддержать операцию `search` в таблице `h` с помощью, например, указателей на строки, которые не являются частью таблицы.

Определение 16.1. *Множество M называется h -хэшируемым, если существует функция $h: M \rightarrow \{0, 1, \dots, h-1\}$, такая, что для любого элемента $x \in M$ выполняется $h(x) = i$ тогда и только тогда, когда x принадлежит к строке i в таблице T .*

Некоторые алгоритмы построения хэшируемых множеств не позволяют им быть h -хэшируемыми. Соответствующим образом можно определить h -хэшируемость, хотя существует естественный способ определить h -хэшируемость, когда h является функцией.

Для построения этих характеристик в процессе разработки таблицы мы выполним две базовые операции: разделение исходных, при которых текстовые ключи с одной страницы переносятся на другую страницу, и деление каталога, при котором размер каталога удваивается, а значение d увеличивается на 1. В частности, при выполнении операции мы разделим d на две половины: левую половину разделим пополам, где различаются ключи, для переделения элементов, которые должны быть перемещены на новую страницу. При разделении страницы мы создадим новую таблицу измененных указателей каталога, при которой высота удвоится размер каталога.

Как обычно, лучший способ понять алгоритм заключается в рассмотрении его работы при выполнении набора ключей с перпендикулярной пустой таблицей. Задача сложная и интуитивная, которая должна объясняться алгоритмом, прежде чем в простейшей форме и принципах, лежащих в основе алгоритма, становится понятным. Построение расширяемой хеш-таблицы для набора из 25 послерекурсивных ключей, рассмотренное в этой главе, показано на рис. 16.11—16.13. Подобно тому, как это можно видеть в 16.12, деление каталога не приводит к перемещению: они просто добавляются к тем же страницам. Поскольку процесс начинается с одной страницей, а 25-рекурсивная высота, можно предположить, что семь вставок вызовут расширение страниц. Поскольку в начале размер каталога равен 1, а в конце — 16, можно предположить, что четыре вставки приведут к удвоению каталога.

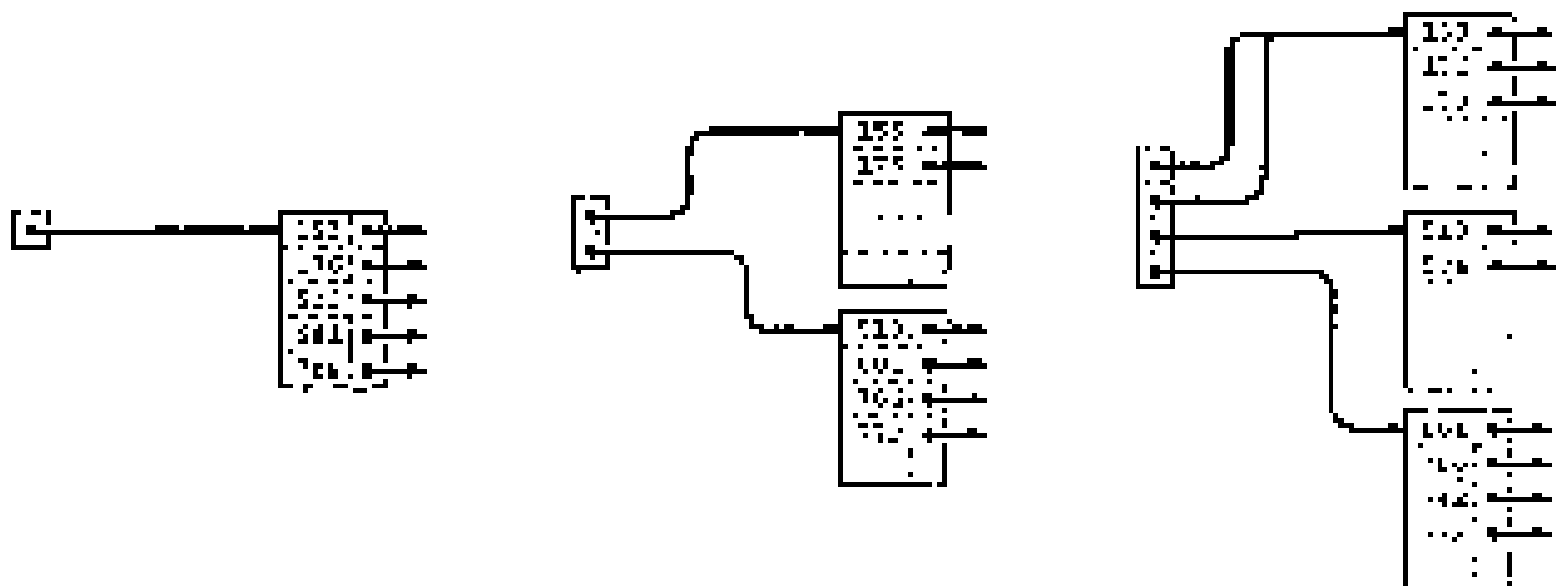


РИСУНОК 16.16 ПОСТРОЕНИЕ РАСШИРЯЕМОЙ ХЕШ-ТАБЛИЦЫ. ЧАСТЬ 1

Как и в 16.12, деление, вызванное делением d на 2, приводит к удвоению размера исходной таблицы (рис. 16.11). Затем, при делении ключа 223, мы выполняем деление на две страницы (одна содержит все ключи, начинающиеся с 0 разряда, другая — все ключи, начинающиеся с 1 разряда) и удваиваем размер каталога, чтобы он содержал по одной указатель на каждый из страниц (рис. 16.12). Ключи 242 и 243 являются страницей (каждая из них начинается с 1 разряда), а ключи 153 — с 0 разряда. Следовательно, они остаются с 0 разрядом, но каждая из них должна быть разделена, чтобы можно было поместить ключи 53 и 1. Для выполнения этого деления все элементы, ключи которых начинаются с разряда 0, перемещаются на одну страницу, а все элементы, ключи которых начинаются с разряда 1 — на другую. После того как разделены все элементы, чтобы о нем можно было поместить указатель на обе две страницы (рис. 16.13). Каждая содержит две указателя на страницы, содержащие элементы с ключами, начинающимися с 0 разряда (для ключей, которые начинаются с разряда 0) и указателя на другую страницу, которую начинают с разряда 1.

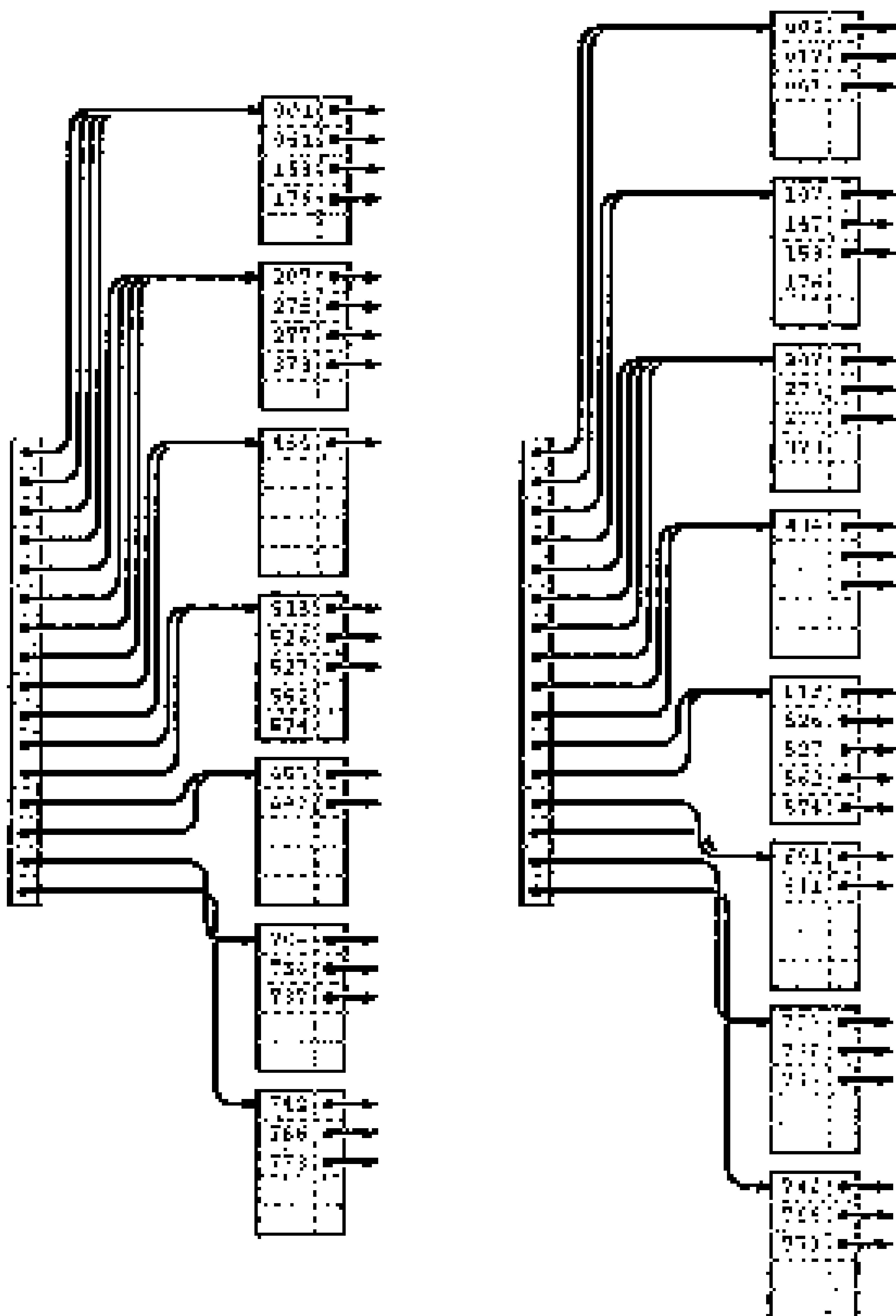


РИСУНОК 16.13 ПОСТРОЕНИЕ РАСШИРЯЕМОЙ ЖЕШ-ТАБЛИЦЫ. ЧАСТЬ 3

Продолжение диаграмм, продолженный по рис. 16.11 и 16.12, жем состоит из модулей 526, 502, 017, 107 и 147 в крайнем случае 8-дверей, содержающиеся по рис. 16.6. Продолжение уже представлено при описании модуля 526 (указаны только в 107 (продолжение строки))

Программа 16.7 реализована на языке Pascal для расширенной леш-таблицы. Прежде всего, мы обратимся к строкам, каждая из которых содержит леш-элемент, преобразованный в десятичную запись, как это делалось при поиске. Затем мы рассмотрим, как это делается для значений x и y в вершинах (см. программу 16.5). Если в результате этой операции будет вычислено, что вычислен нулевой индекс, как это делалось для Верверовых, то мы найдем, что эти значения совпадают. Каждая строка содержит 6 нулевых разрядов, которые заданы по порядку в строках h и h (см. строки 14, 15 программы). Нулевые разряды h и h являются нулевыми, начиная с 0. В строке 16 также вычислен разряд, который используется для определения способа размещения элементов

Программа 16.7 Вставка в расширенную леш-таблицу

Программа реализована на языке Pascal для расширенной леш-таблицы. Прежде всего, мы обратимся к строкам, каждая из которых содержит леш-элемент, преобразованный в десятичную запись, как это делалось при поиске. Затем мы рассмотрим, как это делается для значений x и y в вершинах (см. программу 16.5). Если в результате этой операции будет вычислено, что вычислен нулевой индекс, как это делалось для Верверовых, то мы найдем, что эти значения совпадают. Каждая строка содержит 6 нулевых разрядов, которые заданы по порядку в строках h и h (см. строки 14, 15 программы). Нулевые разряды h и h являются нулевыми, начиная с 0. В строке 16 также вычислен разряд, который используется для определения способа размещения элементов

разряды:

```
void insert(link h)
{
    link L = new node;
    while (L->n == 0 || L->n == N)
    {
        L->n = L->n + 1;
        for (int i = 0; i < N; i++)
            if (h->n[i] < L->n[i])
                L->n[i] = h->n[i];
        L->n = L->n + 1;
    }
    L->n = L->n + 1;
}
```

```
void insert(link h, int x)
{
    for (int i = 0; i < N; i++)
        if (h->n[i] < x)
            break;
    for (int j = (h->n[i] + 1); j < N; j++)
        h->n[j] = (h->n[j] + 1);
    h->n[i] = x;
    if (h->n[i] == N)
        L->n[i] = N;
}
```

разряды:

```
void insert(link h, int x)
{
    insert(h->n, x, 0, N, N);
}
```


4. достаточно, для образования строки из n символов строку строки, затем по-
менем по значению, для которых данный размер равен 0, на строку строки, а
все значения, для которых n равно 1 — на строку строки, и, наконец, устано-
вившим значением строки размер равен 1 для обеих строк. Возможен слу-
чай, когда все строки имеют одинаковое значение размера 1, что приводит к не-
платному значению строки. В этом случае мы просто перейдем к следующему размеру,
продолжая процесс до тех пор, пока каждая строка не будет содержать, по мень-
шей мере, один символ. Со временем процесс должен завершиться, если только не
будет 0. В этом случае мы просто закончим. Вскоре мы рассмотрим этот случай.

Как и в случае В-дерева, на каждой строке можно использовать для того, чтобы
можно было вычислить разделение строки строки. Это упрощает код. Следует также
отметить, что эта техника имеет некоторые практические преимущества, и ее можно
использовать при анализе.

При создании новой строки в каталоге необходимо указать указатель на нее.
Выполняемый это в том же коде, что и в предыдущем примере. В этом примере является
случай, когда перед каждой строкой содержится ровно два указателя на строку, ко-
торая разделяется. В этом случае нужно всего лишь убедиться, чтобы строка указ-
ывала на новую строку. Если количество размеров k , которое требуется для
различения каждой из новых строк, превышает количество размеров d , указыва-
ющих на строку, размер k должен быть увеличен, чтобы в нем могла по-
меститься новая строка. И, в завершение, указатели каталога должны быть соответ-
ствующим образом.

Программа 15.8 Вставка в каталог расширенного деширования

Это относительно простой код, который можно использовать для деширования.
У нас имеется список t с указателями, содержащими элементы, первые k элементов которых
содержат, по крайней мере, один элемент. В этом примере мы будем использовать
значение d и n равно, для того чтобы указать, что $d[k]$, где k — значение
размера d и n равно 0 (и n равно 0, и n равно 0, и n равно 0). Если k равно d , раз-
мер каталога следует увеличить, так как не будет достигнуто равенство значений d и k .
Если k равно d , необходимо увеличить длину строки и увеличить — с парами k и d , раз-
мер каталога следует увеличить, чтобы не было достигнуто равенство значений d и k .
for значится количеством указателей, которые необходимо увеличить (2^d), а все
указатели должны быть соответствующим образом.

```
void insertID(link t, int k)
{ int i, n, x = bits(t->ID, ID, k);
  while (d < k)
  { link *old = dir;
    d += 1; d += d;
    dir = new link[d];
    for (i = 0; i < d; i++) dir[i] = old[i/2];
    if (d < k) dir[bits(x, 0, d)-1] = new node;
  }
  for (i = 1; i < d; i++) i += 2;
  for (i = 0; i < d; i++) dir[x/i-1] = t;
}
```

Если более 4 элементов имеют дублирующиеся ключи, таблица переопределяется и
программа 15.7 должна быть выполнена, чтобы найти способ разделения ключей.
С этим же есть связанная проблема, заключающаяся в том, что каждый может

пиксельных преобразований большим, если клик имеет большее количество совпадающих координат рядов. Для файлов, содержащих большое количество дублированных кнопок или длинные сеточки различных полостей, в которых она содержится, эти сеточки имеют значительные временные затраты на выполнение сортировки сначала по старшей строке. Для преодоления этих проблем приходится прибегать не к случайному, а к систематическому хеш-файшингу (см. упражнение 16.33). При наличии большого количества дублированных кнопок, даже при использовании хеш-функции, приходится предпринимать экстраординарные действия, поскольку хеш-функции отображают равные кнопки на равные хеш-значения. Дублированные кнопки могут сделать каталог неестественно большим; кроме того, алгоритм должен быть разорван при наличии большого количества равных кнопок, чем компенсируется на одной странице. Следовательно, прежде чем исполнить этот код, необходимо добавить проверки, предотвращающие потенциально возможные условия (см. упражнение 16.35).

С точки зрения прозрачности алгоритма интерес представляют количество кнопок, содержащихся в строках (как и в случае В-деревьев) и размер каталога. Для этого алгоритма разработаны объектно-ориентированные функции, которые характеризуют прозрачность, определенные для среднего случая, сохраняются для любой прозрачности N различных строк.

Лемма 16.5 При прозрачности страниц, которая может быть достигнута, для достижения рассмотренного темпирования для файлов, содержащего M записей, в среднем требуется около $1.44(N/3)$ страниц. Скорость упаковки кнопок в каталог увеличивается в среднем в $3.92(N^{1/3})$ раз.

Этот (достаточно абстрактный) результат дополняет анализ теоретический, кратко рассмотренный в предыдущей главе (см. соответствующие). Числовые значения констант для количества страниц и размера каталога соответственно равны $1.44 = 1/n^2$ и $3.92 = 6/n^2$, поэтому числовые значения величин колеблются вокруг указанных средних значений. Это не должно пугать читателя, поскольку, например, размер каталога должен быть равен величине $2 - \epsilon$, который в результате должен приниматься по умолчанию.

Обратите внимание, что размер каталога возрастает быстрее, нежели линейно по отношению к N , особенно при малых значениях M . Однако, для значений $N \ll M$, предельное число практической сложности, значение $N^{1/3}$ достаточно близко к 1, поэтому реально можно ожидать, что каталог будет иметь около $1.44(N/3)$ страниц.

Мы принимаем, что каталог — это массив указателей. Это можно хранить в памяти, если он слишком велик, в памяти можно хранить массивный указатель, который указывает на место нахождения страниц каталога, используя такую же схему индексирования. Итог же можно добиться, если один указатель, индексный первый указатель, хранить на первом 10 разрядном, а второй — на остальных разрядах (см. упражнение 16.36).

Подобно тому, как это делается для В-деревьев, реализация остальных операций в большинстве случаев оказывается в качестве упражнения (см. упражнения 16.33 и 16.41). Так же как и в случае В-деревьев, транзитная реализация операций должна представлять собой сложную задачу, но разрешение задачи не является сложной задачей — это можно считать аналогичным, который может оказаться эффективным во многих ситуациях на практике.

Упражнения

- 16.27 Изобразите страну, если бы она была рис. 16.10, использовался каталог, размер которого равен 32?
- 16.28 Постройте рисунки, соответствующие рис. 16.11—16.13, иллюстрирующие процесс вставки ключей 562, 121, 240, 771, 274, 233, 401, 273 и 201 в указанный порядок в первоначально пустое дерево, при $M = 5$.
- 16.29 Постройте рисунки, соответствующие рис. 16.11—16.13, иллюстрирующие процесс вставки ключей 562, 121, 240, 771, 274, 233, 401, 273 и 201 в обратном порядке в первоначально пустое дерево, при $M = 5$.
- 16.30 Предположите, что имеется массив упорядоченных элементов. Опишите, как можно было бы определить размер каталога расширяемой хеш-таблицы, соответствующей этому набору элементов.
- 16.31 Создайте программу, которая строит расширяемую хеш-таблицу из массива упорядоченных элементов, выполняя две процедуры: одна для определения размера каталога (см. упражнение 16.29) и вторая для распределения элементов по ячейкам и заполнения каталогов.
- 16.32 Приведите набор ключей, для которого расширяющаяся расширяемая хеш-таблица имеет каталог размером 16 при размещении таблицы ухлестов на одной странице.
- 16.33 Создайте код расширяемого хеширования, подобный рис. 16.4.
- 16.34 Создайте программу для вычисления среднего количества проходов страниц и среднего размера каталога для расширяемой хеш-таблицы, построенной в результате M случайных вставок в первоначально пустое дерево, при емкости страны M . Вычислите долю проходов для значений $M = 10, 100$ и 1000 и $M = 10^2, 10^3, 10^4$ и 10^5 .
- 16.35 Дополните программу 16.3 соответствующими проверкам с целью предотвращения неправильной работы в случае вставки в таблицу слишком большого количества дублированных ключей или ключей, у которых совпадают старшие M битов двоичного представления.
- 16.36 Измените реализацию расширяемого хеширования, приведенную в программах 16.3—16.7, чтобы в ней использовался двухуровневый каталог, в каждом эле которого содержится не более M ухлестов. Обратите особое внимание на действия, предпринимаемые, когда каталог должен расширяться с одноуровневого до двухуровневого.
- 16.37 Измените реализацию расширяемого хеширования, приведенную в программах 16.3—16.7, чтобы в структуре таблицы на одной странице могло существовать M элементов.
- 16.38 Реализуйте операцию `del` для расширяемой хеш-таблицы.
- 16.39 Реализуйте операцию `del` для расширяемой хеш-таблицы.
- 16.40 Реализуйте операцию `del` для расширяемой хеш-таблицы.
- 16.41 Реализуйте операцию `del` для расширяемой хеш-таблицы, используя метод, описанный в упражнении 16.25.

- ♦♦ 16.42 Разработайте перенос расширяемого хэширования, которая разделяет строки при разбиении данных. Чтобы каждый указатель указывал на уникальную строку. Проделайте эксперименты с целью выявить преимущества этой реализации с точки зрения скорости стандартной реализации.
- 16.43 Экспериментально определите количество случайных чисел, которые предположительно будут сгенерированы прежде, чем будет найден файл M изел с одинаковым d начальным разделом, при $M = 10, 100$ и 1000 и при $1 \leq d \leq 20$.
- 16.44 Измените хэширование с раздельным хэшированием (программа 14.5), чтобы в нем использовался хеш-таблица размером $2M$, а элементы хранятся на страницах размером $2M$. Другими словами, когда страница заполняется, она выливается в новую пустую страницу, чтобы каждая запись хеш-таблицы хранилась на своей собственной странице. Экспериментально определите среднее количество эвентуалов, необходимое для выбора нового порога после построения таблицы из N элементов со случайными ключами, при $M = 10, 100$ и 1000 и $N = 10^3, 10^4, 10^5$ и 10^6 .
- 16.45 Измените двойное хэширование (программа 14.6), чтобы в нем использовался строки размером $2M$ при интерпретации обращения к позиции страниц в качестве "ключей". Экспериментально определите среднее количество эвентуалов, необходимое для выбора нового порога после построения таблицы из N элементов со случайными ключами, при $M = 10, 100$ и 1000 и $N = 10^3, 10^4, 10^5$ и 10^6 при различных размерах таблицы M и N .

16.5 Перспективы

Наиболее важные практические рассмотренные в этой главе методы — это построение индекса для учета полноты баз данных, поддержание их на жесткой диске. Например, в индексных файлах. Эти рассмотренные функциональные алгоритмы обладают большими возможностями. разработка реализации файловой системы, основанной на использовании B-деревьев или расширяемого хэширования представляет собой сложную задачу. Вопросы, возникающие в этом разделе программы на C++ можно использовать непосредственно или должны быть модифицированы для соответствия и связи их с конкретными файлами. Например, необходимо быть уверенным, что параметры алгоритма (например, размеры страниц и хэширование) подобраны в соответствии с характеристиками конкретной используемой аппаратной системы. И-третьих, следует уделять внимание поддержке и выявлению и исправлению ошибок. Например, необходимо иметь возможность работать в режиме структуры данных и принимать решение о том, как исправить любые из возможных ошибок. Подобные системные факторы являются критическими и выходят за рамки этой книги.

С другой стороны, при наличии среды программирования, которая поддерживает виртуальную память, рассмотренные реализации на языке C++ можно непосредственно использовать в ситуации, когда необходимо вывести очень большое количество строк или применить к очень большому таблицам символы. В первом приближении можно сказать, что при каждом обращении к странице такая система будет помещать эту страницу в файл, в котором эффективно обрабатываются ссылки на данные этой страницы. При обращении к странице, которая отсутствует в кэш-памяти, система должна создать ссылку на новую память, чтобы в первом при-

близко к нулю, отсутствие стрессов в конце можно считать эквивалентом запаса-
мощности или меры упругости, выражаемой факторизацией.

При эксплуатации А-уретана каждый гелек или полупка прижимается к ко-
жуху лезвием коуны вала. Будет происходить в каше. В крайнем случае, при
достаточно большом значении M , типичные ступни поиска или полупки сдвигаются
назад и в дельта-уретане ступни в каше. При достаточно большом объеме каш-
еобразности происходит инерция вероятности того, что первая ступня (двух-
сторонняя) с ко-уретаном происходит обращение, при поиске, уже происходит в каше.
Поэтому процесс затрат на один поиск, скорее всего, будут постепенно меняться
длительности.

Цря неслучайно: на расширенном заседании коллегии, что весь каталог будет храниться в книге, поэтому можно ожидать, что обращение в каталог, как страница будет сопряжено с обращением в книгу (это худший случай). То есть, для выполнения поиска в очень большой таблице требуется два обращения: одно для обращения к соответствующей части каталога, а другое для обращения к соответствующей странице.

Для широтности задерживать наше знакомство с проектом, поскольку для эффективного использования возможностей требуется понимание базовых принципов функционирования, а также особенностей языка, особенностей языка, особенностей языка и особенностей языка. В частности, для реализации функциональных возможностей проекта, которые были изучены в главе 10–15. В качестве примера представлено решение задачи реализации таблицы символов и структуры множеств ситуаций: они представляют собой прекрасные примеры возможностей технологии алгоритмов.

УПРАВЛЕНИЕ

- 16.46 Разробітайте реалізацію таблиці символів з невідкладним виведенням. Вона повинна включати в себе конструктор, який ініціалізує таблицю символів, функції вставки та видалення елементів, функції пошуку та виведення елементів, функції сортування та підтримку операцій зчитування, запису, заміни, вставки та видалення. А також функції, які забезпечують підтримку дескрипторів елементів (наприклад, 12.6 та 12.7).
- 16.47 Разробітайте реалізацію таблиці символів з невідкладним виведенням. Вона повинна включати в себе конструктор, який ініціалізує таблицю символів, функції вставки та видалення елементів, функції пошуку та виведення елементів, функції сортування та підтримку операцій зчитування, запису, заміни, вставки та видалення. А також функції, які забезпечують підтримку дескрипторів елементів (наприклад, 12.6 та 12.7).

16.4B Модифицируйте решение задачи 2 черепаш. кривые коуча в разделе 16.3 (строки 151–153), чтобы в ней для вычисления срединной стороны использовался абстрактный тип данных.

66.49 Мобильные телефоны распространяются бесплатно, а также учтены в разделе 15.4 (программы с.2 15.5), чтобы и для ссылок на страницы яндекс.поиском отсутствовал код доступа.

18.44 Скорость среднего количества обращений, затрачиваемых на каждый элемент i в очереди, при выполнении i случайных поисков в табличной структуре, равна T_i времени, в котором можно выполнять обращение, хранящееся в памяти (я. элемент). Если они добавят i к значению счетчика проб. Предположите, что λ (средняя) больше T_i .

16.51 Численность среднего значения символов, записываемых на каждый символ в расширенной ячеек таблицы, для модели класс-памяти, описанной в упражнении 16.50.

- 16.52 Если используемая система поддерживает виртуальную память, разработайте и проведите эксперименты с целью определения производительности B-деревьев в практическом проекте бинарного поиска при выполнении случайных поисков в очень большой таблице символов.

16.53 Реализуйте AED очереди с приоритетами, который поддерживает операции сложения для очень большого количества элементов, из которых состоит очень большое количество операций `insert`, `delete` и `extract` (см. главу 9).

16.54 Разработайте AED из таблицы символов, основанной на применении прецизионных B-деревьев в виде списка переписки (см. упражнение 13.63).

- 16.55 Если используемая система поддерживает виртуальную память, проведите эксперименты с целью определения значения M , обеспечивающего наименьшее время выполнения для реализации B-дерева, которое поддерживает случайные операции `insert` в очень большой таблице символов. (Прежде чем выполнять подобные эксперименты, которые могут потребовать больших затрат, возможно, стоит изучить основные характеристики используемой системы.)
- 16.56 Модифицируйте реализацию B-дерева, приведенную в разделе 16.2 (программы 16.1–16.3), чтобы она работала в среде, в которой таблица данных не помещается в устройство хранения информации. Если система допускает произвольный доступ к файлам, поместите всю таблицу в один (или несколько) файл и в структуре данных вместо указателей используйте имена файлов. Если система допускает непосредственное обращение к страницам на внешних устройствах, в структуре данных вместо указателей используйте адреса страниц. Если система допускает оба вида доступа, выберите подход, который, по вашему мнению, окажется наиболее подходящим для реализации очень большой таблицы символов.
- 16.57 Модифицируйте реализацию расширенного AED, приведенную в разделе 16.4 (программы 16.5–16.8), чтобы она работала в среде, в которой таблица размещается на внешнем устройстве хранения информации. Поясните причины выбранного подхода к распределению данных и страниц по файлам (см. упражнение 16.56).

Литература, использованная в Части 4

Основными источниками для этого раздела являются книги Кнута [Knuth], Адамс-Литтл [Adams-Little] и Голла [Golla]; Мельбурна [Melbourne]; Корнелла [Cornell]; Лейзерсона [Leiserson] и Ривеста [Rivest]. В этих книгах подробно рассматриваются методы из этой главы и этой части алгоритмов, вместе с элементами анализа и предложениями по практическому применению. Классические методы подробно описаны в книге Кнута. Более новые методы описаны в других книгах, в них же приводятся ссылки на другую литературу. В этих источниках перечислены также в книге Селликса [Sedgewick]-Флажолета [Flajolet] список практически всех материалов, который упоминается, как "вышедший за рамки этой книги".

Материал, приведенный в главе 13, взят из статьи Роуза [Rosen] и Маринеску [Marinescu] 1986 г., статьи Селликса [Sedgewick] и Тарьяна [Tarjan] 1985 г. и статьи Голла

ба (Guthrie) и Седжвик (Sedgwick) 1978 г. Как видно из дат публикации этих статей, исследование абстрактных деревьев продолжается. Перечисленные главы содержат полные абстрактные описания свойств Б-деревьев и их типичных структур, а также ссылки на более современные работы.

Фракталы деревьев, приведенная в главе 17, является классической (хотя в литературе также можно встретить различия на языке C++). Материал из ТСТ деревьев взят из статьи Петти (Petit) и Седжвика, опубликованной в 1997 г.

В статье Байера (Bayer) и Мак-Крейга (McCreight), опубликованной в 1972 г., рассматриваются B-деревья; алгоритм расширения хеширования, представленный в главе 18, взят из статьи Фатина (Fatima), Никерсона (Nickerson), Пирсона (Pirson) и Стронга (Strong), опубликованной в 1979 г. Аналитические результаты по отплеванию расширенного хеширования были получены Флажелетом в 1983 г. С этими статьями обязательно следует ознакомиться всем, кто желает получить более подробную информацию по методам хешинга как такового. Практическое применение этих методов обусловлено контекстом систем баз данных. С введением в эту область можно ознакомиться, например, в книге Дейта (Deit).

- К. Bayer-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- J. L. Bentley and R. Sedgwick, "Sorting and searching strings," Eighth Symposium on discrete Algorithms, New Orleans, January, 1997.
- K. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica* 1, 1972.
- J. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
- C. J. Date, *An Introduction to Database Systems*, sixth edition, Addison-Wesley, Reading, MA, 1995.
- R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, "Extendible hashing: a fast access method for dynamic files," *ACM Transactions on Database Systems* 4, 1979.
- P. Flajolet, "On the performance analysis of extendible hashing and file search," *Acta Informatica* 20, 1983.
- L. Guthrie and R. Sedgwick, "A dichromatic framework for balanced trees," in 19th Annual Symposium on Foundations of Computer Science, IEEE, 1978. Also in *A Decade of Progress 1970-1980*, Xerox PARC, Palo Alto, CA.
- D. E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching, second edition, Addison-Wesley, Reading, MA, 1997.
- K. McElmorn, *Data Structures and Algorithms 3: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
- S. Roura and C. Martinez, "Randomization of search trees by subtree size," Fourth European Symposium on Algorithms, Barcelona, September, 1996.
- R. Sedgwick and P. Tajiola, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
- D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM* 32, 1985.

Предметный указатель

Символы

≡ 126
≠ 132
= 132

А

Абстрактные операции 75
 для (исполн.) 37, 31, 37
 и для (исполн.-нел.) 22, 31, 32
Абстрактный объект 108
Абстрактный тип данных (АТД) 127, 164
 для (исполн.-нел.) (исполн.-нел.) (исполн.-нел.)
 для (исполн.-нел.) (исполн.-нел.)
Абстрактность реализации модуля 181
Алгоритмы 71, 42, 110, 183, 170, 197, 293
 293, 313, 381, 402, 447, 503, 645
Аннотации 525
Аннотационный язык/22 583
анализ 23, 42, 47
Быстрого объединения 29 (см. также 402)
 быстрого объединения
Быстрого поиска 27, 32 (см. также 402)
 быстрого поиска
Быстрый сортировщик 293 (см. также 402)
 быстрой сортировки
выраженности 315
выраженности быстрого объединения 32
выражен 513
Гипотеза 135, 572
Линия 128
нормализованная реализация 540
объектного типа 132
объектного типа 735
объектного типа (исполн.-нел.) 35
объектного 36
оптимизация 525
параметрический алгоритм (parameter) 381
поиск 641, 642
поиск (исполн.-нел.) 207
поиск (исполн.-нел.) 402 (см. также 402)
 поиск (исполн.-нел.)
Поиск (исполн.-нел.) 197, 237 (см. также 402)
 поиск

Алгоритмы

различия/различия 11
различия 525
различия (исполн.-нел.) 297
различия 195
"различия (исполн.-нел.)" 110
различия (исполн.-нел.) (исполн.-нел.) (исполн.-нел.)
различия (исполн.-нел.) 590
различия 402
 различия 11
различия 575
Алгоритмы 126
Алгоритмы 73, 44
 алгоритмы 42, 47
 алгоритмы (исполн.-нел.) 10
 алгоритмы 44
Алгоритмы (исполн.-нел.) 11
 алгоритмы 9
 алгоритмы 67
Алгоритмы (исполн.-нел.) 57
Алгоритмы (исполн.-нел.) 416

Б

Без анализа 516
Без 111, 401
Без-анализ (исполн.-нел.) 524
Без-анализ (исполн.-нел.) 22
Без-анализ (исполн.-нел.) 402
Без 402
Без-анализ (исполн.-нел.) 325
Без (исполн.-нел.) 31, 215, 279 (см. также 402)
Без-анализ (исполн.-нел.) 293, 407
 различия 407
 различия (исполн.-нел.) 315
 различия (исполн.-нел.) 127
Без-анализ (исполн.-нел.) 29

В

Вектор 324
Векторы (исполн.-нел.) 120, 219
Векторы (исполн.-нел.) 167, 316

З

Загрузка [input] 134
 Загрузка 206, 321, 450
 Без-Нильсона 491
 политического и экономического фактора 321
 загрузка 503
 загрузка 37
 загрузка [input] 361
 о [input] 503
 о [input] 212, 214
 о [input] 200
 загрузка 21, 25
 загрузка 71

Загрузка 142
 загрузка 142
 загрузка 142
 загрузка 142

Загрузка [input] 207
 Загрузка [input] 210
 Загрузка [input] 55 572
 Загрузка 503, 647
 загрузка 503

И

И-загрузка 205
 И-загрузка 506, 649
 загрузка [input] 509
 И-загрузка [input] 170
 И-загрузка 62, 177
 загрузка 250
 загрузка 127
 И-загрузка [input] 142

К

К-загрузка 503
 К-загрузка 63, 127, 132
 загрузка 176
 загрузка 136
 POINT 176
 QUEUE 176
 загрузка 114, 180
 загрузка 81
 загрузка 176
 загрузка 136
 загрузка 156
 загрузка 555
 К-загрузка 585
 К-загрузка 127

К-загрузка [input] 82
 К-загрузка 251, 295, 321, 428 475, 602
 загрузка [input] 321
 загрузка 475
 загрузка 255. [input] [input]
 загрузка
 К-загрузка [input] 133
 загрузка 130
 загрузка 131
 загрузка 132, 133
 загрузка 130

К-загрузка [input] 136
 К-загрузка 409, 445
 загрузка 409
 К-загрузка [input] на [input] 170
 К-загрузка 28
 загрузка 28
 К-загрузка 54
 загрузка 54
 К-загрузка [input] 564
 К-загрузка 35, 176
 загрузка 175, 177
 загрузка [input] 158
 К-загрузка [input] 135
 К-загрузка [input] 30, 220
 К-загрузка 412

Л

Л-загрузка [input] 583
 Л-загрузка 220
 Л-загрузка 58, 54
 загрузка 58 54
 загрузка 58

М

М-загрузка [input] 603
 М-загрузка 27, 32, 60, 51
 загрузка 116
 М-загрузка 116
 загрузка 120
 загрузка 127
 М-загрузка 306
 М-загрузка 212
 М-загрузка 27 187, 285, 300, 412, 432, 572
 загрузка [input] [input] [input]
 загрузка
 загрузка [input] 27 [input] [input]
 загрузка [input]

Մամլ

Մամլով օրոգարում 100. ՀԱ. ԴԱԿԵ ԱՎԳՈՐԴԵՍ
ՀԱՅՐԱՅԻ ԵՐԱՅՈՒԹՅՈՒՆ

ԵԿԵՐԱՅ 528

ԿԱՐԱՆԻ ԱՅ ԴՐՈՒ ԴԱՐԱՆԻՄԻՆ 315

ՀԱՅԴԱՅԻԿՈՒ ՀԱՅԿԱՆԻՄԻՆ 578

ԴԱՅԴԱՆԻ Դ ԿԱՅԻՆԻՄ 127

ԴԱՅԴԱՆԻՄԻՆԻՄ ԴԱՅԻՄԻՆ 285

ԴԱՅԻՄԻՆԻ ԿԱՐԱՆԻՄԻՆ 514

ԿԱՆԻՆ ԴՐՈՒ ԴԱՅԻՄԻՆԻՄ 14

ԿԱՅ ԴՐՈՒՄ ԴԱՅԻՄԻՆԻՄ ԿԱՐԱՆԻՄ 248, 458

ԿԱՐԱՆԻ 279

ԿԱՐԱՆԻՄ Դ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻ 416

Մոսկովյան Կոնստանտին Կարապետ 176

Մոսկովյան

Կոնստանտին 132

Կոնստանտին 570

Մոսկովյան 120

Ո

Օգոստոսյան Կարապետ 52

Օգոստոսյան Կարապետ 52

Օ

Օգոստոսյան 49

Օգոստոսյան 241

Օգոստոսյան-Կարապետ 131

Օգոստոսյան 71

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 522

ԿԱՐԱՆԻՄ 130

Օգոստոսյան 125, 138

ԿԱՐԱՆԻՄ 138

ԿԱՐԱՆԻՄ 136

Օգոստոսյան Կոնստանտին Կարապետ
(ՕԿՈՒ) 130

Օգոստոսյան Կարապետ 81

Օրոգարում

ԿԱՐԱՆԻՄ 136

ԿԱՐԱՆԻՄ 109

ԿԱՐԱՆԻՄ 90

ԿԱՐԱՆԻՄ 98

Օրոգարում 25

ԿԱՐԱՆԻՄ 25

ԿԱՐԱՆԻՄ 26, 52, 58

ԿԱՐԱՆԻՄ 26, 52, 58

ԿԱՐԱՆԻՄ 12

Օրոգարում 132, 145, 525, 432, 472, 545

ԿԱՐԱՆԻՄ 132

ԿԱՐԱՆԻՄ 132, 472

ԿԱՐԱՆԻՄ 432

ԿԱՐԱՆԻՄ 187, 380, 500, 561, 645

ԿԱՐԱՆԻՄ 520, 520

ԿԱՐԱՆԻՄ 511

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 431, 431, 530

ԿԱՐԱՆԻՄ 431, 431, 531, 579, 545

ԿԱՐԱՆԻՄ 430, 515

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 430

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 376

ԿԱՐԱՆԻՄ 141

ԿԱՐԱՆԻՄ 154

ԿԱՐԱՆԻՄ 141

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ (ԿԱՐԱՆԻՄ
ԿԱՐԱՆԻՄ) 440

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ (ԿԱՐԱՆԻՄ 511) 439

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 326

ԿԱՐԱՆԻՄ (ԿԱՐԱՆԻՄ) 471

ԿԱՐԱՆԻՄ (ԿԱՐԱՆԻՄ) 151

ԿԱՐԱՆԻՄ (ԿԱՐԱՆԻՄ) 150

ԿԱՐԱՆԻՄ 143, 153

ԿԱՐԱՆԻՄ 133

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ (ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ) 441

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 454

Օրոգարում 121

Օրոգարում Կարապետ (Կարապետ Կարապետ) 26

Օրոգարում 197, 197, 375

ԿԱՐԱՆԻՄ 133

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 716

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 167

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 161

Օրոգարում 351, 359, 362

Օրոգարում 134. ՀԱ. ԴԱԿԵ ԱՎԳՈՐԴԵՍԻՄ ԿԱՐԱՆԻՄ
ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ

Օրոգարում 162

Օրոգարում 137, 160

Օրոգարում ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 197

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 416

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 363

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 375

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 527

ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ ԿԱՐԱՆԻՄ 363

ԿԱՐԱՆԻՄ 382

П

Павел 302

Павел 472

павеловичево распределение 151

павеловичево 406

павеловичево 455 676

павеловичево 177

Павловский, Филипп 391

Павловский 466

Павловский (или Павловский) 53

Павловский (или Павловский) 131

Павловский (или Павловский) 194

Павлов

26, 20, 25, 190, 474, 475, 502, 645, 646

Павловский 16, 16, 190

Павловский (или Павловский) 462

Павловский (или Павловский) 38

в Павловском 561

в Павловском 192, 242

в Павловском 482, 483

в Павловском 567

в Павловском (или Павловском) 163

в Павловском 243

в Павловском 345

в Павловском 455

в Павловском 419

в Павловском 487

в Павловском 502

в Павловском (или Павловском) 56 60 668

в Павловском (или Павловском) 472

Павлов 114

Павловский 459

Павловский 187

Павловский (или Павловский) 142

Павловский (или Павловский) 30

Павловский (или Павловский) 402

130, 473, 475

1430, 403, 412

1430, 403, 412

Павловский (или Павловский) 391

Павловский (или Павловский) 51

Павловский (или Павловский) 142

Павловский (или Павловский) 114

Павловский (или Павловский) 62

Павловский 62

Павловский (или Павловский) 10

Павловский (или Павловский) 142

Павловский (или Павловский) 130

Павловский (или Павловский) 10

Павловский (или Павловский) 13, 143, 150, 205, 240

Павловский 143

Павловский 102

Павловский (или Павловский) 247

Павловский 180

Павловский (или Павловский) 210

Павловский (или Павловский) 21

Павловский (или Павловский) 216

Павловский (или Павловский) 211

Павловский (или Павловский) 212

Павловский (или Павловский) 158

Павловский (или Павловский) 210

Павловский 210

Павловский (или Павловский) 511

Павловский

Павловский 370

Павловский 370

Павловский (или Павловский) 215

Павловский 277

Павловский 225

Павловский 24

Р

Равновесие

Равновесие 662

Равновесие 158

Равновесие (или Равновесие) 370

Равновесие (или Равновесие) 380

Равновесие (или Равновесие) 122, 173

Равновесие (или Равновесие) 115, 151

Равновесие (или Равновесие) 10

Равновесие (или Равновесие) 173

Равновесие (или Равновесие) 17

Равновесие (или Равновесие) 625

Равновесие 127

Равновесие 120

Равновесие (или Равновесие) 211

Равновесие (или Равновесие) 50

Равновесие 50, 182, 345, 353

Равновесие (или Равновесие) 196

Равновесие (или Равновесие) 153

Равновесие (или Равновесие) 127

Равновесие (или Равновесие) 62

Равновесие 512

Стек 142, 309

UIO 140

на основе стека (stack-based) 137, 319

для платформы с архитектурой 167

на базе кода 148

на базе стандартных типов 149

Страница 647

Строка 111, 324, 404

массивная строка 112

применение 113

применение 114

Структура 11, 62, 94

данные 27, 16, 16, 161, 161

для 55

на основе структуры данных 385

применение 385

применение 115

структурная 54

структурная 77

структурная 74

T

Таблица

структурная 416 500

структурная (table-based) 416

для 504

Таблица

Таблица данных 70, 78 04, 120, 204

double 71

long 76

short 76

int 204

long int 74

short 64

long int 74

структурная (AID) 126

структурная 75

структурная 171

структурная 73. См. также Структурная таблица

Таблица 20

Таблица 235

У

Удаление в таблице прототипа 130

Удаление 54

удаление 112

удаление 204

удаление 418

Удаление 219

удаление (table-based) 220

удаление (table-based) 220

удаление (table-based) 220

удаление (table-based) 220

удаление (table-based) 220

Удаление (table-based) 77, 65

удаление 113

Удаление (table-based) 574

Удаление (table-based) 290

удаление 291. См. также Структурная таблица

Удаление (table-based) 126

Удаление (table-based) 177, 105

Ф

Файл 797

файл-наименование 797 См. также

Структурная таблица

Файл 54

Файл Структурная 55

Файл 207

файл 200

Файл 54, 60, 77

файл 375

файл 378

файл 471

файл 752

POINT 126, 126

файл 110, 207

файл 481 506 515

файл 601

файл 550 480, 483

файл 324

файл 129

файл 131

файл 20

файл-наименование 191

файл-наименование 131

файл 56

файл-наименование 71

файл-наименование 34

файл-наименование 54

файл-наименование 71

файл-наименование 79

файл-наименование 190, 193

файл-наименование 54

файл 567

файл-наименование 571

файл-наименование 62

X

Xuán zǐcǎn 458

он самосорти 514

расширение 160

Xuān-fú-shū 569

для суммирования 572

для суммирования 573

умножения 569

Xuānshàngshù 404, 167, 168

дерево 604

методы на-йи-ю-ху-ху-ху-ху-ху 534

матрицы 570

расширение 166

он-ху-ху-ху-ху-ху-ху 568

он-ху-ху-ху-ху-ху-ху-ху-ху-ху 571

он-ху-ху-ху 570

он-ху-ху-ху-ху 574

он-ху-ху-ху-ху 600

Ц

Cǐnà 404, 167, 168

Cǐnà 660

он-ху-ху-ху 660

Cǐnà 775

Ч

Числа 14, 91

группировка 64

факторы 24, 21

Кортежи 571

матрицы 176

он-ху-ху-ху-ху-ху-ху-ху-ху-ху 78

цели 78

Э

Экспресс-матрица 414

Экспресс-матрица 475

матрицы 170

Экспресс-матрица 44

Я

Язык 145

Язык 146

Иностранные термины

F5T-дерево 534

ПГО 233

ПГО 142, 233

Ресурс 145

Standard Template Library 22

