

Фундаментальные алгоритмы и структуры данных в Delphi

для всех версий Delphi и Kylix!

- ◆ Массивы и списки
- ◆ Очереди и деревья
- ◆ Сортировка
- ◆ Поиск
- ◆ Рандомизация
- ◆ Хеширование
- ◆ Алгоритмы на графах
- ◆ Конечные автоматы
- ◆ Технологии сжатия
- ◆ Производительность



Эту книгу я с нетерпением ждал на протяжении нескольких лет. Книга превзошла все мои самые смелые ожидания, поскольку оказалась, с одной стороны, отличной книгой по фундаментальным алгоритмам и структурам данных в Delphi и Kylix, а, с другой – совершенно независимой от версии этой среды разработки.

Огромное преимущество книги заключается в простоте и изяществе изложения материала, а также практической применимости всего кода в Delphi, Kylix и даже C++Builder. Я рекомендую эту книгу всем, кто считает себя мало-мальски разработчиком прикладных приложений. Я сам частенько перечитываю эту книгу.

Боб Сварт, независимый эксперт и консультант, известный также как доктор Боб

Фундаментальные алгоритмы и структуры данных в Delphi™

Джулиан М. Бакнелл

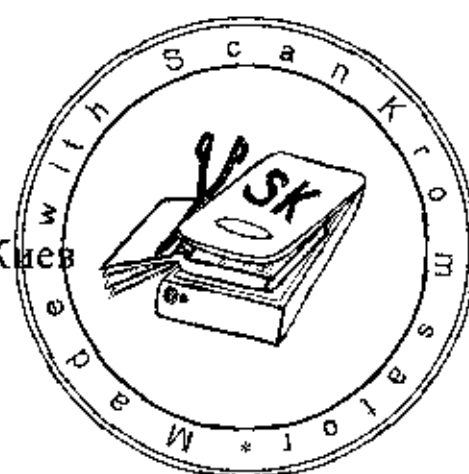


торгово-издательский дом

DiaSoft

Москва • Санкт-Петербург • Киев

2003



The Tomes of Delphi™ Algorithms and Data Structures

Julian Bucknall



Wordware Publishing, Inc.

ББК 32.973.2

УДК 681.3.06(075)

Б 90

Бакнелл Джулиан М.

Б 90 **Фундаментальные алгоритмы и структуры данных в Delphi: Пер. с англ./Джулиан М. Бакнелл. — СПб: ООО «ДиаСофтЮП», 2003.— 560 с.**

ISBN 5-93772-087-3

Книга *Фундаментальные алгоритмы и структуры данных в Delphi* представляет собой уникальное учебное и справочное пособие по наиболее распространенным алгоритмам манипулирования данными, которые зарекомендовали себя как надежные и проверенные многими поколениями программистов. По данным журнала *Delphi Informant* за 2002 год, эта книга была признана сообществом разработчиков прикладных приложений на Delphi как «самая лучшая книга по практическому применению всех версий Delphi». В книге подробно рассматриваются базовые понятия алгоритмов и основополагающие структуры данных, алгоритмы сортировки, поиска, хеширования, синтаксического разбора, сжатия данных, а также многие другие темы, тесно связанные с прикладным программированием. Изобилие тщательно проверенных примеров кода существенно ускоряет не только освоение фундаментальных алгоритмов, но также и способствует более квалифицированному подходу к повседневному программированию. Несмотря на то что книга рассчитана в первую очередь на профессиональных разработчиков приложений на Delphi, она окажет несомненную пользу и начинающим программистам, демонстрируя им приемы и трюки, которые столь популярны у истинных «профи». Все коды примеров, упомянутые в книге, доступны для загрузки на Web-сайте издательства.

ББК 32.973.2

Original English language edition published by Wordware Publishing, Inc. Co., 2320 Los Rios Blvd Suite 200, Plano, Texas USA 75074. Copyright © 2002 by Wordware Publishing, Inc. Co..

Russian-language edition copyright © 2003 by Diasoft Publishing House. All rights reserved.

Лицензия предоставлена издательством Wordware Publishing, Inc. Co.

Все права зарезервированы, включая право на полное или частичное воспроизведение в какой бы то ни было форме.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно остается, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

Все торговые знаки, упомянутые в настоящем издании, зарегистрированы. Случайное неправильное использование или пропуск торгового знака или названия его законного владельца не должно рассматриваться как нарушение прав собственности.

ISBN 5-93772-087-3 (рус.) © Перевод на русский язык. ООО «ДиаСофтЮП», 2003

ISBN 1-55622-736-1 (англ.) © Wordware Publishing, Inc. Co., 2002

© Оформление. ООО «ДиаСофтЮП», 2003

Гигиеническое заключение № 77.99.6.953.П.438.2.99 от 04.02.1999



Краткое оглавление

Введение	11
Глава 1. Что такое алгоритм?	20
Глава 2. Массивы	47
Глава 3. Связные списки, стеки и очереди	84
Глава 4. Поиск	136
Глава 5. Сортировка	153
Глава 6. Рандомизированные алгоритмы	205
Глава 7. Хеширование и хеш-таблицы	249
Глава 8. Бинарные деревья	303
Глава 9. Очереди по приоритету и пирамидальная сортировка	359
Глава 10. Конечные автоматы и регулярные выражения	388
Глава 11. Сжатие данных	442
Глава 12. Дополнительные темы	502
Предметный указатель	554

Оглавление

Введение	11
Глава 1. Что такое алгоритм?	20
Что такое алгоритм?	20
Анализ алгоритмов	22
О-нотация	25
Лучший, средний и худший случаи	27
Алгоритмы и платформы	27
Виртуальная память и страничная организация памяти	28
Пробуксовка	29
Локальность ссылок	30
Кэш процессора	31
Выравнивание данных	31
Пространство или время	33
Длинные строки	35
Использование ключевого слова <code>const</code>	36
Осторожность в отношении автоматического преобразования типов	37
Тестирование и отладка	37
Утверждения	38
Комментарии	41
Протоколирование	41
Трассировка	41
Анализ покрытия	42
Тестирование модулей	42
Отладка	44
Резюме	46
Глава 2. Массивы	47
Массивы	47
Типы массивов в Delphi	48
Стандартные массивы	48
Динамические массивы	52
Новые динамические массивы	60
Класс TList, массив указателей	61
Краткий обзор класса TList	61
Класс TtdObjectList	63
Массивы на диске	69
Резюме	83

Глава 3. Связные списки, стеки и очереди 84

Односвязные списки 84

 Узлы связного списка 86

 Создание односвязного списка 86

 Вставка и удаление элементов в односвязном списке 87

 Прохождение связного списка 88

 Соображения по поводу эффективности 90

 Использование начального узла 90

 Использование диспетчера узлов 91

 Класс односвязного списка 97

Двухсвязные списки 105

 Вставка и удаление элементов в двухсвязном списке 106

 Соображения по поводу эффективности 109

 Использование начального и конечного узлов 109

 Использование диспетчера узлов 109

 Класс двухсвязного списка 109

Достоинства и недостатки связных списков 117

Стеки 118

 Стеки на основе односвязных списков 118

 Стеки на основе массивов 121

 Пример использования стека 124

Очереди 126

 Очереди на основе односвязных списков 126

 Очереди на основе массивов 130

Резюме 135

Глава 4. Поиск 136

Процедуры сравнения 136

Последовательный поиск 139

 Массивы 139

 Связные списки 144

Бинарный поиск 145

 Массивы 145

 Связные списки 147

 Вставка элемента в отсортированный контейнер 150

Резюме 152

Глава 5. Сортировка 153

Алгоритмы сортировки 153

 Тасование массива TList 156

Основы сортировки 158

 Самые медленные алгоритмы сортировки 158

 Пузырьковая сортировка 159

 Шейкер-сортировка 161

Сортировка методом выбора	162
Сортировка методом вставок	163
Быстрые алгоритмы сортировки	167
Сортировка методом Шелла	167
Сортировка методом прочесывания	170
Самые быстрые алгоритмы сортировки	173
Сортировка слиянием	173
Быстрая сортировка	182
Сортировка слиянием для связанных списков	198
Резюме	204
Глава 6. Рандомизированные алгоритмы	205
Генерация случайных чисел	206
Критерий хи-квадрат	207
Метод средних квадратов	211
Линейный конгруэнтный метод	212
Тестирование	217
Тест на однородность	217
Тест на пропуски	218
Тест "покер"	220
Тест "сбор купонов"	221
Результаты выполнения тестов	222
Комбинирование генераторов	224
Аддитивные генераторы	226
Тасующие генераторы	228
Выводы по алгоритмам генерации случайных чисел	230
Другие распределения случайных чисел	231
Списки с пропусками	233
Поиск в списке с пропусками	234
Вставка в список с пропусками	238
Удаление из списка с пропусками	241
Полная реализация класса связанного списка	243
Резюме	248
Глава 7. Хеширование и хеш-таблицы	249
Функции хеширования	251
Простая функция хеширования для строк	252
Функции хеширования PJW	253
Разрешение конфликтов посредством линейного зондирования	255
Преимущества и недостатки линейного зондирования	256
Удаление элементов из хеш-таблицы с линейным зондированием	258
Класс хеш-таблиц с линейным зондированием	260
Другие схемы открытой адресации	270
Квадратичное зондирование	270

Псевдослучайное зондирование	270
Двойное хеширование	271
Разрешение конфликтов посредством связывания	272
Преимущества и недостатки связывания	273
Класс связанных хеш-таблиц	274
Разрешение конфликтов посредством группирования	284
Хеш-таблицы на диске	285
Расширяемое хеширование	286
Резюме	302
Глава 8. Бинарные деревья	303
Создание бинарного дерева	305
Вставка и удаление с использованием бинарного дерева	305
Перемещение по бинарному дереву	308
Обход в ширину, симметричный обход и обход в глубину	308
Обход по уровням	315
Реализация класса бинарных деревьев	316
Деревья бинарного поиска	322
Вставка в дерево бинарного поиска	325
Удаление из дерева бинарного поиска	327
Реализация класса дерева бинарного поиска	330
Перекомпоновка дерева бинарного поиска	331
Скошенные деревья	335
Реализация класса скошенного дерева	336
Красно-черные деревья	340
Вставка в красно-черное дерево	342
Удаление из красно-черного дерева	347
Резюме	358
Глава 9. Очереди по приоритету и пирамидальная сортировка	359
Очередь по приоритету	360
Первая простая реализация	360
Вторая простая реализация	363
Сортирующее дерево	366
Вставка в сортирующее дерево	366
Удаление из сортирующего дерева	367
Реализация очереди по приоритету при помощи сортирующего дерева	369
Пирамидальная сортировка	374
Алгоритм Флойда	374
Завершение пирамидальной сортировки	375
Расширение очереди по приоритету	378
Восстановление свойства пирамидальности	379
Отыскание произвольного элемента в сортирующем дереве	380

Реализация расширенной очереди по приоритету	381
Резюме	387
Глава 10. Конечные автоматы и регулярные выражения	388
Конечные автоматы	388
Использование конечного автомата: синтаксический анализ	388
Синтаксический анализ файлов с разделяющими запятыми	394
Детерминированные и недетерминированные конечные автоматы	398
Регулярные выражения	410
Использование регулярных выражений	412
Синтаксический анализ регулярных выражений	412
Компиляция регулярных выражений	420
Сопоставление строк с регулярными выражениями	433
Резюме	441
Глава 11. Сжатие данных	442
Представление данных	442
Сжатие данных	443
Типы сжатия	443
Потоки битов	444
Сжатие с минимальной избыточностью	449
Кодирование Шеннона-Фано	449
Кодирование Хаффмана	454
Кодирование с использованием скошенного дерева	469
Сжатие с использованием словаря	479
Описание сжатия LZ77	479
Особенности кодирования литеральных символов и пар расстояние/длина	482
Восстановление с применением алгоритма LZ77	483
Сжатие LZ77	490
Резюме	501
Глава 12. Дополнительные темы	502
Алгоритм считывания-записи	502
Алгоритм производителей-потребителей	512
Модель с одним производителем и одним потребителем	513
Модель с одним производителем и несколькими потребителями	521
Поиск различий между двумя файлами	532
Вычисление LCS двух строк	532
Вычисление LCS двух файлов	547
Резюме	551
Эпилог	551
Список литературы	552
Предметный указатель	554

Введение

Вы взяли в руки эту книгу, она вас чем-то заинтересовала, и вы даже подумываете, не купить ли ее?.. В голове возникают мысли наподобие: "Да, наверное, стоит купить, однако прежде бы выяснить ряд вопросов..."

Почему книга посвящена алгоритмам именно на Delphi?

Несмотря на существование относительно большого количества книг, посвященных алгоритмам, очень и очень немногие из них отходят от стандартного начального курса в рамках компьютерной инженерии при изложении фундаментальных алгоритмов для практического применения. Коды, приводимые в таких книгах, зачастую относятся только к конкретному рассматриваемому алгоритму, без каких-либо соображений по поводу его практической реализации в среде реальных бизнес-приложений. Хуже того, с точки зрения разработчиков этих самых бизнес-приложений, существует немало книг из числа используемых в качестве учебных пособий в колледжах и университетах, в которых опущено множество интересных тем, или, в крайнем случае, они оставлены читателям на самостоятельную проработку, в виде упражнений, зачастую без указания правильных ответов.

Разумеется, для описания большинства алгоритмов в подобного рода книгах не используется Delphi, Kylix или Pascal. Некоторые авторы предпочитают для описания алгоритмов пользоваться псевдокодом, некоторые – языком C, другие выбирают для этих целей язык C++, а часть авторов – вообще какой-либо суперсовременный язык. В самой знаменитой, давно и часто используемой книге, посвященной алгоритмам, для иллюстрации самих алгоритмов выбран язык ассемблера, которого вообще не существует (язык ассемблера MIX в книге "Искусство программирования для ЭВМ" Дональда Кнута [11, 12, 13]). Действительно, в книгах, содержащих в своих названиях слово "практический", для иллюстрации реализации алгоритмов используются языки C, C++ и Java. Является ли это большой проблемой? В конце концов, алгоритм – это алгоритм, стало быть, какая разница, на чем демонстрировать его работу? И зачем, собственно, покупать книгу, посвященную алгоритмам с иллюстрациями на Delphi?

Я утверждаю, что на сегодняшний день Delphi представляет собой уникальную систему из числа языков и сред, используемых для разработки приложений. Во-первых, подобно Visual Basic, Delphi является средой для быстрой разработки приложений для 16- и 32-разрядных операционных систем Windows, а также, в случае Kylix, для Linux. Умело пользуясь мышью, компоненты можно швырять на форму также просто, как пшеницу на молодоженов. Еще немного щелчков мышью и чуть-чуть кодирования – и, пожалуйста! – компоненты связаны между собой сложным и непротиворечивым образом, снабжены обработчиками событий, и все вместе, образуют ни что иное, как завершенное бизнес-приложение.

Во-вторых, подобно C++, Delphi дает возможность близко подобраться к сердцу операционной системы через множество API-интерфейсов. В ряде случаев доступ к API-интерфейсам предоставляет компания Borland (Inprise) в рамках среды Delphi, в других ситуациях разработчики переносят заголовочные файлы на C в среду Delphi (в рамках проекта Jedi (Джедай) на Web-сайте www.delphi-jedi.org). Так или иначе, но Delphi благополучно делает эту работу и манипулирует функциями операционной системы по собственному усмотрению.

Программисты на Delphi условно делятся на два "лагеря" – программисты прикладных приложений и так называемые системные программисты. Иногда можно встретить и уникальных представителей, которые делают обе работы. Тем не менее, есть у представителей обеих "лагерей" одна общая черта – они должны хорошо разбираться в глубинной сути мира алгоритмов. Какой бы длинной или короткой была ваша программистская практика, рано или поздно, вы дойдете до момента, когда крайне необходимо самостоятельно закодировать, скажем, бинарный поиск. И, конечно же, перед тем как приступить к решению упомянутой проблемы, потребуется решить задачу, связанную с разработкой процедуры сортировки определенного вида данных, дабы бинарный поиск смог корректно функционировать. Иногда при помощи профилировщика удастся идентифицировать узкое место в **TStringList**, и, в конечном счете, понять, что более эффективное решение задачи может обеспечить совершенно другая структура данных.

По сути, алгоритмы представляют собой своего рода кровеносные сосуды той работы, которую мы называем программированием. Начинающие программисты очень часто остерегаются иметь дело с формальными алгоритмами. Полагаю, что поначалу пугает даже само слово, правда, лишь до тех пор, пока не состоится более близкое знакомство с алгоритмами. Запомните одну важную вещь: любая программа может трактоваться как некий алгоритм, который должен получить у пользователя данные, должным образом обработать их и выдать обратно предсказуемый результат.

Стандартные алгоритмы были разработаны и обкатаны учеными в области компьютерных наук с целью использования их "рабочими лошадками", коими являемся мы с вами. Профессиональное использование базовых алгоритмов – это то, что удерживает нас на плаву как профессионалов, придает уверенности и дает нам возможность заявлять о знании того или иного языка программирования. Например, если вам хорошо известно, что такое хеш-таблицы, их достоинства и недостатки, где и почему они применяются, когда какой реализации отдавать предпочтение, то вы сможете совершенно по-другому взглянуть на рабочий проект подсистемы или приложения и найти места, где возможно получить выгоду от использования хеш-таблиц. Если алгоритмы сортировки вызывают не панику, а лишь легкую улыбку, вы понимаете глубинные основы их функционирования и знаете, когда отдавать предпочтение сортировке простыми вставками, а когда – быстрой сортировке, возможно, вы безо всяких колебаний реализуете один из ал-

горитмов в рамках своего приложения, а не будете бесцельно терять время на эксперименты со стандартными компонентами Delphi. (Приведу лишь одну "жуткую" историю из современной программистской лирики. Некий программист использовал скрытый на форме компонент **TListBox**, добавлял в него набор строк, а затем устанавливал значение свойства **Sorted** равным **true**, тем самым, надеясь отсортировать эти строки.)

Полагаю, сейчас в ваших головах крутится одна мысль: "Понятно, писать книги по алгоритмам – это хорошо, но зачем при этом беспокоиться о каких-то там Delphi или Kylix?"

Кстати, давайте примем следующее соглашение, иначе мне придется ужасно много раз писать "Delphi или Kylix". Когда я говорю "Delphi или Kylix", в действительности я имею в виду либо Delphi, либо Kylix. В конце концов, Kylix получил известность, в основном, как система Delphi для Linux, находящаяся на этапе предварительного выпуска. Таким образом, в этой книге под "Delphi или Kylix" понимается либо Delphi для Windows, либо Kylix для Linux.

Итак, почему Delphi? На самом деле, на то имеются две причины: язык Object Pascal и операционная система. Язык, встроенный в среду Delphi, имеет множество конструкций, которые не доступны в других языках, конструкций, которые существенно упрощают инкапсуляцию эффективных алгоритмов и структур данных и делают ее более естественной. Примером могут послужить такие вещи, как свойства. Или, скажем, механизм исключений, генерируемых в случае возникновения непредвиденных ситуаций и ошибок. Несмотря на то что стандартные алгоритмы можно кодировать на Delphi и без применения таких специфических языковых конструкций, я довольно-таки твердо убежден, что в этом случае мы безвозвратно теряем и красоту, и эффективность реализаций, предпосылками которых является язык. Мы лишаем себя возможности исследовать все "закоулки" этого замечательного языка программирования. В этой книге мы собираемся повсеместно использовать всю мощь, присущую языку Object Pascal в среде Delphi. Я не думаю, что у программистов на Java будут возникать какие-то сложности с интерпретацией и переводом кода на свой язык. Однако раз уж я выбрал Delphi, то Delphi и буду придерживаться.

Следует принять во внимание еще одну вещь. Как традиционно предполагается, алгоритмы являются общими, по крайней мере, на одном и том же центральном процессоре и в среде одной и той же операционной системы. Конечно, алгоритмы можно оптимизировать под среду Windows или Linux. Можно добиться большей эффективности при их выполнении на семействе процессоров Pentium, в случае использования различных типов кэш-памяти или подсистем виртуальной памяти в средах разных операционных систем. Подобным возможностям оптимизации в книге уделяется отдельное внимание. Тем не менее, мы не будем доходить в своей погоне за эффективностью до кодирования на языке ассемблера, оптимизированного

под конвейерную архитектуру новых процессоров, – я должен был хоть где-нибудь это сказать!

В конечном счете, в книге, посвященной алгоритмам, нуждается само сообщество разработчиков на Delphi, причем в такой, которая бы отражала этот конкретный язык программирования, используемые операционные системы и процессоры. Ну, так вот она, книга. Она не суть переписанная книга, посвященная алгоритмам с реализацией на другом языке программирования. Напротив, книга написана с нуля автором, который на протяжении всей своей практики работал с Delphi ежедневно, зарабатывает себе на жизнь тем, что пишет библиотечное программное обеспечение и немало знает о сложностях, связанных с созданием коммерческих подпрограмм, классов и инструментальных средств.

Что я должен предварительно знать?

В этой книге отнюдь не предпринимается попытка обучить кого-либо программированию на Delphi. Необходимо знать основы разработки приложений на Delphi: создание новых проектов, написание кода, компиляцию, отладку и так далее. Я вынужден предупредить, что в книге не используются компоненты. Вы должны четко представлять, что такое классы, процедуры и методы, а также ссылки на них, владеть механизмом нетипизированных указателей, уметь использовать тип **TList** и потоки, инкапсулированные в семейство **TStream**. Очень важно владеть основами объектно-ориентированной методологии, в частности, представлять, что такое инкапсуляция, наследование, полиморфизм и делегирование. Вас не должна пугать объектная модель, реализованная в рамках Delphi!

Обладая упомянутым выше багажом знаний, большинство концепций, описанных в книге, покажутся просто детскими игрушками. Начинающие программисты почерпнут из книги неоценимые основы стандартной алгоритмической теории и структур данных, что позволит использовать им эту книгу как хороший учебник. В самом деле, даже простой просмотр кода, которым изобилует книга, даст возможность ознакомиться с множеством приемов и трюков, столь характерных для истинных профессионалов. Разбор более сложных моментов можно оставить на какой-нибудь скучный дождливый вечерок, если только они действительно не понадобятся в реальной работе.

Итак, на данный момент можно с уверенностью заявить, что вы должны обладать определенным опытом программирования на Delphi. То и дело придется сталкиваться со структурами данных, лежащими в основе **TList** и иже с ними, посему следует четко представлять себе, какие структуры данных доступны, и как их использовать. Может статься, что вам необходимо разработать простую подпрограмму сортировки, однако все, что содержит доступный вам источник – так это написанный кем-то код на языке C++, а ни времени, ни желания переводить этот код на Delphi нету. А, может, вас интересует книга по алгоритмам, в которой вопросы увеличения производительности и эффективности описываются столь же хорошо, как и сами алгоритмы? Такая книга перед вами.

Какая версия Delphi мне нужна?

Готовы ли вы к тому, что я сейчас скажу? Любая версия. За исключением раздела, посвященного использованию динамических массивов в Delphi 4 и тех же массивов в Kylix в главе 2, части материала в главе 12 и небольших фрагментов кода тут и там, приведенный в книге код будет компилироваться и выполняться под управлением любой версии Delphi. Не считая небольших порций кода, специфических для конкретной версии, о которой только что было упомянуто, я протестировал весь код, приведенный в книге, во всех версиях Delphi и Kylix.

Таким образом, вы смело можете полагать, что все примеры кода в книге функционируют во всех версиях Delphi. Если тот или иной фрагмент кода все-таки зависит от версии, это специальным образом оговаривается в комментариях.

Что и где я могу найти в книге, или, другими словами, из чего состоит эта книга?

Книга состоит из двенадцати глав и списка использованной литературы.

В *главе 1* вводятся несколько основных правил. Глава начинается с обсуждения проблемы производительности. Мы ознакомимся с вопросами измерения эффективности алгоритмов, начав с изучения O-нотации. Затем мы рассмотрим методику измерения времени выполнения алгоритмов и завершим исследованиями способов применения профилировщика. Мы обсудим эффективность представления данных в контексте современных процессоров и операционных систем, акцентируя особое внимание на кэш-памяти, механизмах подкачки и подсистемах виртуальной памяти. В конце главы приводятся рассуждения по поводу тестирования и отладки, которые можно встретить во множестве других книг, однако, по причине их чрезвычайной важности, непростительно было бы упустить эту тему из виду.

Глава 2 покрывает практически все основные вопросы, связанные с массивами. Мы посмотрим на стандартную языковую поддержку массивов, в том числе и динамических массивов, обсудим достоинства, недостатки и методику применения класса **TList**, а затем разработаем класс, инкапсулирующий в себе массив записей. Ввиду того, что строка, как структура данных, также представляет собой массив, мы кратко коснемся и ее.

В *главе 3* вводятся понятие связного списка в двух его ипостасях: односвязный и двухсвязный списки. Мы ознакомимся с тем, как создавать стеки и очереди с использованием для их внутреннего представления как связных списков, так и массивов.

Глава 4 представляет собой введение в алгоритмы поиска, в особенности, в алгоритмы последовательного и бинарного поиска. Будет показано, как при помощи бинарного поиска осуществлять вставку элементов в сортированные массивы и связные списки.

Глава 5 посвящена алгоритмам сортировки. Мы посмотрим на различные методы сортировки: пузырьковую и шейкер-сортировку, сортировку выбором и про-

стыми вставками, сортировку методом Шелла, быструю сортировку и сортировку слиянием. Алгоритмы сортировки будут применяться в отношении к массивам и связным спискам.

В *главе 6* обсуждаются алгоритмы, которые генерируют или требуют для своего функционирования случайные числа. Будут рассмотрены различные реализации генераторов псевдослучайных чисел, а также сортированной структуры данных с возможностью пометки, именуемой списком с пропусками, в которой для поддержания сбалансированного состояния используется генератор псевдослучайных чисел.

Глава 7 вводит понятия хеширования и хеш-таблиц, включая их базовые определения, области и причины применения, а также связанные с ними достоинства и недостатки. Рассматривается множество стандартных алгоритмов хеширования. Одной из проблем, которые возникают при использовании хеш-таблиц, является так называемый конфликт, или коллизия. Мы посмотрим, как разрешать коллизии при помощи разнообразных видов зондирования и связывания.

В *главе 8* представлены бинарные деревья, исключительно важная структура данных с широчайшим спектром случаев применения. Подробно рассматриваются вопросы построения и поддержки бинарных деревьев, а также методы прохода по узлам дерева. Затрагиваются вопросы несбалансированных деревьев, образующихся в результате вставки данных в сортированном порядке. В главе приводится набор алгоритмов балансировки, среди которых скошенное дерево и красно-черное дерево.

Глава 9, в основном, имеет дело с очередями по приоритету. Во время обсуждения таких очередей рассматривается структура сортирующего дерева. Подробно изучаются базовые операции на сортирующем дереве, такие как пузырьковый подъем и просачивание вниз. Кроме того, анализируется новый алгоритм сортировки на сортирующем дереве – пирамидальная сортировка.

В *главе 10* можно найти исчерпывающую информацию о конечных автоматах и об их применении для решения определенного класса задач. После рассмотрения некоторых стандартных примеров использования детерминированных конечных автоматов приводятся глубокие исследования регулярных выражений, а также алгоритмы их синтаксического анализа и компиляции в недетерминированные конечные автоматы. В конце главы приводятся примеры применения конечных автоматов для ввода или отклонения строк.

Глава 11 сконцентрирована вокруг нескольких технологий сжатия. Подробно рассматриваются такие алгоритмы сжатия, как Шеннона-Фано, Хаффмана, с применением скошенного дерева и LZ77.

В *главу 12* включено несколько дополнительных сложных тем, которые смогут удовлетворить аппетит даже самых искушенных программистов, склонных к исследованию алгоритмов и структур данных. Глава принесет несомненную пользу также и рядовым программистам.

В самом конце книги приводится список использованной литературы, который поможет быстро найти источники, содержащие дополнительную или более подробную информацию, касающуюся рассмотренных в книге алгоритмов. Список включает, помимо прочих, и чисто академические источники.

Что это за странные конструкции **\$IFDEF** в коде?

Все коды примеров, представленных в книге, за несколькими специальным образом помеченными исключениями, будут компилироваться в средах Delphi 1, 2, 3, 4, 5 и 6, а также Kylix 1. (Впрочем, должны поддерживаться и будущие версии компиляторов. Дополнительную информацию по этому поводу можно найти по адресу <http://www.boyet.com/dads>.) Несмотря на приложенные мною усилия, некоторые отличия в коде для различных версий Delphi и Kylix все же имеют место.

Дабы решить все вопросы, связанные с этими отличиями, я решил поместить в код множество конструкций **\$IFDEF**, которые обеспечивают условную компиляцию отдельных фрагментов кода. Компания Borland (Inprise) предлагает набор определений для официальных платформ **WINDOWS**, **WIN32** и **LINUX**, а также набор определений для версий компиляторов **VERnnn**.

Для решения упомянутых проблем каждый файл с исходным кодом, сопровождающий данную книгу, содержит в самом начале следующее включение:

```
{ $I TDDefine.inc }
```

В этом включаемом файле находятся читабельные определения компилятора для различных версий:

DelphiN	определение для конкретной версии Delphi, N = 1, 2, 3, 4, 5, 6
DelphiNPlus	определение для конкретной или более поздней версии Delphi, N = 1, 2, 3, 4, 5, 6
KylixN	определение для конкретной версии Kylix, N = 1
KylixNPlus	определение для конкретной или более поздней версии Kylix, N = 1
HasAssert	определение, поддерживает ли компилятор Assert

Кроме того, я предполагаю, что каждый компилятор, за исключением Delphi 1, поддерживает длинные строки.

Типографские соглашения

Основной текст книги, то есть обсуждения, описания, постановки задач, представлен этим шрифтом.

Коды всех листингов напечатаны моноширинным шрифтом.

Базовые понятия, термины и ключевые фразы выделены *курсивом*.

Во многих местах книги можно встретить примечания наподобие этого. Примечания служат для того, чтобы подчеркнуть нечто очень важное, а также предупредить о тех или иных особенностях или предостеречь о подводных камнях.

Благодарности

Любую книгу готовит далеко не один человек, и, естественно, эта – не исключение. Есть очень много людей, без усилий которых эта книга так и не увидела бы свет. Я хотел бы представить этих людей в, так сказать, исторической последовательности, последовательности, в соответствие с которой они оказывали на меня свое благотворное влияние.

С первыми двумя джентльменами, к сожалению, я никогда не разговаривал лично и не встречался, однако это те люди, которые до сих пор держат мои глаза широко открытыми и постоянно питают мой энтузиазм в безбрежном мире алгоритмов. Если бы не они, кто знает, где бы я был сейчас и чем бы занимался. Я имею в виду Дональда Кнута (Donald Knuth) (www.cs.stanford.edu/~knuth/) и Роберта Седжвика (Robert Sedgewick) (www.cs.princeton.edu/~rs/). В действительности, книга Седжвика, посвященная алгоритмам [20], как раз и сподвигла меня на начало моей творческой деятельности. Это была первая книга подобного рода, которую я даже решился приобрести, причем в то время я только-только вникал в программирование на Turbo Pascal. Дональд Кнут вообще не требует какого-либо представления. Его поистине великолепные труды [11, 12, 13] по-прежнему находятся на верхушке всего дерева алгоритмов; впервые я воспользовался этой книгой во время учебы в Королевском колледже Лондонского университета при получении степени бакалавра по математике.

Несколько лет спустя, следующим человеком, кому я хотел бы выразить свою благодарность, стал Ким Кокконен (Kim Kokkonen). Он предоставил мне работу в компании TurboPower Software (www.turbopower.com) и дал мне возможность настолько досконально изучить компьютерные науки, насколько я даже не мог и мечтать. Хочу поблагодарить также и всех сотрудников компании TurboPower Software, а также тех ее клиентов, которых я знал на протяжении многих лет. Благодарю Роберта Делросси (Rober DelRossi), президента TurboPower Software, за поощрение всех моих начинаний.

Следующей идет маленькая компания, к сожалению, ныне не работающая, которая называлась Natural Systems. В 1993 году эта компания выпустила продукт под названием Data Structures for Turbo Pascal (Структуры данных для Turbo Pascal). Я купил этот продукт, и, по моему мнению, это было не особенно удачное приобретение. О да, продукт работал неплохо, однако я не был согласен ни с его дизайном, ни с реализацией, кроме того, скорость функционирования оставляла желать лучшего. Именно этот продукт побудил меня написать собственную бесплатную библиотеку EZSTRUCS для Turbo Pascal 7, от которой, собственно и пошла моя хорошо известная бесплатная библиотека структур данных для Delphi. Эти усилия были первым проблеском моего действительного понимания структур данных, поскольку зачастую понимание полностью съедается необходимостью быстрой сдачи работы.

Хочу выразить благодарность Крису Фризеллу (Chris Frizell), редактору и владельцу журнала *The Delphi Magazine* (www.thedelphimagazine.com). Он как в воду глядел, предоставив мне возможность обсудить множество алгоритмов на страницах его поистине бесценного журнала, в конечном итоге отдав мне отдельную ежемесячную колонку *Algorithms Affresco*. Без его содействия и поддержки эта книга могла бы и не появиться, во всяком случае, даже если бы она и появилась, она бы уж точно была намного хуже. Я настоятельно рекомендую всем разработчикам подписаться на журнал *The Delphi Magazine*, поскольку, по моему мнению, он был, есть и будет наиболее глубоким, фундаментальным и серьезным периодическим изданием для широчайшего круга программистов на Delphi. Спасибо всем читателям моей колонки за их суждения и комментарии.

Под занавес, я хотел бы поблагодарить всех сотрудников издательства Wordware (www.wordware.com), в том числе моих редакторов, издателя Джима Хилла (Jim Hill) и выпускающего редактора Вес Беквис (Wes Beckwith). Джим поначалу был несколько обескуражен моим предложением издать книгу, посвященную алгоритмам, однако он очень быстро проникся моими идеями и оказывал всемерную поддержку на протяжении всего времени подготовки книги. Кроме того, я хочу выразить самые теплые благодарности научным редакторам: Стиву Тексейра (Steve Teixeira), одному из авторов *Delphi X Developer's Guide*, и моему другу Энтону Паррису (Anton Parris).

И, в заключение, хочу выразить благодарность и еще раз признаться в любви своей жене, Донне (вот кто был основной движущей силой, постоянно побуждающей меня к работе над книгой). Без ее любви, энтузиазма и поощрения я не смог бы прожить все эти годы. Спасибо тебе, любящее сердце. Ты знаешь, что рядом с тобой бьется второе такое же!

Джулиан М. Бакнелл

Колорадо Спрингс,

апрель, 1999 года – февраль 2001 года



Что такое алгоритм?

Для книги, посвященной алгоритмам, очень важно заранее оговорить, о чем, собственно, пойдет речь. Как мы увидим, одной из самых важных причин для понимания и исследования алгоритмов является ускорение работы приложений. Да, конечно, иногда нужны алгоритмы, для которых более важную роль играет занимаемое ими пространство памяти, а не их быстродействие, но все же в большинстве случаев решающим фактором является именно быстродействие.

Несмотря на то что темой этой книги будут алгоритмы, структуры данных и их реализация в коде, мы рассмотрим и некоторые чисто процедурные моменты: как написать код, который позволит упростить отладку в случае возникновения проблем, как выполнять тестирование кода и как убедиться в том, что изменения, вносимые в одном месте, не вызовут ошибок в другом месте.

Что такое алгоритм?

Может показаться странным, но алгоритмы используются при написании любой программы, просто мы не считаем их алгоритмами: "Это вовсе не алгоритм, а просто порядок вычислений".

Алгоритм (algorithm) представляет собой пошаговую инструкцию выполнения вычислений или процесса. Это достаточно вольное определение, но как только читатель поймет, что ему нечего бояться алгоритмов, он легко научится их идентифицировать.

Вернемся к дням, когда мы учились в начальных классах и рассматривали простое сложение в столбик.

Учитель писал на доске пример сложения:

$$\begin{array}{r} 45 \\ 17+ \\ \hline \end{array}$$

а затем просил кого-нибудь из учеников вычислить сумму. Каждый про себя думал, как это сделать: начинаем со столбца единиц, складываем 5 и 7, получаем 12, пишем 2 в столбце единиц, а затем 1 переносим над 4.

$$\begin{array}{r} 1 \\ 45 \\ 17+ \\ \hline 2 \end{array}$$

Затем складываем перенесенную единицу с 4 и еще одну 1, в результате получаем 6, которое и пишем под столбцом десятков. Вот и получился ответ 62.

Обратите внимание, что описанный выше ход мыслей представлял собой *алгоритм* сложения. Учитель не говорил, как складывать числа 45 и 17, он просто объяснил общий принцип сложения двух чисел. Вскоре любой ученик, применяя тот же алгоритм, мог складывать одновременно несколько чисел, содержащих много цифр. Конечно, в те дни вы не знали, что это был алгоритм, вы просто складывали числа.

В мире программирования мы представляем себе алгоритмы как сложные методы выполнения определенных вычислений. Например, если имеется массив записей покупателей, в котором необходимо найти определенного покупателя (скажем, Джона Смита (John Smith)), то можно действовать следующим образом: считывать каждый элемент массива, пока не будет найдена нужная запись или не будет достигнут конец массива. Для вас это может показаться очевидным методом решения поставленной задачи, тем не менее, в мире алгоритмов он известен как *последовательный поиск* (*sequential search*).

Существуют и другие методы поиска элемента "John Smith" в нашем гипотетическом массиве. Например, если элементы в массиве отсортированы по фамилии, то можно воспользоваться алгоритмом *бинарного поиска* (*binary search*). Согласно ему, мы берем средний элемент массива. Это "John Smith"? Если да, то поиск закончен. Если элемент меньше чем "John Smith" (под "меньше" здесь понимается, что он стоит "раньше" в алфавитном порядке), то можно сказать, что искомый элемент находится во второй половине массива, если же он больше, то нужный нам элемент находится в первой половине массива. Далее операции повторяются (т.е. мы снова берем средний элемент выбранной части массива, сравниваем его с элементом "John Smith" и выбираем ту часть, в которой этот элемент должен находиться) до тех пор, пока элемент не будет найден, или пока левая часть массива после очередного разбиения не окажется пустой.

Такой алгоритм кажется более сложным, чем первый рассмотренный алгоритм. Последовательный поиск можно очень просто и удобно организовать с помощью

цикла **For**, вызвав в нужный момент оператор **Break**. Бинарный поиск требует выполнения более сложной операции с локальными переменными. Таким образом, может показаться, что последовательный поиск быстрее только потому, что его реализация проще.

Что ж, добро пожаловать в мир анализа алгоритмов, в котором мы постоянно проводим эксперименты и пытаемся сформулировать законы работы различных алгоритмов!

Анализ алгоритмов

Рассмотрим два возможных варианта поиска в массиве элемента "John Smith": последовательный поиск и бинарный поиск. Мы напишем код для обоих вариантов, а затем определим производительность каждого из них. Реализация простого алгоритма последовательного поиска приведена в листинге 1.1.

Листинг 1.1. Последовательный поиск имени в массиве элементов

```
function SeqSearch( aStrs : PStringArray; aCount : integer;
                   const aName : string5) : integer;

var
  i : integer;
begin
  for i := 0 to pred(aCount) do
    if CompareText(aStrs[i], aName) = 0 then begin
      Result := i;
      Exit;
    end;
  Result := -1;
end;
```

В листинге 1.2 содержится код более сложного бинарного поиска. (Пока что мы не будем объяснять, что происходит в этом коде. Алгоритм бинарного поиска подробно рассматривается в главе 4.)

Очень трудно оценить быстродействие каждого из приведенных кодов только по самому их виду. Это основной принцип, которому мы должны всегда следовать: нельзя оценивать скорость работы кода по его виду. *Единственным* методом определения быстродействия должно быть его выполнение. И только. Если есть возможность выбирать между несколькими алгоритмами, как в рассматриваемом случае, то для выбора более эффективного алгоритма с нашей точки зрения нужно *оценить время выполнения* кода в различных условиях и на различных исходных данных.

Традиционно для оценки времени работы кода используется *профилировщик* (*profiler*). Профилировщик загружает тестируемое приложение и точно измеряет время выполнения отдельных подпрограмм. Профилировщик рекомендуется использовать во всех случаях. Только профилировщик поможет определить, на что тратится большая часть времени выполнения кода, а, следовательно, над какими

подпрограммами стоит поработать с целью увеличения быстродействия всего приложения.

Листинг 1.2. Бинарный поиск имени в массиве элементов

```
function BinarySearch( aStrs : PStringArray; aCount : integer;
                      const aName : string5) : integer;
var
  L, R, M : integer;
  CompareResult : integer;
begin
  L := 0;
  R := pred(aCount);
  while (L <= R) do begin
    M := (L + R) div 2;
    CompareResult := CompareText(aStrs^ [M], aName);
    if (CompareResult = 0) then begin
      Result := M;
      Exit;
    end
    else if (CompareResult < 0) then
      L := M + 1
    else
      R := M - 1;
    end;
  Result := -1;
end;
```

В компании TurboPower Software, где работает автор книги, используется профессиональный профилировщик из пакета Sleuth QA Suite. Все коды, приведенные в книге, были протестированы как с помощью Stopwatch (название профилировщика из пакета Sleuth QA Suite), так и с помощью CodeWatch (название отладчика использования ресурсов и утечки памяти из пакета Sleuth QA Suite). Тем не менее, даже если у вас нет своего профилировщика, вы можете проводить тестирование и определять время выполнения. Просто это не совсем удобно, поскольку в код приходится помещать вызовы функций работы со временем. Нормальные профилировщики не требуют внесения в код изменений, они оценивают время за счет изменения выполняемого файла в памяти компьютера непосредственно в процессе выполнения.

Для тестирования и определения времени выполнения алгоритмов поиска была написана специальная программа. Фактически она определяет системное время вначале перед, а затем и после выполнения кода. По результатам определения времени вычисляется время выполнения. Принимая во внимание, что в настоящее время компьютеры стали достаточно мощными, а часы системного времени характеризуются сравнительно низкой точностью, как правило, для более точной оценки быстродействия код выполняется несколько сот раз, а затем определяется сред-

нее значение. (Кстати, эта программа была написана в среде 32-разрядной Delphi и не будет компилироваться под Delphi 1, поскольку она выделяет память для массивов из кучи, которая превышает граничное для Delphi 1 значение 64 Кб.)

Эксперименты по оценке быстродействия алгоритмов проводились различными способами. Сначала для обоих алгоритмов было определено время, необходимое для поиска фамилии "Smith" в массивах из 100, 1000, 10000 и 100000 элементов, которые содержали искомый элемент. В следующей серии экспериментов осуществлялся поиск того же элемента в массивах того же размера, но при отсутствии в них искомого элемента. Результаты экспериментов приведены в таблице 1.1.

Таблица 1.1. Времена выполнения последовательного и бинарного поиска

	Элемент найден	Элемент не найден
Последовательный поиск		
100	0.14	0.10
1 000	1.44	1.05
10 000	15.28	10.84
100 000	149.42	106.35
Бинарный поиск		
100	0.01	0.01
1 000	0.01	0.01
10 000	0.02	0.02
100 000	0.03	0.02

Как видно из таблицы, эксперименты показали очень интересные результаты. Время выполнения последовательного поиска пропорционально количеству элементов в массиве. Таким образом, можно сказать, что характеристики выполнения последовательного поиска линейны.

Результаты выполнения бинарного поиска проанализировать сложнее. Может даже показаться, что из-за очень быстрого выполнения алгоритма при определении времени мы столкнулись с проблемой потери точности. Очевидно, что зависимость между количеством элементов в массиве и временем выполнения алгоритма не является линейной. Но по приведенным данным трудно определить тип зависимости.

Эксперименты были проведены повторно. При этом времена выполнения умножались на коэффициент 100.

Таблица 1.2. Повторное тестирование бинарного поиска

	Элемент найден	Элемент не найден
100	0.14	0.10
1 000	1.44	1.05
10 000	15.28	10.84
100 000	149.42	106.35

Эти данные более достоверны. Из них видно, что десятикратное увеличение количества элементов в массиве приводит к увеличению времени выполнения на определенную постоянную величину (примерно на 0.5). Это логарифмическая зависимость, т.е. время бинарного поиска пропорционально логарифму количества элементов в массиве.

(Если вы не математик, то вам будет не так легко это понять. Вспомните из своих школьных дней, что для вычисления произведения двух чисел можно вычислить их логарифмы, сложить их, а затем определить антилогарифм суммы. Поскольку в рассматриваемых экспериментах количество элементов умножается на 10, то в логарифмической зависимости это будет эквивалентно прибавлению константы. Как раз это мы и видим в результатах экспериментов: для каждого последующего массива время увеличивается на 0.5.)

Что мы узнали из результатов проведенных экспериментов? Во-первых, теперь мы знаем, что единственным методом определения быстродействия алгоритма является оценка времени его выполнения.

В общем случае, единственным методом определения быстродействия отдельной части кода является оценка времени ее выполнения. Это справедливо как в отношении широко известных алгоритмов, так и в отношении алгоритмов, разработанных лично вами. Не нужно предполагать, просто измерьте время выполнения.

Во-вторых, мы определили, что по своей природе последовательный поиск является линейным, а бинарный поиск – логарифмическим. Если быть поближе к математике, то можно взять эти статистические результаты и теоретически доказать их справедливость. Тем не менее, в этой книге мы не будем перегружать текст математическими выкладками. Можно найти немало книг, в которых приведены эти выкладки (см., например, тома "Фундаментальные алгоритмы на C++" и "Фундаментальные алгоритмы на C" Роберта Седжвика, вышедшие в свет в издательстве "Диасофт").

О-нотация

Для выражения характеристик быстродействия удобно иметь более компактное определение, нежели "быстродействие алгоритма X пропорционально количеству элементов в третьей степени" или что-нибудь в этом роде. В вычислительной технике уже есть короткая и более удобная схема – *О-нотация* (*big-Oh notation*).

В этой нотации используется специальная математическая функция от n , т.е. количества элементов, которой пропорционально быстродействие алгоритма. Таким образом, мы говорим, что алгоритм принадлежит к классу $O(f(n))$, где $f(n)$ – некоторая функция от n . Приведенное обозначение читается как "О большое от $f(n)$ " или, менее строго, "пропорционально $f(n)$ ".

Например, наши эксперименты показали, что последовательный поиск принадлежит к классу $O(n)$, а бинарный – к классу $O(\log(n))$. Поскольку для положитель-

ных чисел $n \log(n) < n$, можно сделать вывод о том, что бинарный поиск всегда быстрее, чем последовательный. Тем не менее, немного ниже будут приведены несколько замечаний, касающихся выводов, сделанных из О-нотации.

О-нотация проста и удобна. Предположим, что экспериментальным путем было определено, что алгоритм X принадлежит к классу $O(n^2 + n)$. Другими словами, его быстродействие пропорционально $n^2 + n$. Под словом "пропорционально" понимается, что можно найти такую константу k , для которой

$$\text{Быстродействие} = k * (n^2 + n)$$

Из приведенного уравнения видно, что умножение математической функции внутри скобок в О-нотации на константу не оказывает никакого влияния на смысл нотации. Так, например, $O(3*f(n))$ эквивалентно $O(f(n))$, поскольку 3 можно без последствий вынести как коэффициент пропорциональности, который мы игнорируем.

Если величина n при тестировании алгоритма X достаточно велика, можно сказать, что влияние члена "+n" поглощается членом " n^2 ". Другими словами, при больших значениях n алгоритм $O(n^2 + n)$ эквивалентен алгоритму $O(n^2)$. То же можно сказать и для n более высоких степеней. Так, для достаточно больших n влияние члена n^2 будет поглощено влиянием члена n^3 . В свою очередь, влияние члена $\log(n)$ будет поглощаться влиянием члена n и т.д.

Из приведенного примера видно, что О-нотация подчиняется очень простым арифметическим правилам. Давайте предположим, что есть алгоритм, который выполняет несколько различных задач. Первая задача сама по себе принадлежит к классу $O(n)$, вторая – к классу $O(n^2)$, а третья – к классу $O(\log(n))$. Необходимо определить быстродействие алгоритма в целом. Ответом будет $O(n^2)$, поскольку к этому классу принадлежит доминантная часть алгоритма.

В этом и заключается первое замечание, касающееся выводов, следующих из О-нотации. Значения О большого являются репрезентативными для *больших* значений n . Для *маленьких* значений О-нотация не имеет смысла, а на общий результат оказывают влияние другие члены нотации. Например, предположим, что проводилось тестирование двух алгоритмов. На основе статистических данных были выведены следующие зависимости:

$$\text{Быстродействие первого алгоритма} = k1 * (n + 100000)$$

$$\text{Быстродействие второго алгоритма} = k2 * n^2$$

Пусть константы $k1$ и $k2$ сравнимы по величине. Какой алгоритм лучше использовать? Если следовать О-нотации, то предпочтительнее будет первый алгоритм, поскольку он принадлежит к классу $O(n)$. Тем не менее, если известно, что значение n в реальных условиях не будет превышать 100, более эффективным окажется второй алгоритм.

Таким образом, алгоритм нужно выбирать и с учетом его назначения – не только на основании *O*-нотации, но принимая во внимание время выполнения при средних значениях количества элементов (или, если угодно, условий использования), на которых алгоритм будет применяться. Следовательно, выбор алгоритма должен осуществляться только на основе измерения профилировщиком времени выполнения *вашего* приложения для *ваших* данных. Не полагайтесь ни на какие книги (в том числе и на эту), верьте только измеренному времени.

Лучший, средний и худший случаи

Помимо всего прочего, необходимо рассмотреть еще один вопрос. *O*-нотация относится к *среднему случаю*. Вернемся к нашим экспериментам, связанным с поиском элемента в массиве. Если бы фамилия "Smith" всегда была первым элементом в массиве, последовательный поиск был бы быстрее бинарного, – искомый элемент был бы обнаружен при первом же выполнении цикла. Такая ситуация известна под названием *лучший случай*. Для нашего примера в *O*-нотации ее можно представить как $O(1)$ (т.е. выполнение алгоритма занимает одно и то же время независимо от количества элементов в массиве).

Если бы фамилия "Smith" всегда была последним элементом в массиве, последовательный поиск был бы очень медленным. Такая ситуация известна под названием *худший случай*. В нашем примере ее можно представить как $O(n)$, точно так же, как и для среднего случая.

Несмотря на то что для бинарного поиска быстроедействие в лучшем случае (искомый элемент всегда находится в середине массива) равно быстроедействию в лучшем случае для последовательного поиска, тем не менее, его быстроедействие в худшем случае намного выше. Собранные нами статистические данные при поиске элемента, которого нет в массиве, являются значениями для худшего случая.

В общем, при выборе алгоритма следует учитывать значения в *O*-нотации для среднего и худшего случаев. Лучшие случаи, как правило, не интересны, поскольку программисты всегда более обеспокоены "граничными" условиями, по которым и будут судить о быстроедействии приложения.

Таким образом, мы увидели, что *O*-нотация – очень ценное средство оценки быстроедействия различных алгоритмов. Кроме того, следует помнить, что *O*-нотация в общем случае имеет смысл только для больших n . Для небольших n выбор алгоритма лучше осуществлять на основе статистических данных о времени его выполнения. Единственным достоверным методом оценки эффективности алгоритма является определение времени его работы. Поэтому не гадайте, а интенсивно используйте профилировщик.

Алгоритмы и платформы

В обсуждении быстроедействия алгоритмов мы до сих пор не затрагивали вопросов, касающихся операционной системы и оборудования компьютера, на котором выполняется реализация алгоритма. *O*-нотация справедлива только для

какой-то виртуальной вычислительной машины, в которой, например, нет никаких узких мест в операционной системе или оборудовании. К сожалению, мы живем и работаем в реальном мире, и наши приложения и алгоритмы будут выполняться на реальных физических компьютерах. Поэтому при анализе алгоритмов следует учитывать и данный фактор.

Виртуальная память и страничная организация памяти

Первым узким местом быстрогодействия приложения является страничная организация виртуальной памяти. Его легче понять на примере 32-разрядных приложений. 16-разрядные приложения тоже страдают от тех же проблем, но сама механика их возникновения разная. Обратите внимание, что в этом разделе мы будем говорить языком непрофессионалов, – целью раздела является обсуждение концептуальной информации, достаточной для понимания принципов происходящего, а не детальное рассмотрение системы страничной памяти.

При запуске приложения под управлением современной 32-разрядной операционной системы ему для кода и данных предоставляется блок виртуальной памяти, размером 4 Гб. Очевидно, что операционная система не дает *физически* эти 4 Гб из оперативной памяти (ОЗУ); понятно, что далеко не каждый может себе позволить выделить лишние 4 Гб ОЗУ под каждое приложение. Фактически предоставляется пространство логических адресов, по которым, теоретически, может храниться до 4 Гб данных. Это и есть виртуальная память. На самом деле ее нет, но если мы все делаем правильно, операционная система может предоставить нам физические участки памяти, если возникнет такая необходимость.

Виртуальная память разбита на *страницы*. В системах Win32 с процессорами Pentium размер одной страницы составляет 4 Кб. Следовательно, Win32 разбивает блок памяти объемом 4 Гб на страницы по 4 Кб. При этом в каждой странице содержится небольшой объем служебной информации о самой странице. (Память в операционной системе Linux работает примерно таким же образом.) Здесь содержатся данные о том, *занята* страница или нет. Занятая страница – это страница, в которой приложение хранит данные, будь то код или реальные данные. Если страница не занята, ее нет вообще. Любая попытка сослаться на нее вызовет ошибку доступа.

Далее, в служебную информацию входит ссылка на таблицу перевода страниц. В типовой системе с 256 Мб памяти (через несколько лет эта фраза, наверное, будет вызывать смех) доступно только 65536 физических страниц. Таблица трансляции страниц связывает отдельную виртуальную страницу памяти приложения с реальной страницей, доступной в ОЗУ. Таким образом, при попытке доступа приложения к определенному адресу операционная система выполняет трансляцию виртуального адреса в физический адрес ОЗУ.

Если в системе Win32 запущено несколько приложений, неизбежно будут возникать моменты, когда все физические страницы ОЗУ заняты, а одному из приложений требуется занять новую страницу. Но это невозможно, поскольку свободных

страниц нет. В таком случае операционная система записывает физическую страницу на жесткий диск (этот процесс называется *подкачкой* или *свопингом* (*swapping*)) и отмечает в таблице трансляции, что страница была записана на диск, после чего физическая страница помечается как занятая приложением.

Все это хорошо до тех пор, пока приложение, которому принадлежит страница на диске, не пытается обратиться к ней. Процессор определяет, что физическая страница уже недоступна и возникает *ошибка отсутствия страницы* (*page fault*). Операционная система принимает управление на себя, записывает другую страницу на диск, освобождает физическую страницу, записывает на освободившееся место запрашиваемую страницу и продолжает выполнение приложения. Само приложение ничего не знает о происходящем внутри операционной системы процессе. Оно, например, считывает первый байт страницы памяти, и именно это (в конечном счете) происходит.

Все описанное выше в 32-разрядной операционной системе происходит постоянно. Физические страницы записываются на диск и считываются с диска. При этом изменяются таблицы трансляции страниц. В большинстве случаев простой пользователь ничего не замечает, за исключением одной ситуации. И эта ситуация называется *пробуксовка* (*thrashing*).

Пробуксовка

Пробуксовка может негативно сказаться на вашем приложении, превращая его из высокоэффективной оптимизированной программы в медленную и ленивую. Предположим, что существует приложение, которое требует большого объема памяти, скажем, половину всей имеющейся в компьютере физической памяти. Оно создает большие массивы крупных блоков, выделяя память из кучи. Такое выделение приведет к тому, что будут занимать новые страницы, а старые, скорее всего, будут записываться на диск. Затем приложение считывает эти большие блоки, начиная с начала массива и в направлении его конца. Операционная система при необходимости будет считывать запрашиваемые страницы из ОЗУ. При этом никаких проблем возникать не будет.

А теперь представим себе, что приложение считывает блоки в произвольном порядке. Скажем, сначала оно считывает данные из блока 56, затем из блоков 123, 12, 234 и т.д. В таком случае вероятность возникновения ошибки отсутствия страницы увеличивается. При этом все большее и большее количество страниц будет записываться на диск и считываться с диска. Индикатор работы диска будет гореть почти постоянно, а скорость работы приложения упадет. Это и есть пробуксовка – непрерывный обмен страницами между диском и памятью, вызванный запросами приложения страниц в произвольном порядке.

В общем случае лекарства от пробуксовки нет. Большую часть времени блоки памяти выделяются из программы динамического распределения памяти Delphi. Кроме того, программист не может управлять конкретным расположением блоков памяти. Может случиться, например, что связанные блоки данных хранятся в раз-

ных страницах. (Здесь под словом "связанные" понимается блоки памяти, данные из которых, вероятно, будут считываться одновременно, поскольку сами данные связаны.) Одним из методов снижения риска возникновения пробуксовки является использование отдельных куч для выделения памяти для структур и данных разных приложений. Но алгоритм такого выделения в настоящей книге не приводится.

Рассмотрим пример. Предположим, что мы выделили память под элементы объекта **TList**. Каждый из элементов содержит, по крайней мере, одну строку, память для которой выделяется из кучи (например, мы пользуемся 32-разрядным Delphi и элемент использует длинные строки). А теперь представим себе, что приложение уже проработало некоторое время, и элементы в объекте **TList** неоднократно добавлялись и удалялись. Вполне возможно, что экземпляр **TList**, его элементы и строки элементов распределены по разным страницам памяти. Теперь при последовательном считывании элементов объекта **TList** от начала до конца приложение будет обращаться ко многим страницам, что приведет к активному обмену страницами между диском и памятью. Если количество элементов достаточно мало, все страницы, относящиеся к данному приложению, могут находиться в памяти. Но если в объекте **TList** элементов насчитывается несколько миллионов, при их считывании приложение может породить состояние пробуксовки.

Локальность ссылок

Самое время обсудить еще одну концепцию – *локальность ссылок*. Этот принцип представляет собой метод представления приложений, который помогает свести вероятность возникновения пробуксовки к минимуму. Это понятие предполагает, что связанные данные должны находиться в виртуальной памяти как можно ближе друг к другу. Если принцип локальности ссылок соблюдается, при считывании части данных другую их часть можно будет найти на соседних страницах памяти.

Например, массив записей имеет высокий уровень локальности ссылок. Так, элемент с индексом 1 в памяти находится рядом с элементом с индексом 2 и т.д. Если приложение последовательно считывает все записи массива, локальность ссылок будет очень высокой. Обмен страницами между диском и памятью будет минимальным. Экземпляр объекта **TList**, содержащий указатели на тот же тип записей, несмотря на то, что это тоже массив, фактически содержащий те же данные, будет иметь низкий уровень локальности ссылок. Как было показано ранее, каждый элемент такого массива может находиться на отдельной странице. Таким образом, последовательное считывание элементов вызовет обмен данными между диском и памятью. Связанные списки (см. главу 3) также обладают низким уровнем локальности ссылок.

Существуют специальные методы повышения уровня локальности ссылок для различных структур данных и алгоритмов, и некоторые из них будут рассмотрены в настоящей книге. К нашему сожалению, диспетчер динамического распределения памяти Delphi является слишком общим. Программист не может вынудить

Delphi выделить память под серию элементов из одной страницы. Еще хуже тот факт, что все объекты представляют собой экземпляры, память для которых выделяется из кучи. Возможность выделения памяти для отдельных объектов из определенных страниц позволила бы избежать многих неприятностей. (В действительности это возможно за счет подмены метода класса **NewInstance**, но подмену приходится делать для всех классов, для которых нужна такая возможность.)

До сих пор мы говорили о локальности ссылок в смысле расстояния ("один объект находится в памяти рядом с другим объектом"), но локальность ссылок можно трактовать и по отношению ко времени. Это означает, что если элемент недавно использовался, он скоро будет использоваться снова, или, скажем, элемент X всегда используется вместе с элементом Y. Воплощением локальности ссылок во времени является кэш-память. *Кэш-память (cache)* представляет собой небольшой блок памяти для некоторого процесса, содержащий элементы, которые использовались недавно. При каждом использовании элемента он копируется в кэш-память. Если кэш заполнен, при удалении элементов применяется алгоритм с *удалением наиболее давно использованных элементов* (least recently used, LRU), по которому элемент, который давно не использовался, замещается недавно использованным элементом. Таким образом, кэш-память содержит несколько близких в пространственном смысле элементов, которые, помимо всего прочего, близки и в смысле времени их использования.

Обычно кэш-память применяется для элементов, которые хранятся на медленных устройствах. В качестве классического примера можно привести дисковый кэш. Тем не менее, теоретически кэш виртуальной памяти мог бы работать точно таким же образом, особенно с приложениями, которые требуют большого объема памяти и используются на вычислительных машинах с небольшими объемами ОЗУ.

Кэш процессора

Оборудование, на котором мы все программируем и запускаем приложения, использует кэш в памяти. Так, например, на компьютере автора этой книги применяется высокоскоростная кэш-память объемом 512 Кб между процессором и его регистрами и основной памятью (объем которой на том же компьютере составляет 192 Мб). Эта высокоскоростная кэш-память представляет собой буфер: когда процессору необходимо считать из памяти определенные данные, кэш проверяет, есть ли эти данные в памяти, и если требуемых данных нет, считывает их. Таким образом, данные, доступ к которым осуществляется часто (обладающие высоким уровнем временной локальности ссылок) будут большую часть времени находиться в кэш-памяти.

Выравнивание данных

Еще один вопрос, касающийся оборудования, о котором следует помнить, связан с выравниванием данных. Современные процессоры устроены таким образом,

что они считывают данные отдельными кусками по 32 бита. Кроме того, эти куски всегда *выравниваются* по границе 32 бит. Это означает, что адреса памяти, передаваемые от процессора в кэш-память, всегда делятся на четыре без остатка (4 байта = 32 бита), т.е. два младших бита адреса являются нулевыми. Когда 64- и более разрядные процессоры станут достаточно распространенными, адресация превратится в 64-битную (или 128-битную) и выравнивание будет производиться уже по новой границе.

Какое отношение имеет выравнивание данных к приложениям? При программировании необходимо убедиться, что переменные типа **longint** и указатели выровнены по четырехбайтовой или 32-битовой границе. Если они переходят через границу 4 байт, процессору придется выдать *две* команды на считывание кэш-памяти: первая команда для считывания первой части, а вторая – второй части. Затем процессору потребуется соединить две части значения и отбросить ненужные биты. (В ряде процессоров 32-битные значения всегда должны выравниваться по границе 32 бит. В противном случае возникает ошибка нарушения доступа. К счастью, процессоры Intel не требуют этого, что, в свою очередь, провоцирует программистов на некоторую небрежность.)

Всегда убеждайтесь в том, что 32-битные значения выровнены по границе 32 бит, а 16-битные значения – по границе 16 бит. Для увеличения быстродействия следует убедиться, что 64-битные значения (например, переменные типа **double**) выровнены по 64-битной границе.

Все это звучит достаточно сложно, но в действительности программисту очень помогает компилятор Delphi. В результате особое внимание нужно уделять только объявлению типа **record**. Все глобальные и локальные атомарные переменные (т.е. переменные простых типов) выравниваются должным образом. Если тип выравнивания не установлен, то 32-разрядный компилятор Delphi будет автоматически выравнивать и поля типа **record**. Для этого он добавляет незначащие байты. В 16-разрядной версии автоматическое выравнивание переменных атомарных типов не используется, поэтому будьте осторожны.

Автоматическое выравнивание переменных иногда может ввести программиста в заблуждение. Если, например, объявлен следующий тип **record** в 32-разрядной версии Delphi, каким будет результат выполнения операции **sizeof(TMyRecord)**?

```
type
  TMyRecord = record
    aByte : byte;
    aLong : longint;
  end;
```

Многие без сомнения ответят, что, дескать, 5 байт (и это было бы правильно для Delphi 1). Однако верным ответом будет 8 байт. Компилятор автоматически вставит три байта между полем **aByte** и **aLong**, просто чтобы выровнять последнее поле по границе 4 байт.

Если тип **record** объявить следующим образом:

```
type
  TMyRecord = packed record
    aByte : byte;
    aLong : longint;
  end;
```

то функция **sizeof(TMyRecord)** даст результат 5. Однако в этом случае доступ к полю **aLong** потребует больше времени, чем в предыдущем примере, поскольку значение поля будет переходить через границу 4 байт. Следовательно, правило можно сформулировать так: если используется ключевое слово **packed**, поля в записи должны располагаться таким образом, чтобы данные были выровнены по границе 4 байт. Сначала объявите 4-байтные поля, а затем уже все остальные. Это правило применяется во всех кодах, приведенных в настоящей книге. И еще одно, никогда не угадывайте размер записи, а пользуйтесь для этого функцией **sizeof**.

Кстати, следует знать, что диспетчер динамического распределения памяти Delphi также помогает выполнять выравнивание данных. Он выравнивает не только 4-байтные значения по границе 4 байт, но и 8-байтные значения по границе 8 байт. Это имеет большое значение для переменных типа **double**: операции с числами с плавающей запятой выполняются быстрее, если переменные типа **double** выровнены по границе 8 байт. Если задачи по программированию связаны с использованием большого количества числовых переменных, убедитесь, что поля типа **double** в записях выровнены по границе 8 байт.

Пространство или время

Чем больше мы изучаем, разрабатываем и анализируем алгоритмы, тем чаще мы сталкиваемся с одним универсальным законом вычислительной техники: быстрые алгоритмы, как правило, требуют больше памяти. Таким образом, для использования быстрого алгоритма необходимо располагать большим объемом памяти.

Рассмотрим это на простом примере. Предположим, что требуется разработать алгоритм, который бы определял количество установленных бит в байте. Первый вариант алгоритма показан в листинге 1.3.

Листинг 1.3. Первоначальная функция определения количества установленных битов в байте

```
function CountBits1(B : byte) : byte;
begin
  Result := 0;
  while (B <> 0) do begin
    if Odd(B) then
      inc(Result);
    B := B shr 1;
  end;
end;
```

Как видите, в этой функции не используются промежуточные переменные. Она просто считает установленные биты путем деления значения на 2 (сдвиг целого значения на один бит вправо эквивалентно делению на 2) и определения количества полученных нечетных результатов. Цикл завершается, когда будет получено значение 0, поскольку в этом случае очевидно, что установленных битов больше нет. Значение O большого для приведенного алгоритма зависит от количества установленных битов в параметре, и в худшем случае внутренний цикл будет выполнен восемь раз. Таким образом, это алгоритм класса $O(n)$.

Описанный алгоритм вполне очевиден и, если не принимать во внимание возможность его реализации на языке ассемблера, улучшить его практически невозможно.

Тем не менее, давайте рассмотрим назначение алгоритма с другой точки зрения. В качестве входного значения функция принимает 1-байтный параметр, с помощью которого можно передать всего 256 значений. В таком случае, почему бы нам заранее не вычислить все возможные ответы и не записать их в статический массив? Реализация нового алгоритма приведена в листинге 1.4.

Листинг 1.4. Улучшенная функция определения количества установленных битов в байте

```
const
  BitCounts : array [0..255] of byte =
    (0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
     1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
     1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
     2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
     1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
     2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
     2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
     3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
     1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
     2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
     2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
     3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
     2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
     3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
     3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
     4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8);
function CountBits2(B : byte) : byte;
begin
  Result := BitCounts[B];
end;
```

Здесь за счет статического 256-байтного массива значений функция намного упростилась. Более того, приведенный алгоритм не содержит цикла. Независимо от значения входного параметра, количество установленных битов вычисляется за один простой шаг. (Обратите внимание, что значения для статического массива

были вычислены автоматически с помощью простой программы, использующей первую функцию.)

На компьютере автора книги последний алгоритм оказался в 10 раз быстрее, чем первый: 10 вызовов второй функции занимает столько же времени, сколько один вызов первой. (Обратите внимание, что здесь речь идет о среднем случае. В лучшем случае для первой функции значение параметра равно 0 и функция практически не будет требовать времени на выполнение.)

Таким образом, за счет введения 256-байтного массива мы разработали алгоритм, который быстрее в 10 раз. Увеличение скорости было достигнуто за счет увеличения требуемого объема памяти: можно получить быструю функцию, использующую большой статический массив (который будет скомпилирован в выполняемый файл, об этом также следует помнить), или более медленную функцию, не требующую больших объемов памяти. (Существует еще одна альтернатива. Можно заполнять массив значениями в процессе выполнения функции, при ее первом вызове. В этом случае массив не будет компилироваться в выполняемый файл, но первый вызов функции займет достаточно длительное время.)

Этот простой пример служит демонстрацией выбора компромиссного варианта между требуемым объемом памяти и быстродействием. Часто для увеличения скорости работы приходится вычислять все возможные результаты заранее, однако это требует дополнительной памяти.

Длинные строки

Дискуссию о быстродействии алгоритмов нельзя считать законченной без краткого рассмотрения длинных строк. С ними связан целый ряд проблем, касающихся эффективности. Длинные строки появились в Delphi 2 и присутствовали во всех последующих компиляторах Delphi и Kylix (программисты, работающие в Delphi 1, могут не беспокоиться о длинных строках и без последствий пропустить этот раздел).

Длинная строковая переменная **string** – это всего лишь указатель на специальным образом отформатированный блок памяти. Другими словами, **sizeof(stringvar) = sizeof(pointer)**. Если указатель содержит **nil**, строка считается пустой. В противном случае указатель указывает непосредственно на последовательность символов, составляющих строку. Функции для работы с длинными строками в библиотеке времени выполнения гарантируют, что строка всегда завершается нулем (**null**-символом). Благодаря этому, строковую переменную всегда можно легко привести к типу **PChar**, используемому при вызове API-функций системы. Но, наверное, не все знают, что блок памяти, на который указывает указатель, содержит и некоторую дополнительную информацию. Четыре байта, расположенные до последовательности символов, представляют собой целочисленное значение – длину строки (за исключением завершающего нуля). Предшествующие четыре байта содержат целочисленное значение, представляющее собой счет-

чик ссылок (для постоянных строк это значение равно -1). Если память для строки выделена из кучи, то предшествующие четыре байта содержат целочисленное значение, представляющее собой полный объем используемого строкой блока памяти, включая все скрытые целочисленные поля, последовательность символов, составляющих строку, и скрытый завершающий **null**-символ, округленные до ближайших четырех байтов.

Счетчик ссылок присутствует в блоке памяти, поэтому операция

```
MyOtherString := MyString
```

выполняется очень быстро. Компилятор преобразует это присвоение за два шага: сначала он увеличивает на 1 счетчик ссылок для строки, на которую указывает **MyString**, а затем устанавливает указатель **MyOtherString** равным указателю **MyString**.

Вот и все, что можно сказать об увеличении быстродействия приложения при использовании длинных строк. Все остальные операции со строками будут требовать выделения памяти.

Использование ключевого слова **const**

Если функции передать строку, которая в процессе выполнения функции не будет изменяться, объявляйте ее как **const**. В большинстве случаев это исключает скрытое добавление блока **try..finally**. Если не использовать ключевое слово **const**, компилятор будет считать, что значение, возможно, будет изменяться, и поэтому вводит скрытую локальную переменную для хранения строки. В начале выполнения функции счетчик ссылок будет увеличен на 1, а в конце – уменьшен на 1. Чтобы гарантировать, что значение счетчика всегда будет уменьшаться, компилятор вставляет скрытый блок **try..finally**.

В листинге 1.5 приведена функция определения количества гласных в строке.

Листинг 1.5. Подсчет количества гласных в строке

```
function CountVowels(const S : string) : integer;  
var  
  i : integer;  
begin  
  Result := 0;  
  for i := 1 to length(S) do  
    if upcase(S[i]) in ['A', 'E', 'I', 'O', 'U'] then  
      inc(Result);  
end;
```

Если из строки объявления функции убрать ключевое слово **const**, ее быстродействие снизится приблизительно на 12% – это и есть влияние скрытого блока **try..finally**.

Осторожность в отношении автоматического преобразования типов

Часто мы используем совместно символы и строки, не обращая на это никакого внимания. Преобразованием типов занимается компилятор, и программист зачастую не подозревает, что происходит на самом деле. Возьмем, например, функцию **Pos**. Как вы знаете, эта функция возвращает положение подстроки в строке. Если использовать ее для поиска символа:

```
PosOfCh := Pos(SomeChar, MyString);
```

нужно помнить, что компилятор автоматически преобразует символ в длинную строку. Он выделит память для строки из кучи, установит длину равной 1 и скопирует в строку символ. Затем вызывается функция **Pos**. Поскольку фактически будет использоваться скрытая длинная строка, для уменьшения значения счетчика ссылок в функцию будет автоматически добавлен блок **try..finally**. Функция, приведенная в листинге 1.6, в *пять* (да-да, в пять!) раз быстрее, несмотря на то, что она была написана на языке Pascal, а не на ассемблере.

Листинг 1.6. Определение позиции символа в строке

```
function TDPosCh(aCh : AnsiChar; const S : string) : integer;
var
  i : integer;
begin
  Result := 0;
  for i := 1 to length(S) do
    if (S[i] = aCh) then begin
      Result := i;
      Exit;
    end;
  end;
end;
```

Можно порекомендовать проверять синтаксис вызываемых функций при использовании символа, чтобы убедиться, что параметры действительно являются символами, а не строками.

И еще одна небольшая рекомендация. Операция конкатенации, **+**, тоже работает только со строками. Если конкатенация символа со строкой выполняется в цикле, попытайтесь найти для этого другой способ (например, предварительно задать длину строки, а затем присвоить значения отдельными символам строки), поскольку компилятору придется автоматически преобразовывать все символы в строки.

Тестирование и отладка

Теперь давайте на минутку забудем об анализе быстродействия алгоритмов и немного поговорим о процедурных алгоритмах – алгоритмах, предназначенных для выполнения процесса разработки, а не вычислений.

Независимо от того, как мы пишем код, в какой-то момент потребуется провести тестирование [31], дабы убедиться в том, что код работает, как задумывалось. Дает ли код правильные результаты для определенного набора входных значений? Записываются ли данные в базу данных при нажатии на кнопку ОК? Естественно, если тестирование проходит неудачно, необходимо найти ошибку и устранить ее. Этот процесс известен как отладка – тестирование показало наличие ошибки и нужно найти ее и устранить. Таким образом, тестирование и отладка неразрывно связаны между собой – по сути, это две стороны одной медали.

Поскольку мы никак не можем обойтись без тестирования (хотелось бы думать, что мы безупречны, а наш код не содержит ошибок, но, к сожалению, это не так), каким образом можно упростить этот процесс?

Вот первое золотое правило: *код всегда содержит ошибки*. И в этом нет причин для стыда. Код с ошибками – это часть нормальной работы любого программиста. Нравится вам это или нет, но программисты склонны делать ошибки. Одно из удовольствий программирования и заключается в обнаружении и устранении самых неуловимых ошибок.

Правило № 1. Код всегда содержит ошибки.

И хотя выше было сказано, что нечего смущаться при обнаружении кода с ошибками, есть все же одна ситуация, когда это выставляет вас в плохом свете – когда вы не протестировали код в достаточном объеме.

Утверждения

Поскольку первое правило гласит, что отладку нужно проводить всегда, а вывод подразумевает, что не хотелось бы краснеть за недостаточный объем тестирования кода, необходимо научиться создавать защищенные программы. И первым средством из защитного арсенала является *утверждение*.

Утверждение – это программная проверка, предназначенная для определения того, выполняется ли некоторое условие или нет. Если условие, несмотря на ваши ожидания, не выполняется, возникает исключение и на экране появляется диалоговое окно, в котором объясняется, в чем состоит проблема. Диалоговое окно – это предупреждающий сигнал о том, что либо ваше предположение было ошибочным, либо код в некоторых случаях работает не так, как ожидалось. Проверки утверждений должны привести прямо к той части кода, где содержится ошибка. Утверждения представляют собой основной элемент защитного программирования: если они присутствуют в коде, значит, вы однозначно говорите, что для дальнейшего выполнения кода в выбранной точке какое-то условие должно соблюдаться.

Джон Роббинс (John Robbins) [19] установил второе правило: "Утверждения, утверждения и еще раз утверждения". В соответствии с его книгой, он считает количество утверждений достаточным, если коллеги начинают жаловаться, что при вызове его кода они постоянно получают сообщения о проверках утверждений.

Таким образом, второе правило можно выразить так: *используйте утверждения много и часто*. Вставляйте утверждения при каждом удобном случае.

Правило № 2. Используйте утверждения много и часто.

К сожалению, некоторые программисты при использовании утверждений столкнулись с проблемами. Дело в том, что поддерживаемые компилятором утверждения появились только в версии Delphi 3. С тех пор утверждения можно применять безнаказанно. Компилятор позволяет указывать, вносить ли проверки в выполняемый файл или же игнорировать их. При тестировании и отладке компиляция выполняется с утверждениями. При создании окончательного выполняемого файла утверждения игнорируются.

В Delphi 1 и Delphi 2 приходится применять другие способы. Существует два метода. Первый – написать метод **Assert**, реализация которого должна быть пустой при создании окончательной версии приложения и будет содержать проверку с вызовом **Raise** в противном случае. Пример такой процедуры утверждения приведен в листинге 1.7.

Листинг 1.7. Процедура утверждения для Delphi 1 и Delphi 2

```
procedure Assert(aCondition : boolean; const aFailMsg : string);
begin
  {$IFDEF UseAssert}
  if not aCondition then
    raise Exception.Create(aFailMsg);
  {$ENDIF}
end;
```

Как видно из листинга, применяется директива **\$IFDEF** компилятора. Несмотря на то что приведенная процедура проста в использовании и легко может быть вызвана из основного кода, она подразумевает, что в окончательной версии приложения будет присутствовать вызов пустой процедуры. Альтернативным вариантом организации проверки утверждений может быть использование оператора **\$IFDEF** не в процедуре, а в основном коде при вызове процедуры. В таком случае блоки проверки будут выглядеть следующим образом:

```
...
{$IFDEF UseAssert}
Assert(MyPointer<>nil, "MyPointer should be allocated by now");
{$ENDIF}
MyPointer^.Field := 0;
...
```

Преимущество последнего метода заключается в том, что процедура в окончательном варианте выполняемого файла отсутствует совсем. Поскольку в настоящей книге все коды предназначены для компиляции на всех версиях Delphi, используется процедура **Assert**, код которой показан в листинге 1.7.

Проверка утверждений может применяться тремя способами: предусловие, постусловие и инвариант. *Предусловие (pre-condition)* – это утверждение, находящееся в начале функции. Оно однозначно указывает, какое условие, касающееся окружения и входных параметров, должно соблюдаться перед выполнением функции. Например, предположим, что создана функция, которой в качестве параметра передается объект. При этом разработчик решил, что функции не должен передаваться `nil`. Помимо доведения этой информации до всех разработчиков, он должен в начале функции вставить утверждение, которое будет проверять, передается ли функции `nil`. В таком случае, если сам разработчик функции или кто-то из его коллег забудет о существующем ограничении, предупреждение напомним об этом.

Постусловие (post-condition) по назначению обратно предусловию – это утверждение, находящееся в конце функции и предназначенное для проверки того, что функция была выполнена правильно. Этот тип проверки можно считать менее полезным, нежели предыдущий. В конце концов, мы программируем, подразумевая, что код будет выполняться правильно. Если при проверке постусловия возникает исключение, оставшаяся часть функции пропускается.

И последний тип утверждений – *инвариант (invariant)*. Он представляет собой нечто среднее между предусловием и постусловием. Это утверждение, которое находится в середине кода и гарантирует, что определенная часть функции выполняется корректно.

Единственной проблемой, связанной с утверждениями, является выбор случая, когда они более предпочтительны, чем "нормальные" исключения. Это малоисследованная область, но мы постараемся охватить и ее. В общем случае, ошибки, обнаруживаемые при тестировании, можно разбить на два типа: ошибки программиста и ошибки входных данных. Давайте попытаемся разобраться, в чем же состоит отличие между этими двумя типами.

Классическим примером может служить исключение "List index is out of bounds" (Индекс в списке вышел за допустимые пределы), особенно тот случай, когда используется индекс -1. Ошибка подобного типа вызвана тем, что программист не проверяет индекс элемента перед тем, как записать или считать его из `TList`. Код объекта `TList` проверяет все передаваемые ему индексы элементов. Если индекс находится вне допустимого диапазона, возникает исключение. Пользователь приложения не может вызвать такую ошибку (по крайней мере, это покажется глубоко бессмысленным для большинства пользователей). Ошибка возникает исключительно из-за недостаточного объема проведенного тестирования. По мнению автора, это исключение должно быть утверждением.

А теперь рассмотрим другой случай. Предположим, что мы разрабатываем функцию, которая должна разворачивать данные из файла, например, из архива. Формат сжатого файла достаточно сложен, по крайней мере, он считается простой последовательностью битов, и все последовательности выглядят примерно одинаково. Если в последовательности битов функция разархивирования встретит

ошибку (например, последовательность исчерпана, но логически она не закончилась), будет это утверждением или же исключением? По мнению автора, это должно быть простым исключением. Вполне вероятно, что функции в качестве входных данных будут переданы поврежденные файлы или файлы, которые не являются архивами в требуемом формате. Очевидно, что это ошибка не программиста. Она полностью вызвана внешними условиями.

Таким образом, утверждения служат для проверки качества работы программиста и предупреждают о наличии его собственных ошибок в коде, а исключения возникают в ситуациях, когда программа используется в условиях, для которых она не предназначена.

Комментарии

Это правило выглядит достаточно просто:

Правило № 3. Помещайте в код комментарии. Объясняйте ваши допущения (более того, проверяйте их с помощью утверждений). Описывайте сложные блоки кода. При изменении кода изменяйте и соответствующие комментарии. Не допускайте, чтобы комментарии устаревали.

Протоколирование

Рассмотрим еще одно средство из арсенала защитного программирования – *протоколирование* (*logging*). Под протоколированием здесь понимается вставка дополнительного кода, закрытого директивами компилятора, который записывает в файл состояние или значения основных переменных.

Этот метод уходит корнями в те времена программирования на языке Pascal, когда программисты при любом удобном случае вставляли оператор **writeln** и надеялись, что он поможет обнаружить ошибку. В наши дни ценность этого метода существенно снизилась. Автор книги зачастую для протоколирования состояния классов пишет методы **DumpToFile**. Их можно помещать в условные блоки компилятора, вызывать несколько раз в стратегически важных точках и получать описание всего жизненного цикла определенного объекта.

Правило № 4. Пишите код протоколирования и защищайте его директивами компилятора. Однажды протокол может пригодиться, и вполне вероятно, что такой день таки наступит.

В кодах, приведенных в книге, будут приведены примеры использования этого метода.

Трассировка

В прошлом трассировка была тесно связана с протоколированием. *Трассировка* (*tracing*) представляла собой метод вставки операторов **writeln** в начале и в конце функций. Операторы применялись для вывода на экран или в файл таких

простых сообщений, как "Вход в функцию X" или "Выход из функции X". Запись сообщений в файл помогала восстановить ход выполнения приложения и порядок вызова функций. В настоящее время существуют специальные программы, которые делают все это сами, без участия программиста. Приложение запускается внутри такой программы, а она автоматически идентифицирует все функции и подпрограммы, их начало и завершение, и формирует журнал трассировки приложения. Никаких изменений вносить в код не потребуется.

В последнее время большинство программистов не пользуются трассировкой. Намного проще запустить отладчик и при возникновении ошибки просмотреть стек вызовов.

Анализ покрытия

Это современный метод и для его использования вам понадобится специальное программное обеспечение. *Анализ покрытия (coverage analysis)* представляет собой запись в журнал того, какие операторы приложения были "покрыты", т.е. выполнены. Если при тестировании отдельная строка или блок кода не выполняются, в этой строке или блоке может содержаться ошибка. Такую ошибку можно будет выявить только с помощью теста, при выполнении которого выполняется код с ошибкой.

Правило № 5. При тестировании пользуйтесь анализатором покрытия. Убедитесь, что во время тестирования выполняются все строки кода.

Тестирование модулей

Тестирование модулей (unit testing) представляет собой процесс тестирования отдельных частей независимо от самой программы.

Одним из новых методов разработки программного обеспечения, который появился уже при написании этой книги, является *экстремальное программирование (extreme programming)* [3]. Этот метод состоит в целом наборе рекомендаций. Некоторые рекомендации достаточно спорны, но, по крайней мере, одна из них имеет смысл: пишите тест тогда, когда вы пишете метод класса. Если метод требует не одного теста, разработайте несколько тестов. Такой порядок обладает двумя преимуществами: во-первых, код вам знаком – вы только что его написали, и, во-вторых, в дальнейшем разработанный тест может быть включен в тестовый набор и применяться для тестирования приложения после внесения в него изменений. Таким образом, вы можете быть уверены, что изменения не повлекли за собой возникновение ошибок.

Эта рекомендация, вероятно, не соответствует тому, как большинство из нас тестирует программы. Мы, как правило, пишем блоки кода, а затем, два-три месяца спустя, пытаемся объединить их в одно целое с множеством других блоков и только потом приступаем к тестированию всей системы.

Тестирование модулей требует наличия специального средства, которое помогало бы собирать тесты, поддерживать их актуальность и периодически, в автоматическом режиме, запускать их с целью проверки правильности кода. К счастью, существует одна библиотека с открытым исходным кодом, которую можно использовать свободно, – **DUnit**. Она представляет собой порт для Delphi инструментальных средств тестирования Java-модулей, частично написанный автором книги "Extreme Programming Explained" Кентом Беком (Kent Beck). (**DUnit** можно использовать, только начиная с версии Delphi 3.)

DUnit представляет собой *средство тестирования*, или *тестовый каркас*, реализованный на Delphi. Используя его, программист пишет отдельные тесты, предназначенные для проверки своего кода. Тесты могут быть совсем простыми (например, в тесте может создаваться объект, проверяться значения заданных по умолчанию свойств, после чего объект удаляется), но в общем случае они должны быть предназначены для выполнения всего кода класса или модуля. (Чтобы убедиться, что выполняются все строки кода, можно воспользоваться анализатором покрытия.) Сам тестовый каркас предоставляет пользовательский интерфейс, который позволяет программисту выбрать один или несколько тестов и запустить их. После выполнения теста или тестов программист может просмотреть результаты: успешное выполнение или ошибка (**DUnit** при выводе результатов использует различные цвета, благодаря чему результат выполнения теста можно оценить с первого взгляда.) Конечно, по истечении некоторого времени тест может оказаться неактуальным, поскольку, например, класс настолько изменился, что выполнение существующего теста не позволяет определить правильность кода. В таком случае тест потребуется написать заново.

Правило № 6. Для организации набора тестов для проверки модулей воспользуйтесь тестовым каркасом. При изменении кода выполните тесты повторно.

Если у вас есть **DUnit** для определенного класса или модуля, его можно использовать для *регрессионного тестирования* (*regression testing*). Регрессионное тестирование представляет собой тестирование всего класса или модуля после внесения изменений в этот класс или модуль. Часто поиск и устранение одной ошибки приводит к возникновению другой ошибки.

Рассмотрим пример. В библиотеке TurboPower Internet Professional (библиотека Delphi для реализации таких протоколов Internet, как FTP, HTTP и т.д.) имеется функция, которая разбивает URL на различные части. URL-адрес может указывать на Web-сайт или на FTP-сайт, это может быть относительный путь (например, путь к графическому изображению на Web-странице, может указываться относительно папки, в которой находится основная Web-страница), или MAILTO-адрес, или просто файл на жестком диске. Формат URL-адреса достаточно сложен. Его можно видеть в адресной строке Web-браузера. Синтаксический

разбор URL-адреса представляет собой весьма сложную задачу, которая, к сожалению, не достаточно четко определена.

Примером может служить URL-адрес перечня опечаток для настоящей книги – `http://www.boyet.com/dads`. Он состоит из трех частей. Первая, "`http://`" определяет протокол, вторая, "`www.boyet.com`", указывает имя сервера, а третья, "`/dads`" – имя папки на сервере.

Перед написанием пакета тестов для проверки модуля синтаксического разбора URL-адресов было вполне обычным делом внести исправление, которое позволяло правильно разбирать одну часть адреса, но вызывало ошибку при разборе другой части адреса.

Написание пакета тестов для проверки модуля синтаксического разбора URL-адресов позволило одним выстрелом убить сразу нескольких зайцев. Во-первых, теперь можно быть уверенным, что устранение ошибок в одной части модуля не приведет к возникновению ошибок в другой части модуля. Во-вторых, это дало возможность разработать методы кодирования URL-адреса и методы его синтаксического разбора. Внесение дополнительных тестов стало очень простой задачей. В-третьих, это позволило изменять код модуля с целью его упрощения, при этом разработанные тесты гарантируют правильность получаемого результата.

Тестовый каркас **DUnit** можно найти в Internet по адресу `http://dunit.sourceforge.net`. Все коды, приведенные в книге, были протестированы с помощью тестов, написанных с использованием **DUnit**. Некоторые тесты включены в материалы, сопровождающие книгу, которые доступны на Web-сайте издательства.

Отладка

При разработке приложений всегда наступает момент, когда приходится переходить к поиску и устранению ошибок. В настоящей книге мы не будем подробно описывать процесс отладки, давать советы по использованию отладчика и описывать методы поиска и устранения основных типов ошибок. Здесь будут приведены лишь основные правила, которые позволят читателю существенно упростить сам процесс отладки. Все они взяты из книги Роббинса (Robbins) [19].

Правило отладки № 1. Выбирайте воспроизводимый случай тестирования.

По нечеткому описанию проблемы можно обнаружить только самые простые ошибки. Тестовый случай, который при необходимости может воспроизвести ошибку, – это, по крайней мере, 90% на пути к ее обнаружению и устранению. Возможность воспроизведения ошибки позволяет с помощью отладчика определить место, где она возникает. Если же у вас нет теста, который может воспроизвести ошибку, то у вас нет и надежды.

Второе правило отладки намного сложнее.

Правило отладки № 2. Исходите из того, что ошибка внесена вами.

Может быть, вы неправильно используете API-интерфейс операционной системы или библиотека компонентов требует определенной последовательности операций. Или, в конце концов, может быть, вызываемая вами функция не может принимать `nil` в каком-либо входном параметре. Маловероятно, чтобы ошибка была вызвана неправильной работой API-интерфейса или компилятора. Более вероятно, что ошибка присутствует в библиотеке компонентов, тем не менее, попытайтесь выделить проблему (см. правило отладки 1), что позволит с уверенностью сказать, что ошибка находится не в вашем коде. Конечно, если ошибка находится не в вашем коде, ваша задача только усложняется, поскольку придется положиться на разработчиков операционной системы, компилятора или библиотеки компонентов, что они достаточно быстро устранят имеющуюся проблему.

Следующее правило вытекает из уже рассмотренного нами материала.

Правило отладки № 3. Для проверки того, что код работает, как того ожидалось, используйте утверждения.

Кроме того, можно применять и протоколирование, которое позволит отслеживать состояние различных объектов.

А теперь перейдем собственно к отладке.

Правило отладки № 4. Используйте автоматизированные инструментальные средства отладки.

Возможно, ваша ошибка вызвана перезаписью какой-то области памяти или получением доступа к памяти после того, как она была освобождена, либо, скажем, при вызове API-функции не проверяется код возвращаемой ошибки. Все описанные типы проблем можно обнаружить в таком автоматизированном средстве отладки, как TurboPower Sleuth QA Suite. Приобретите средство отладки и используйте его не только в процессе тестирования, но и в процессе обнаружения ошибок.

Естественно, после этого следует сама отладка, которая может выполняться даже с помощью отладчика. Именно здесь наука программирования превращается в настоящее искусство. Отладку нельзя назвать алгоритмом, скорее, это соревнование. Единственным советом в этом соревновании может быть "не делайте никаких допущений". Если вы считаете, что некоторая переменная должна содержать определенное значение, проверьте это. Пользуйтесь средством визуализации значений отладчика. Для наблюдения за блоком памяти можно воспользоваться окном процессора. Попробуйте предсказать значения переменных, а затем проверить свои предположения и при необходимости устраните ошибки. Без боязни добавляйте в код новые утверждения с целью проверки своих предположений.

Резюме

Эта глава была достаточно насыщена информацией и, честно говоря, была посвящена не столько алгоритмам и структурам данных, сколько технологиям увеличения быстродействия кода и процедурным методам.

Иногда быстродействие является результатом правильного выбора алгоритма или структуры данных (или того и другого). В других случаях увеличение скорости может быть достигнуто благодаря глубоким знаниям оборудования компьютера и операционной системы. Прежде всего, важно понимать, что единственным методом увеличения быстродействия приложения является использование профилировщика. Только предоставляемая профилировщиком статистика поможет определить, на что приложение тратит время, и лишь глубокое изучение отдельных блоков кода позволит оптимизировать приложение и увеличить его быстродействие. Хотелось бы подчеркнуть, что просто выбор "правильного" алгоритма или структуры данных отнюдь не означает, что приложение будет работать быстрее. Приведенная в книге информация поможет вам понять возможные способы ускорения выполнения отдельных частей кода, но только не нужно вносить в код сложные реализации алгоритмов там, где этого не требуется – вы только зря потеряете время.

Кроме того, в этой главе мы кратко рассмотрели тестирование и отладку – это тоже не алгоритмы, а скорее методы, гарантирующие, что код не содержит ошибок. А если ошибки все-таки присутствуют, то в коде находится столько разного рода следов и указателей, что поиск и устранение ошибок не должны отнимать много времени.



Глава 2

Массивы

Несмотря на то что при стандартном (и не совсем стандартном) программировании используется огромное количество разного рода структур данных, большинство из них основаны на одном из двух фундаментальных контейнеров: массив и связный список. Если после прочтения этой книги вы научитесь правильно применять эти два типа структур, цель книги можно будет считать достигнутой. Они важны не только благодаря своей простоте, но и вследствие своей высокой эффективности. Массивы будут подробно рассмотрены в этой главе, а связные списки – в следующей. Кроме того, в главе 3 после связных списков будут описаны некоторые простые типы структур данных, основанные на этих двух фундаментальных типах. В главах 4 и 5, посвященных поиску и сортировке соответственно, мы также коснемся фундаментальных типов структур данных, но под несколько другим углом.

Невзирая на то что в книге будут приводиться полные реализации этих двух типов структур данных, иногда будет удобнее написать свою собственную реализацию. Поэтому важно четко понимать все аспекты, которые будут рассматриваться в этой и последующих главах.

Массивы

Во многих отношениях массивы являются простейшей структурой данных. Проще могут быть только такие базовые типы данных, как **integer** или **Boolean**. *Массив (array)* представляет собой последовательный список определенного количества элементов. Все элементы в массиве принадлежат к одному типу данных, и, как правило, хранятся в одном блоке памяти, т.е. каждый последующий элемент в

памяти находится непосредственно после предыдущего. В таком случае говорят, что элементы массива являются *смежными* в памяти. Если ссылаться на элементы массива по их числовым индексам, то первый элемент будет иметь индекс 0 (или 1, или любое другое число, по крайней мере, в Delphi), значение индекса второго элемента будет больше на единицу и т.д. В коде элемент с индексом i обозначается как $A[i]$, где A – идентификатор массива.

В Delphi имеется большой набор встроенных типов массивов. Кроме того, отдельные удобные типы массивов определены в библиотеке визуальных компонент VCL (Visual Component Library) в виде классов (и не только классов). Для поддержки таких классов, как массивы, разработчики Delphi предусмотрели возможность перегрузки операции массива, `[]`, добавляя к нему новые свойства. Это единственная операция в Delphi, помимо `+` (сложение и конкатенация строк), которую можно перегружать.

Типы массивов в Delphi

В Delphi имеется три типа поддерживаемых языком массивов. Первый – стандартный массив, который объявляется с помощью ключевого слова **array**. Второй тип был впервые введен в Delphi 4 в качестве имитации того, что было давным-давно доступно в Visual Basic, – динамический массив, т.е. массив, длина которого может изменяться в процессе выполнения кода.

И последний тип массивов, как правило, не считается массивом, хотя в языке Object Pascal имеется несколько его вариаций. Конечно, мы говорим о строках: однобайтных строках (тип **shortstring** в 32-разрядной версии Delphi), строках с завершающим нулем (тип **Pchar**) и длинных строках в 32-разрядных версиях Delphi (которые имеют отдельную вариацию для "широких" символов).

Все массивы имеют одну и ту же структуру. Они состоят из одного или большего количества повторений другого типа данных, например, **char**, **integer** или **record**, которые в памяти находятся рядом друг с другом. Именно это последнее свойство стандартных массивов позволяет очень быстро получить доступ к отдельным элементам массивов. Весь процесс доступа к элементу сводится к простому вычислению адреса, для чего требуются, как мы вскоре увидим, всего несколько машинных инструкций.

Стандартные массивы

Можно даже не сомневаться, что все вы знаете стандартный способ объявления массивов в Delphi. Так, объявление

```
var  
  MyIntArray : array [0..9] of integer;
```

создает массив из 10 элементов типа **integer**. В языке Object Pascal диапазон изменения индексов элементов можно выбирать любым (в приведенном случае – от 0

до 9). В следующем примере объявляется еще один массив из 10 элементов типа **integer**, но здесь индексация элементов следует от 1 до 10:

```
var  
  MyIntArray : array [1..10] of integer;
```

Некоторые считают, что работать с массивом, объявленным во втором примере, удобнее (в конце концов, *первый* элемент имеет индекс 1).

Тем не менее, нужно сказать несколько слов о работе с массивами, индексация которых начинается с нуля. Во-первых, очень часто в API-интерфейсах операционных систем Windows и Linux, а также Delphi-библиотеках VCL и CLX предполагается, что первый элемент в массиве имеет индекс 0. Кроме того, в языках программирования C, C++ и Java индексация *всех* массивов обязательно начинается с 0. Поскольку и Windows, и Linux реализованы на C (или C++), при вызове API-функций считается, что индекс первого элемента массива равен 0.

Во-вторых, индексация динамических массивов начинается с 0. Поэтому, если вы хотите использовать этот очень гибкий тип, начинайте нумерацию элементов массивов с 0.

В-третьих, если вы передаете массивы в качестве параметров функциям (скоро мы перейдем к рассмотрению открытых массивов), то функция **Low** (которая возвращает индекс первого элемента массива) внутри некоторой функции будет возвращать 0 независимо от того, как массив объявлен вне этой функции. (Обратите внимание, что сказанное было справедливо для всех версий Delphi на момент написания книги; в будущих версиях, возможно, будет введена возможность индексирования элементов массивов в функциях по реальным индексам.)

Еще один момент, о котором необходимо помнить, – для основных типов массивов, элементы которых располагаются в памяти непрерывно, вычисление адреса элемента N (т.е. элемента **MyArray[N]**) в случае индексации с 0 производится по следующему выражению:

```
AddressOfElementN := AddressOfArray + (N * sizeof(ElementType));
```

Если же индексация массива начинается с X , то адрес элемента N будет вычисляться в соответствии с выражением:

```
AddressOfElementN :=  
  AddressOfArray + ((N - X) * sizeof(ElementType));
```

Как видите, в последнем случае выражение несколько сложнее. Несмотря на то что вычисление адреса каждого отдельного элемента лишь немного медленнее (о снижении быстродействия можно даже не говорить), при использовании нескольких массивов снижение скорости работы приложения вследствие того, что индексация элементов всех массивов начинается не с 0, может оказаться весьма существенной.

И в качестве последнего аргумента в пользу применения массивов, индексация элементов которых начинается с нуля, может служить удобство вычислений и программирования. Например, если доступ ко всем элементам осуществляется в цикле **For**, компилятор получает возможность оптимизировать цикл таким образом, чтобы последним значением переменной цикла был 0, поскольку сравнение с 0 в конце цикла будет быстрее, нежели сравнение с произвольным числом. В книге можно будет встретить несколько таких примеров. Таким образом, учитывая все вышесказанное, имеет смысл использовать массивы, первый элемент которых имеет индекс 0.

Так что же такого замечательного в использовании массивов в качестве структуры данных? Во-первых, вычисление адресов элементов выполняется очень быстро. Как уже говорилось, для этого нужно всего лишь умножение и сложение. При получении доступа к элементу N (**MyArray[N]**) компилятор для вычисления адреса добавляет простой машинный код. Независимо от значения числа N , формула для вычисления адреса будет одной и той же. Другими словами, получение доступа к элементу с индексом N принадлежит к классу операций $O(1)$ и не зависит от величины N .

Во-вторых, можно вспомнить понятие локальности ссылок. Элементы массива расположены в памяти последовательно друг за другом. При последовательном прохождении всех элементов сама операционная система способствует высокой скорости работы, поскольку в одной странице памяти будут находиться сразу несколько элементов, поэтому дополнительные операции обмена страницами между диском и памятью выполнять не придется.

До сих пор мы рассматривали только преимущества массивов, но хотелось бы знать и об их недостатках. Первый недостаток связан с операциями вставки и удаления элементов. Что происходит, если, например, в массив необходимо вставить новый элемент с индексом n ? В общем случае, все элементы с индексами, начиная с n и до конца массива, потребуется переместить на одну позицию, чтобы освободить место под новый элемент. А фактически выполняется следующий блок кода:

```
{ сначала освободить место под новый элемент }
for i := LastElement downto N do
  MyArray[i+1] := MyArray[i];
{ вставить новый элемент в позицию с индексом N }
MyArray[N] := NewElement;
{ увеличить значение длины массива на единицу }
inc(LastElementIndex);
```

(Конечно, на практике цикл заменяется вызовом процедуры **Move**.)

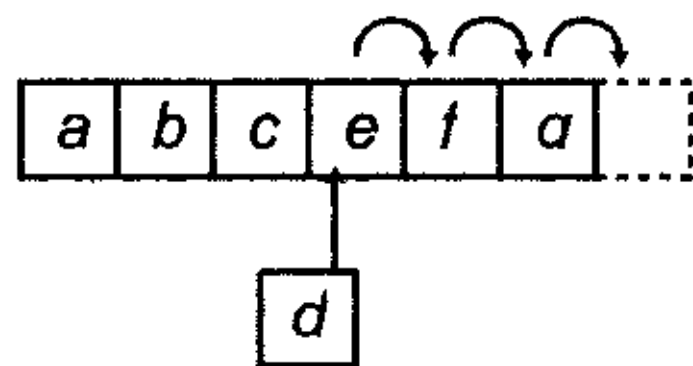


Рисунок 2.1. Вставка в массив нового элемента

Объем памяти, который будет затронут при вставке нового элемента, зависит от значения n и количества элементов в самом массиве. Чем больше количество элементов, которые необходимо переместить, тем больше времени потребуется на выполнение операции. То есть, время, требуемое на выполнение цикла **For**, будет пропорционально количеству элементов в массиве. Другими словами, вставка нового элемента в массив принадлежит к классу операций $O(n)$.

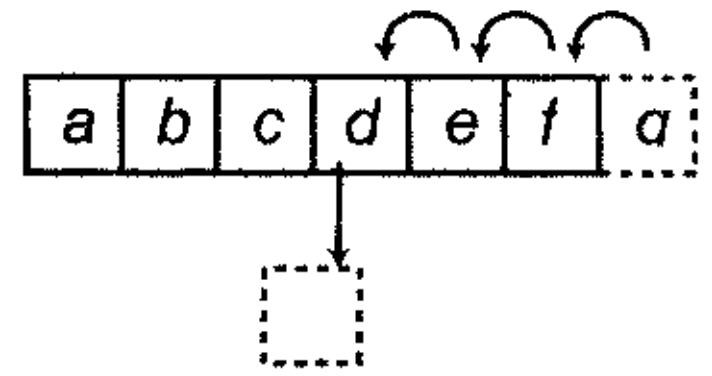


Рисунок 2.2. Удаление элемента из массива

Тот же ход рассуждений справедлив и для операции удаления элемента из массива. Но в этом случае удаление элемента с индексом n означает, что элементы, начиная с индекса $n + 1$ и до конца массива, будут перенесены на одну позицию к началу массива, чтобы "закрыть" образовавшуюся от удаления элемента "дыру". Как и в случае со вставкой, удаление принадлежит к классу операций $O(n)$.

{удалить элемент, переместив следующие за ним элементы на одну позицию вперед}

for $i := N+1$ **to** LastElementIndex **do**

$\text{MyArray}[i-1] := \text{MyArray}[i];$

{уменьшить значение длины массива на единицу}

$\text{dec}(\text{LastElementIndex});$

(Конечно, на практике цикл заменяется вызовом процедуры **Move**.)

Таким образом, важно понимать, что операции вставки и удаления элемента при увеличении количества элементов в массиве будут выполняться медленнее, поскольку они принадлежат к классу операций $O(n)$.

Кроме того, есть еще один вопрос, связанный со вставкой и удалением элементов, – необходимо контролировать количество активных элементов, т.е. в качестве последнего элемента массива нужно ввести *сигнальный* (*sentinel*) элемент, который будет использоваться в качестве метки конца массива. (В строках с завершающим нулем таким сигнальным элементом является символ `#0`.) Как правило, во время компиляции объявляются массивы фиксированного размера (сейчас мы говорим о методах увеличения размеров массивов), а, следовательно, для этого нам необходимо знать количество активных элементов. В двух приведенных выше примерах для хранения количества активных элементов использовалась переменная **LastElementIndex**. В строках и длинных строках, например, в самой строке, содержится счетчик количества символов. Но если мы не планируем использовать вставку или удаление элементов, никаких дополнительных элементов не требуется.

Стоит упомянуть и об еще одной проблеме, которая касается только программирования в Delphi 1. В Delphi 1 максимальный объем непрерывного выделяемого блока памяти (по крайней мере, без написания дополнительного кода на ассемблере) равен 64 Кб. Если объем одного элемента массива составляет 100 байт, то это означает, что в массиве не может быть больше 655 таких элементов. Не так уж и

много. Это 64-Кбное ограничение может вызвать определенные проблемы и привести к тому, что придется использовать указатели на элементы (как, например, в знаменитом классе **TList**), а не сами элементы (в массиве **TList** в Delphi 1 количество элементов ограничено числом 16 383).

Динамические массивы

Часто приходится сталкиваться с программированием процедур, которые требуют использования массива, причем количество элементов в таком массиве заранее не известно – их может быть десять, сто или тысяча, но окончательно количество элементов будет известно только во время выполнения процедур. Более того, из-за незнания количества элементов, его трудно объявить как локальную переменную (объявление массива с максимально возможным количеством элементов может привести к перегрузке стека, особенно это касается Delphi 1). Таким образом, память под элементы массива лучше выделять из кучи.

Но даже в этом случае не все недостатки устраняются. Предположим, что вы решили, что количество элементов в массиве не может превысить 100. Но никогда не говорите "никогда", поскольку в один прекрасный день количество элементов может оказаться 101. Это приведет к перезаписи памяти или возникновению ошибок нарушения доступа (если, конечно, в коде не использовались утверждения, которые проверяли возможность превышения количества элементов над ожидаемым значением).

Одним из методов, которые уходят корнями еще к временам языка Pascal, является создание типа массива со всего одним элементом и указателя на этот массив:

type

PMmyArray : ^TMyArray;

TMyArray : **array**[0..0] of TMyType;

Теперь, если вам необходим массив типа **TMyType**, можно легко указать требуемое количество элементов:

var

MyArray : PMmyArray;

begin

...

GetMem(MyArray, 42*sizeof(TMyType));

... использование массива MyArray ...

FreeMem(MyArray, 42*sizeof(TMyType));

Обратите внимание, что процедура **FreeMem** при освобождении выделенного блока памяти только в Delphi 1 требует указания размера блока. Все 32-разрядные версии Delphi и Kylix хранят размер выделенного блока в самом блоке. Размер блока находится непосредственно перед блоком, который код получает с помощью процедуры **GetMem**. В последних версиях Delphi передаваемый в качестве входного параметра размер блока игнорируется, а вместо него используется скрытое значение.

До освобождения памяти **MyArray** указывает на массив, состоящий из 42 элементов типа **TMyType**. Несмотря на свою простоту, приведенный метод обладает некоторыми недостатками, о которых всегда нужно помнить. Во-первых, такой код нельзя компилировать с включенной проверкой диапазонов (**\$R+**), поскольку компилятор считает, что массив должен содержать только один элемент, а, следовательно, может использоваться только индекс 0.

(От этого недостатка можно избавиться, если при объявлении массива указать, что он содержит не один элемент, а некоторое, достаточно большое, количество элементов. Но такое решение приносит свою проблему: все индексы до указанной верхней границы будут действительными. Так, например, если выделить массив из 42 элементов, основанный на массиве из 1000 элементов, то для компилятора индексы от 42 до 999 также будут действительными.)

Тем не менее, описанный метод очень широко применяется в повседневном программировании. Например, в модуле **SysUnit** содержится очень гибкий тип массива **TByteArray**, указатель на который имеет тип **PByteArray**. Используя этот тип (точнее сказать, указатель на тип) можно легко преобразовывать любой нетипизированный параметр, содержащийся в буфере, в массив байтов. Существуют и другие типы массивов: массив элементов типов **longint**, **word** и т.д.

Наиболее удобным методом решения второй проблемы является создание класса массива, который бы позволил выделять произвольное количество элементов, получать доступ и задавать значения отдельных элементов и даже уменьшать или увеличивать количество элементов в массиве. Другие возможности, например, сортировка, удаление и вставка, тоже были бы оказаться очень кстати. Фактически, программист создавал бы экземпляр класса, объявляя в конструкторе размер каждого элемента, а выделением памяти под элементы занимался бы сам класс.

Обратите внимание, что мы здесь говорим не о классе **TList**. **TList**, к рассмотрению которого мы вскоре перейдем, представляет собой массив указателей. По сути, при использовании массива **TList** память для размещения каждого отдельного элемента выделяется из кучи, а затем код просто манипулирует указателями на элементы.

Вместо этого давайте создадим структурный тип массива, **TtdRecordList**, который по функциям был бы аналогичен классу **TList**, но выделял память для самих элементов. Интерфейс такого класса приведен в листинге 2.1.

Если вы уже знакомы с интерфейсом класса **TList**, то наверняка обратите внимание, что класс **TtdRecordList** содержит все те же методы и свойства, что и **TList**. Таким образом, например, метод **Add** будет добавлять новый элемент в конец списка, а **Insert** — вставлять в список новый элемент в позицию с заданным индексом. Оба метода при необходимости будут приводить к расширению внутренней структуры массива, и увеличивать счетчик элементов. Метод **Sort** в этой главе мы рассматривать не будем. Описание его реализации будет приведено в главе 5.

ЛИСТИНГ 2.1. Объявление класса TtdRecordList

```

TtdRecordList = class
  private
    FActElemSize : integer;
    FArray : PAnsiChar;
    FCount : integer;
    FCapacity : integer;
    FElementSize : integer;
    FMaxElemCount : integer;
    FName : TtdNameString;
  protected
    function rlGetItem(aIndex : integer) : pointer;
    procedure rlSetCapacity(aCapacity : integer);
    procedure rlSetCount(aCount : integer);
    procedure rlError(aErrorCode : integer;
      const aMethodName : TtdNameString; aIndex : integer);
    procedure rlExpand;
  public
    constructor Create(aElementSize : integer);
    destructor Destroy; override;
    function Add(aItem : pointer) : integer;
    procedure Clear;
    procedure Delete(aIndex : integer);
    procedure Exchange(aIndex1, aIndex2 : integer);
    function First : pointer;
    function IndexOf(aItem : pointer;
      aCompare : TtdCompareFunc) : integer;
    procedure Insert(aIndex : integer; aItem : pointer);
    function Last : pointer;
    procedure Move(aCurIndex, aNewIndex : integer);
    function Remove(aItem : pointer;
      aCompare : TtdCompareFunc) : integer;
    procedure Sort(aCompare : TtdCompareFunc);
    property Capacity : integer read FCapacity write rlSetCapacity;
    property Count : integer read FCount write rlSetCount;
    property ElementSize : integer read FActElemSize;
    property Items[aIndex : integer] : pointer read rlGetItem; default;
    property MaxCount : integer read FMaxElemCount;
    property Name : TtdNameString read FName write FName;
end;

```

Конструктор **Create** сохраняет переданный ему размер элементов и вычисляет размер каждого элемента, округляя его до ближайших 4 байт. Округление будет гарантировать, что элементы всегда выровнены по границе 4 байт. Это вызвано соображениями увеличения скорости работы. В качестве последней операции, конструктор будет вычислять максимальное количество элементов, которое может содержаться в классе при заданном размере одного элемента. Фактически такая опе-

рация необходима только для Delphi 1, поскольку в этой версии максимальный объем выделяемой из кучи памяти не может превышать 64 Кб и нужно убедиться, что мы не выходим за установленную границу.

Листинг 2.2. Конструктор класса TtdRecordList

```
constructor TtdRecordList.Create(aElementSize : integer);
begin
    inherited Create;
    {сохранить фактический размер элемента}
    FActElemSize := aElementSize;
    {округлить фактический размер элемента до 4 байт}
    FElementSize := ((aElementSize + 3) shr 2) shl 2;
    {вычислить максимальное количество элементов}
    {$IFDEF Delphi1}
    FMaxElemCount := 65535 div FElementSize;
    {$ELSE}
    FMaxElemCount := MaxInt div FElementSize;
    {$ENDIF}
end;
```

Обратите внимание, что класс не выделяет память для элементов массива. Выделение памяти происходит при добавлении элементов или, другими словами, при фактическом использовании экземпляра класса.

(В коде, приведенном в листинге 2.2, используется нестандартная директива компилятора – **Delphi1**. Эта директива определена во включаемом файле **TDDefine.inc**, который применяется во всех приведенных в книге модулях. Директиву **Delphi1** намного легче запомнить, чем ее более официальное название **VER80**. Кроме того, официальное название сложнее запомнить, поскольку свое официальное название имеет для каждой версии. Так, например, для Delphi 3 – это **VER100**, для Delphi 4 – **VER120** и т.д. Тем не менее, существуют и соответствующие неофициальные названия – **Delphi3** и **Delphi4**.)

Деструктор ничуть не сложнее конструктора. В нем мы просто устанавливаем емкость экземпляра класса равным 0 (немного ниже мы подробно рассмотрим, что такое емкость) и вызываем унаследованный деструктор **Destroy**.

Листинг 2.3. Деструктор класса TtdRecordList

```
destructor TtdRecordList.Destroy
begin
    Capacity := 0;
    inherited Destroy;
end;
```

А теперь давайте перейдем к более интересным вещам: добавлению и вставке новых элементов. Реализация метода **Add** достаточно проста. В ней вызывается **Insert** для вставки нового элемента в конец массива. Метод **Insert** в качестве

входного параметра принимает значение, представляющее собой индекс позиции, в которую требуется вставить новый элемент. Сам элемент задается указателем (есть еще один способ представления вставляемого элемента – в виде нетипизированного параметра **var**, однако указатели позволяют упростить реализацию и понимание других методов и, кроме того, обеспечивают непротиворечивость). При вызове метода **Insert** для передачи адреса вставляемого элемента в виде указателя используется операция **@**, определенная в Delphi.

Поскольку новый элемент является указателем, он может содержать **nil**, поэтому сначала необходимо проверить, что указатель не равен **nil**. Затем в реализации метода выполняется проверка выхода индекса за границы допустимого диапазона. Только после этого можно приступить к собственно вставке. Если количество элементов равно текущей емкости массива, то для расширения массива вызывается метод **rlExpand**. Теперь мы перемещаем элементы, начиная с индекса **aIndex** до конца массива, на один элемент, дабы тем самым освободить место под новый элемент. И, наконец, мы вставляем элемент в образовавшуюся "дыру" и увеличиваем значение счетчика элементов на единицу.

Листинг 2.4. Добавление и вставка новых элементов

```
function TtdRecordList.Add(aItem : pointer) : integer;
begin
    Result := Count;
    Insert(Count, aItem);
end;

procedure TtdRecordList.Insert(aIndex : integer; aItem : pointer);
begin
    if (aItem = nil) then
        rlError(tdeNilItem, 'Insert', aIndex);
    if (aIndex < 0) or (aIndex > Count) then
        rlError(tdeIndexOutOfBounds, 'Insert', aIndex);
    if (Count = Capacity) then
        rlExpand;
    if (aIndex < Count) then
        System.Move((FArray + (aIndex * FElementSize))^,
                    (FArray + (succ(aIndex) * FElementSize))^,
                    (Count - aIndex) * FElementSize);
        System.Move(aItem^,
                    (FArray + (aIndex * FElementSize))^, FActElemSize);
        inc(FCount);
    end;
```

Реализация метода **Delete**, предназначенного для удаления элементов из массива, показана в листинге 2.5. Как и для **Insert**, сначала проверяется переданный методу индекс, затем элементы, начиная с индекса **aIndex**, переносятся на одну позицию к началу массива, за счет чего требуемый элемент удаляется. После удаления количество элементов в массиве уменьшается, поэтому из значения счетчика элементов вычитается единица.

Листинг 2.5. Удаление элемента массива

```

procedure TtdRecordList.Delete(aIndex : integer);
begin
  if (aIndex < 0) or (aIndex >= Count) then
    rLError(tdeIndexOutOfBounds, 'Delete', aIndex);
  dec(FCount);
  if (aIndex < Count) then
    System.Move((FArray + (succ(aIndex) * FElementSize))^,
      (FArray + (aIndex * FElementSize))^,
      (Count - aIndex) * FElementSize);
end;

```

Метод **Remove** аналогичен **Delete** в том, что с его помощью также удаляется отдельный элемент, но при этом не требуется знание его индекса в массиве. Нужный элемент находится с помощью метода **IndexOf** и вспомогательной процедуры сравнения, которая является внешней по отношению к классу. Таким образом, метод **Remove** требует не только самого удаляемого элемента, но и вспомогательной процедуры, которая бы идентифицировала элемент, подлежащий удалению. Такая процедура имеет тип **TtdCompareFunc**. Она будет вызываться для каждого элемента массива до тех пор, пока возвращаемое значение для определенного элемента не окажется нулевым (что означает "равно"). Если процедура выполняется для всех элементов, а нулевое возвращаемое значение так и не получено, метод **IndexOf** возвращает значение **tdc_ItemNotPresent**.

Листинг 2.6. Методы Remove и IndexOf

```

function TtdRecordList.Remove(aItem : pointer;
  aCompare : TtdCompareFunc) : integer;

begin
  Result := IndexOf(aItem, aCompare);
  if (Result <> tdc_ItemNotPresent) then
    Delete(Result);
end;

function TtdRecordList.IndexOf(aItem : pointer;
  aCompare : TtdCompareFunc) : integer;

var
  ElementPtr : PAnsiChar;
  i : integer;
begin
  ElementPtr := FArray;
  for i := 0 to pred(Count) do begin
    if (aCompare(aItem, ElementPtr) = 0) then begin
      Result := i;
      Exit;
    end;
    inc(ElementPtr, FElementSize);
  end;
  Result := tdc_ItemNotPresent;
end;

```

Для расширения массива (т.е. для увеличения его емкости) используется свойство **Capacity**. При его установке вызывается защищенный метод **rlSetCapacity**. Реализация метода несколько сложнее, чем могла бы быть. Это вызвано тем, что процедура **ReAllocMem** в версии Delphi 1 не делает всего того, что она делает в 32-разрядных версиях.

Соответствующий метод назван **rlExpand**. Это защищенный метод, построенный на базе простого алгоритма и предназначенный для установки значения свойства **Capacity** на основе его текущего значения. Метод **rlExpand** вызывается автоматически при использовании метода **Insert** для увеличения емкости массива, если будет определено, что в настоящее время массив полностью заполнен (т.е. емкость равна количеству элементов в массиве).

Листинг 2.7. Расширение массива

```

procedure TtdRecordList.rlExpand;
var
    NewCapacity : integer;
begin
    {если текущая емкость массива равна 0, установить новую емкость равной
     4 элемента}
    if (Capacity = 0) then
        NewCapacity := 4
    {если текущая емкость массива меньше 64, увеличить ее на 16 элементов}
    else if (Capacity < 64) then
        NewCapacity := Capacity + 16
    {если текущая емкость массива 64 или больше, увеличить ее на 25%}
    else
        NewCapacity := Capacity + (Capacity div 4);
    {убедиться, что мы не выходим за верхний индекс массива}
    if (NewCapacity > FMaxElemCount) then begin
        NewCapacity := FMaxElemCount;
        if (NewCapacity = Capacity) then
            rlError(tdeAtMaxCapacity, 'rlExpand', 0);
        end;
    {установить новую емкость}
    Capacity := NewCapacity;
end;

procedure TtdRecordList.rlSetCapacity(aCapacity : integer);
begin
    if (aCapacity <> FCapacity) then begin
        {запретить переход через максимально возможное количество элементов}
        if (aCapacity > FMaxElemCount) then
            rlError(tdeCapacityTooLarge, 'rlSetCapacity', 0);
    
```

```

{повторно распределить или освободить память, если емкость массива
  уменьшена до нуля}
{$IFDEF Delphi1}
if (aCapacity = 0) then begin
  FreeMem(FArray, word(FCapacity) * FElementSize);
  FArray := nil;
end
else begin
  if (FCapacity = 0) then
    GetMem(FArray, word(aCapacity) * FElementSize)
  else
    FArray := ReallocMem(FArray,
      word(FCapacity) * FElementSize,
      word(aCapacity) * FElementSize);
end;
{$ELSE}
ReallocMem(FArray, aCapacity * FElementSize);
{$ENDIF}
{емкость уменьшается? если да, проверить счетчик}
if (aCapacity < FCapacity) then begin
  if (Count > aCapacity) then
    Count := aCapacity;
end;
{сохранить новую емкость}
FCapacity := aCapacity;
end;
end;

```

Конечно, любой класс массива оказался бы бесполезным, если бы было невозможно считать элемент из массива. В классе **TtdRecordList** для этой цели имеется свойство **Items**. Единственным средством доступа для этого свойства является метод считывания **rlGetItem**. Во избежание ненужного копирования данных в элемент, метод **rlGetItem** возвращает указатель на элемент массива. Это позволяет не только считать, но и легко изменить элемент. Именно поэтому для свойства **Items** нет специального метода записи. Поскольку свойство отмечено ключевым словом **default**, доступ к отдельным элементам можно получить с помощью кода **MyArray[i]**, а не **MyArray.Items[i]**.

Листинг 2.8. Получение доступа к элементу массива

```

function TtdRecordList.rlGetItem(aIndex : integer) : pointer;
begin
  if (aIndex < 0) or (aIndex >= Count) then
    rlError(tdeIndexOutOfBounds, 'rlGetItem', aIndex);
  Result := pointer(Tarray + (aIndex * FelementSize));
end;

```

И последний метод, который мы сейчас рассмотрим, – это метод, используемый для установки свойства **Count** – **rlSetCount**. Установка свойства **Count** позволяет предварительно выделить память для элементов массива и работать с ней аналогично тому, как Delphi работает со стандартными массивами. Обратите внимание, что методы **Insert** и **Delete** будут автоматически изменять значение свойства **Count** при вставке и удалении элементов. Установка свойства **Count** явным образом будет гарантировать и корректную установку свойства **Capacity** (метод **Insert** делает это автоматически). Если новое значение свойства **Count** больше текущего, значения всех новых элементов будут равны нулю. В противном случае элементы, индексы которых больше или равны новому количеству элементов, станут недоступными (фактически их можно будет считать удаленными).

Листинг 2.9. Установка количества элементов в массиве

```

procedure TtdRecordList.rlSetCount(aCount : integer);
begin
  if (aCount <> FCount) then begin
    {если новое значение количества элементов в массиве больше емкости массива,
     расширить массив}
    if (aCount > Capacity) then
      Capacity := aCount;
    {если новое значение количества элементов в массиве больше старого
     значения, установить значения новых элементов равными нулю}
    if (aCount > FCount) then
      FillChar((FArray + (FCount * FElementSize))^,
               (aCount - FCount) * FElementSize, 0);
    {сохранить новое значение счетчика элементов}
    FCount := aCount;
  end;
end;

```

Полный код класса **TtdRecordList** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDRecLst.pas**. В файле находятся также реализации таких стандартных методов, как **First**, **Last**, **Move** и **Exchange**.

Новые динамические массивы

В Delphi 4 компания Borland ввела *динамические массивы* – расширение языка, которое позволило использовать массивы, размер которых на этапе программирования не известен. Код, вносимый компилятором в приложение, аналогичен тому, который используется для длинных строк. Как и для строк, размер массива можно установить с помощью стандартной процедуры **SetLength**. Кроме того, динамические массивы ведут счетчики ссылок. И даже больше, функция **Copy** перегружена, что позволяет копировать отдельные части массива. Как и для стандартных стати-

ческих массивов, доступ к отдельным элементам осуществляется с помощью операции `[]`.

В настоящей книге мы не будем подробно рассматривать динамические массивы. Их применение ограничено, поскольку они доступны только в версиях, начиная с Delphi 4 и Kylix. И, кроме того, они не имеют той функциональности, которую нам предоставляет класс **TtdRecordList**. Если вы хотите больше узнать о динамических массивах, изучите документацию по своей версии Delphi.

Класс **TList**, массив указателей

С самой первой версии в Delphi существовал еще один стандартный массив — класс **TList**. В отличие от всех ранее нами рассмотренных массивов, **TList** представляет собой массив указателей.

Краткий обзор класса **TList**

Класс **TList** хранит указатели в формате массива. Указатели могут быть любыми. Они могут указывать на записи, строки или объекты. Класс имеет специальные методы для вставки и удаления элементов, поиска элемента в списке, перестановки элементов и, в последних версиях компилятора, для сортировки элементов в списке. Как и любой другой массив, **TList** может использовать операцию `[]`. Поскольку свойство **Items** является свойством по умолчанию, то для получения доступа к указателю с индексом *i* вместо **MyList.Item[i]** можно записывать **MyList[i]**. Индексация в классе **TList** всегда начинается с 0.

Несмотря на высокую гибкость класса **TList**, иногда при его использовании возникают проблемы.

Одна из проблем встречается очень часто: при уничтожении экземпляра **TList** память, выделенная под оставшиеся в нем элементы, не освобождается. В некотором роде это даже преимущество, поскольку можно быть уверенным, что **TList** никогда не освободит память, используемую его элементами. Один и тот же элемент можно поместить одновременно в несколько списков, не боясь, что он будет удален по ошибке. К сожалению, многие программисты склонны считать, что **TList** работает точно так же, как любой компонент формы (т.е. при уничтожении формы уничтожаются и все ее компоненты). Но в отношении **TList** это не так, поэтому необходимо отдельно позаботиться о том, чтобы при уничтожении списка удалялись и все его элементы.

Однако существует еще одна трудноуловимая ошибка, которую очень часто совершают при написании кода удаления всех элементов из списка. Во многих случаях код удаления выглядит следующим образом:

```
for i := 0 to pred(MyList.Count) do begin
  if SomeConditionApplies(i) then begin
    TObject(MyList[i]).Free;
    MyList.Delete(i);
  end;
end;
```

где **SomeConditionApplies** – некоторая произвольная функция, которая определяет, удалять или нет элемент с индексом i .

Все мы привыкли к тому, что значение переменной цикла должно увеличиваться. Именно в этом-то и заключается ошибка. Предположим, что в массиве находится три элемента. В таком случае код в цикле будет выполнен три раза: для индексов 0, 1 и 2. Пусть при первом выполнении цикла условие выполняется. При этом освобождается объект с индексом 0, а затем элемент с индексом 0 удаляется из списка. После первого выполнения цикла в списке остается два элемента, но их индексы теперь 0 и 1, а не 1 и 2. При втором выполнении цикла, при соблюдении условия, освобождается объект с индексом 1 (который, если вы помните, был изначально элементом с индексом 2), после чего удаляется элемент с индексом 1. После этого в списке остается всего один элемент. И его индекс 0. При третьем выполнении цикла код пытается освободить память, ранее выделенную под объект, индекс которого 2, и в результате генерируется исключение "list index out of bounds".

Правильно было бы организовать обратный цикл, который бы начал удалять элементы с конца списка. Так мы могли бы избежать ошибки.

Для освобождения всех элементов списка используется следующий код, а не вызов метода **Delete** для каждого элемента:

```
for i := 0 to pred(MyList.Count) do  
  TObject(MyList[i]).Free;  
end;
```

Еще одной проблемой при использовании класса **TList** является создание производного класса. Если попытаться это сделать, можно столкнуться с разного рода проблемами, вызванными тем, что методы **TList** являются статическими, к тому же имеют приватные поля, которые не доступны, и т.д. Можно только посоветовать не пытаться порождать новые классы от **TList**. **TList** – это *не* тот класс, на основе которого можно создавать производные классы. Он был создан не таким расширяемым, как, например, **TString**. При необходимости можно создать отдельный класс, который для хранения данных использует класс **TList**. Применяйте в данном случае *делегирование*, а не *наследование*.

При первом написании предыдущего параграфа автор книги не знал, что компания Borland сделала с классом **TList** в версии Delphi 5. В Delphi 5 по каким-то непостижимым причинам было изменено функционирование класса **TList** с целью обеспечения поддержки нового производного класса – **TObjectList**. **TObjectList** предназначен для хранения экземпляров объектов. Он находится в модуле **Contrns**, о котором мы поговорим чуть позже.

Что же изменилось? В версиях до Delphi 5 **TList** очищался путем освобождения внутреннего массива указателей, что было операцией класса $O(1)$. Поскольку компания Borland хотела, чтобы класс **TObjectList** при определенных условиях мог освобождать содержащиеся в нем объекты, она для обеспечения такой функ-

циональности изменила основной принцип работы **TList**. В Delphi, начиная с версии 5, и, конечно же, Kylix, класс **TList** очищается путем вызова для каждого элемента нового виртуального метода **Notify**. Метод **TList.Notify** не выполняет никаких операций, но метод **TObjectList.Notify** при удалении элементов из списка освобождает занимаемую ими память.

Вы можете спросить: "Ну и что?" Дело в том, что этот новый метод очистки содержимого класса **TList** принадлежит к операциям класса $O(n)$. Таким образом, чем больше элементов в списке, тем больше времени потребуется на его очистку. По сравнению с предыдущими версиями **TList**, новая версия стала работать гораздо медленнее. Каждый экземпляр каждого класса, использующего **TList**, теперь будет работать медленнее. И помните, единственной причиной снижения быстродействия стало нежелание компании Borland воспользоваться делегированием, вместо наследования. По мнению компании, было намного удобнее изменить стандартный класс.

И что еще хуже с точки зрения объектно-ориентированного программирования, мы получили ситуацию, когда для поддержки производного класса был изменен родительский класс. **TList** не должен освобождать свои элементы – это стало правилом еще с версии Delphi 1. Тем не менее, он был изменен для того, чтобы такая возможность поддерживалась его дочерними классами (а фактически только одним классом из VCL Delphi 5 – классом **TObjectList**).

Денни Торп (Denny Thorpe), один из самых толковых разработчиков в отделе научных исследований компании Borland, в своей книге "Разработка компонент Delphi" (Delphi Component Design) [23] сказал следующее:

"**TList** – это рабочая лошадка, а не порождающий класс... При необходимости использования списка создайте простой интерфейсный класс (порожденный от **TObject**, а не от **TList**), который будет иметь только нужные вам эквиваленты свойств и функций **TList**, и возвращаемые типы значений функций и параметров методов которого соответствуют типу хранящихся в списке данных. В таком случае интерфейсный класс будет содержать в себе класс **TList** и использовать его для хранения данных. Реализуйте функции и методы доступа к свойствам этого интерфейсного класса в виде однострочных вызовов соответствующих методов или свойств внутреннего класса **TList** с применением соответствующих преобразований типов".

Очень жаль, что книга Денни Торпа не внесена в перечень книг, обязательных для прочтения в отделе научных исследований компании Borland.

Класс **TtdObjectList**

А сейчас мы создадим новый класс списка, который работает как **TList**, но имеет два отличия: он хранит экземпляры некоторого класса (или его дочерних классов) и при необходимости уничтожает все содержащиеся в нем объекты. Другими словами, это будет специализированный список, в котором не будет двух

описанных в предыдущем разделе недостатков. Назовем наш класс **TtdObjectList**. Он отличается от класса **TObjectList** в Delphi 5 и более поздних версиях тем, что будет безопасным к типам.

Он не будет дочерним классом **TList**. Конечно, в нем будут содержаться те же методы, но их реализация будет основана на делегировании к методам с теми же именами внутреннего класса **TList**.

Класс **TtdObjectList** имеет один новый очень важный атрибут – владение данными. Это класс будет функционировать либо точно так же, как **TList**, т.е. при уничтожении его элементы не будут освобождаться (он не владеет данными), либо будет иметь полный контроль над своими элементами и при необходимости будет их удалять (он владеет данными). Установка атрибута владения данными выполняется при создании экземпляра класса **TtdObjectList**, и после этого уже не будет возможности изменить тип владения данными.

Кроме того, класс будет обеспечивать *безопасность к типам (type safety)*. При создании экземпляра класса, необходимо указывать какой тип (или класс) объектов будет в нем храниться. Во время добавления или вставки нового элемента специальный метод будет проверять соответствие типа нового объекта объявленному типу элементов списка.

Интерфейс класса **TtdObjectList** напоминает интерфейс класса **TList**. В нем не реализован метод **Pack**, поскольку в список будут добавляться только объекты, не равные **nil**. Подробное описание метода **Sort** будет приведено в главе 5.

Листинг 2.10. Объявление класса **TtdObjectList**

```
TtdObjectList = class
  private
    FClass : TClass;
    FDataOwner : boolean;
    FList : TList;
    FName : TtdNameString;
  protected
    function olGetCapacity : integer;
    function olGetCount : integer;
    function olGetItem(aIndex : integer) : TObject;

    procedure olSetCapacity(aCapacity : integer);
    procedure olSetCount(aCount : integer);
    procedure olSetItem(aIndex : integer; aItem : TObject);
    procedure olError(aErrorCode : integer;
                      const aMethodName : TtdNameString;
                      aIndex : integer);
  public
    constructor Create(aClass : TClass; aDataOwner : boolean);
    destructor Destroy; override;

    function Add(aItem : TObject) : integer;
    procedure Clear;
```

```

procedure Delete(aIndex : integer);
procedure Exchange(aIndex1, aIndex2 : integer);
function First : TObject;
function IndexOf(aItem : TObject) : integer;
procedure Insert(aIndex : integer; aItem : TObject);
function Last : TObject;
procedure Move(aCurIndex, aNewIndex : integer);
function Remove(aItem : TObject) : integer;
procedure Sort(aCompare : TtdCompareFunc);

property Capacity : integer read olGetCapacity write olSetCapacity;
property Count : integer read olGetCount write olSetCount;
property DataOwner : boolean read FDataOwner;
property Items[Index : integer] : TObject
    read olGetItem write olSetItem; default;
property List : TList read FList;
property Name : TtdNameString read FName write FName;
end;

```

Целый ряд методов класса **TtdObjectList** является простыми интерфейсами для вызова соответствующих методов внутреннего класса **FList**. Например, вот реализация метода **TtdObjectList.First**:

Листинг 2.11. Метод **TtdObjectList.First**

```

function TtdObjectList.First : TObject;
begin
    Result := TObject(FList.First);
end;

```

В тех методах, которые в качестве входного параметра принимают индекс, до вызова соответствующего метода класса **FList** индекс проверяется на предмет попадания в допустимый диапазон. Строго говоря, эта процедура не обязательна, поскольку сам класс **FList** будет производить аналогичную проверку, но в случае возникновения ошибки методы класса **TtdObjectList** позволят получить больший объем информации. Вот один из примеров – метод **Move**:

Листинг 2.12. Метод **TtdObjectList.Move**

```

procedure TtdObjectList.Move(aCurIndex, aNewIndex : integer);
begin
    {проверяем индексы сами, а не перекладываем эту обязанность на список}
    if (aCurIndex < 0) or (aCurIndex >= FList.Count) then
        olError(tdeIndexOutOfBounds, 'Move', aCurIndex);
    if (aNewIndex < 0) or (aNewIndex >= FList.Count) then
        olError(tdeIndexOutOfBounds, 'Move', aNewIndex);
    {переместить элементы}
    FList.Move(aCurIndex, aNewIndex);
end;

```

Конструктор класса в качестве входных параметров принимает тип объектов, которые будут храниться в списке (чем обеспечивается безопасность класса к типам), и атрибут владения данными. После этого создается внутренний экземпляр класса **FList**. Деструктор очищает список и освобождает память, занимаемую списком.

Листинг 2.13. Конструктор и деструктор класса **TtdObjectList**

```

constructor TtdObjectList.Create(aClass : TClass;
                                aDataOwner : boolean);
begin
    inherited Create;
    {сохранить класс и флаг владения данными}
    FClass := aClass;
    FDataOwner := aDataOwner;
    {создать внутренний список}
    FList := TList.Create;
end;

destructor TtdObjectList.Destroy;
begin
    {если список содержит элементы, очистить их и уничтожить список}
    if (FList <> nil) then begin
        Clear;
        FList.Destroy;
    end;
    inherited Destroy;
end;

```

Если вы не уверены, каким образом передавать значение параметра **aClass**, приведем пример с использованием класса **TButton**:

```

var
    MyList := TtdObjectList;
begin
    . . .
    MyList := TtdObjectList.Create(TButton, false);

```

Первым реальным отличием нового списка от стандартного класса **TList** является метод **Clear**. Он предназначен для проверки того, владеет ли список данными. В случае положительного результата, перед уничтожением списка все его элементы будут удалены. (Обратите внимание, что здесь для удаления каждого отдельного элемента не используется метод **Delete** класса **FList**. Намного эффективнее очищать список после освобождения памяти, занимаемой его элементами.)

Листинг 2.14. Метод **TtdObjectList.Clear**

```

procedure TtdObjectList.Clear;
var
    i : integer;

```

```

begin
  {если данные принадлежат списку, перед очисткой списка освобождаем память,
   занимаемую элементами}
  if DataOwner then
    for i := 0 to pred(FList.Count) do
      TObject(FList[i]).Free;
    FList.Clear;
end;

```

Методы **Delete** и **Remove** перед удалением выполняют один и тот же тип проверки, и если список владеет данными, объект освобождается, после чего удаляется и список. Обратите внимание, что в методе **Remove** используется не вызов метода **FList.Remove**, а полная реализация метода. Такой подход называется "кодированием на основе главных принципов". Он обеспечивает более глубокий контроль и дает более высокую эффективность.

Листинг 2.15. Удаление элемента из списка TtdObjectList

```

procedure TtdObjectList.Delete(aIndex : integer);
begin
  {проверяем индексы сами, а не перекладываем эту обязанность на список}
  if (aIndex < 0) or (aIndex >= FList.Count) then
    olError(tdeIndexOutOfBounds, 'Delete', aIndex);
  {если список владеет объектами, освобождаем память, занимаемую удаляемым
   элементом}
  if DataOwner then
    TObject(FList[aIndex]).Free;
  {удалить элемент из списка}
  FList.Delete(aIndex);
end;

function TtdObjectList.Remove(aItem : TObject) : integer;
begin
  {найти требуемый элемент}
  Result := IndexOf(aItem);
  {если элемент найден...}
  if (Result <> -1) then begin
    {если список владеет объектами, освобождаем память, занимаемую удаляемым
     элементом}
    if DataOwner then
      TObject(FList[Result]).Free;
    {удалить элемент из списка}
    FList.Delete(Result);
  end;
end;

```

В методе **olSetItem** (метод записи свойства **Items** массива), который устанавливает значение или вставляет элемент в список, можно обнаружить небольшой недостаток. Предположим, что программист написал следующий блок кода:

```
var
  MyObjectList : TtdObjectList;
  SomeObject : TObject;
begin
  . . .
  MyObjectList[0] := SomeObject;
```

Все кажется довольно-таки безобидным, но подумайте, что случится, если данные принадлежат списку. В результате выполнения оператора присваивания элемент с индексом 0 будет замещен новым объектом, **SomeObject**. Предыдущий объект будет безвозвратно потерян, и ссылки на него окажутся недействительными. Таким образом, перед заменой старый объект нужно освободить. Конечно, сначала следует проверить принадлежит ли новый объект к требуемому типу.

Листинг 2.16. Запись элемента в TtdObjectList

```
procedure TtdObjectList.olSetItem(aIndex : integer; aItem : TObject);
begin
  {проверить тип элемента}
  if (aItem = nil) then
    olError(tdeNilItem, 'olSetItem', aIndex);
  if not (aItem is FClass) then
    olError(tdeInvalidClassType, 'olSetItem', aIndex);
  {проверяем индексы сами, а не перекладываем эту обязанность на список}
  if (aIndex < 0) or (aIndex >= FList.Count) then
    olError(tdeIndexOutOfBounds, 'olSetItem', aIndex);
  {если список владеет объектами и объект с текущим индексом должен быть
  заменен новым объектом, сначала освобождаем старый объект}
  if DataOwner and (aItem <> FList[aIndex]) then
    TObject(FList[aIndex]).Free;
  {сохранить в списке новый объект}
  FList[aIndex] := aItem;
end;
```

И, наконец, рассмотрим методы **Add** и **Insert**. Как и **Remove**, метод **Add** написан с учетом главных принципов, поэтому вместо **FList.Add** используется **FList.Insert**.

Листинг 2.17. Методы Add и Insert класса TtdObjectList

```
function TtdObjectList.Add(aItem : TObject) : integer;
begin
  {проверить тип элемента}
  if (aItem = nil) then
    olError(tdeNilItem, 'Add', FList.Count);
```

```

if not (aItem is FClass) then
  olError(tdeInvalidClassType, 'Add', FList.Count);
  {вставить новый элемент в конец списка}
  Result := FList.Count;
  FList.Insert(Result, aItem);
end;

procedure TtdObjectList.Insert(aIndex : integer; aItem : TObject);
begin
  {проверить тип элемента}
  if (aItem = nil) then
    olError(tdeNilItem, 'Insert', aIndex);
  if not (aItem is FClass) then
    olError(tdeInvalidClassType, 'Insert', aIndex);
  {проверяем индексы сами, а не перекладываем эту обязанность на список}
  if (aIndex < 0) or (aIndex > FList.Count) then
    olError(tdeIndexOutOfBounds, 'Insert', aIndex);
  {вставить новый элемент в список}
  FList.Insert(aIndex, aItem);
end;

```

Полный код класса **TtdObjectList** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDObjLst.pas**.

Массивы на диске

Одним из приложений массивов, которое описывается во многих книгах, – это массивы на диске (или, если хотите, дисковые массивы, но не путайте их с RAID!), т.е. файлы записей фиксированной длины. Этот тип массивов обладает своими собственными особенностями, заслуживающими отдельного рассмотрения, после которого мы напишем класс, заключающий в себе файл записей (или данных). Постоянные массивы известны как файлы данных или файлы записей, а элементы таких массивов представляют собой записи. Индекс элементов в постоянных массивах называется порядковым номером записи.

Язык Pascal всегда поддерживал файлы записей и Delphi продолжает эту традицию. Стандартный метод работы с файлами записей выглядит следующим образом:

```

var
  MyRecord : TMyRecord;
  MyFile : file of TMyRecord;
begin
  {открыть файл данных}
  System.Assign(MyFile, 'MyData.DAT');
  System.Rewrite(MyFile);

```

```
try
    {сохранить запись в позицию 0}
    ..установить поля MyRecord..
    System.Write(MyFile, MyRecord);

    {считать запись с позиции 0}
    System.Seek(MyFile, 0);
    System.Read(MyFile, MyRecord);

finally
    System.Close(MyFile);
end;
end;
```

В приведенном блоке кода открывается файл данных (процедуры **Assign** и **Rewrite**), затем в файл записывается новая запись (процедура **Write**) и, наконец, запись считывается (процедуры **Seek** и **Read**). Обратите внимание, что перед считыванием необходимо с помощью процедуры **Seek** установить указатель позиции в файле на начало записи. Если этого не сделать, будет считана вторая запись файла. Код примера включает блок **try..finally**, который гарантирует, что файл будет закрыт независимо от того, что происходит при выполнении процедуры **Rewrite**.

Однако в приведенном примере способа получения доступа к записям файла присутствуют две ошибки. Первая из них, хотя и небольшая, тем не менее, очень важная. Единственным методом определения размера каждой записи является считывание ее из исходного кода программы, которая осуществляет доступ к файлу. Если есть файл записей, то для определения длины записи необходимо поработать с окном шестнадцатеричного представления. Если длина записи и объем файла известны, можно легко определить количество записей в файле.

И вторая проблема – файлы данных не содержат информации о структуре записей, количестве полей и их типах. Если бы в файле хранился больший объем информации, работать с записями и самими файлами было бы намного проще.

Какую информацию, помимо записей, потребовалось бы хранить в файле? Как уже говорилось, одним из дополнительных полей могла бы быть длина записи, а вторым – количество находящихся в файле записей. При помощи этих двух полей можно определить допустимость файла (т.е. равен ли объем файла количеству записей, умноженному на длину записи, плюс размер служебной информации).

Предположим, что в файле находится специальный служебный блок данных. Пусть этот блок содержит некоторые важные данные о файле, за которыми следует определенное количество записей одинакового размера. Другими словами, служебный блок данных содержит постоянную информацию о массиве (размер элемента, количество элементов и, может быть, ряд других данных).

В таком случае мы можем написать свой класс, который будет открывать файл и вносить в него записи (и, конечно, соответствующим образом изменять содержимое служебного блока), считывать записи по заданному порядковому номеру, за-

писывать и обновлять записи по порядковому номеру и закрывать файл. А как же удаление записей? Не хотелось бы перемещать записи в файле на одну позицию с целью закрытия "дыры", образованной после удаления одной записи, как мы это делали в массивах в памяти. Подобная процедура заняла бы слишком много времени.

Существует два возможных решения для организации удаления записей. Первое – самое простое, которое используется в файлах данных dBASE. Для каждой записи в файле устанавливается префикс, состоящий из одного байта и содержащий флаг удаления. Флаг может быть булевым значением (**true/false**) или символом (например, 'Y'/'N' или '*'/'пусто'). При удалении записи устанавливается флаг удаления, который и будет говорить о том, что данная запись удалена. Все кажется достаточно простым, но что делать с удаленными записями? Вариант А – просто игнорировать. К сожалению, в этом случае в файле будет накапливаться все большее и большее число удаленных записей и в некоторый момент времени файл придется *уплотнять*, дабы избавиться от ненужных записей и уменьшить размер файла данных. Вариант В – повторно использовать место, занимаемое удаленными записями. При добавлении в файл новой записи по файлу выполняется поиск удаленной записи, на место которой и будет добавлена новая запись. Очевидно, что вариант В неэффективен. Представьте себе, что в файле, содержащем 10000 записей, удалена только одна запись. Для того чтобы найти всего одну удаленную запись, нам придется выполнить цикл, по крайней мере, по 5000 записям. Эта операция принадлежит к классу $O(n)$, поэтому вариант В лучше не реализовывать.

Тем не менее, вариант В имеет и свои положительные стороны, в частности, повторное использование места, занимаемого удаленными записями. Если бы только нам удалось привести его к классу $O(1)$! Такие рассуждения привели к разработке еще одного метода удаления записей – цепочке удаленных записей (для этого метода наличие служебного блока данных обязательно, поэтому будем считать, что служебные данные присутствуют).

Перед каждой записью находится 4-байтный префикс – значение типа **longint**. Он предназначен для хранения флага удаления. Его нормальное значение -1 – значение, которое указывает, что запись не удалена. Любое другое значение будет означать, что запись удалена. Но это еще не все. Обратите внимание, что размер каждой записи увеличивается на 4 байта. В свою очередь, пользователь считает, что размер записи не изменился. В служебном заголовке хранится еще одно значение типа **longint**, которое представляет собой порядковый номер первой удаленной записи. Нормальное значение для этого поля -2, которое означает, что в файле нет удаленных записей.

При удалении первой записи мы поступаем следующим образом. Сначала устанавливаем значение флага удаления записи равным значению поля порядкового номера первой удаленной записи служебного заголовка, т.е. значению -2. Затем значение флага удаления записывается на диск. После этого в поле порядкового

номера первой удаленной записи служебного заголовка записываем порядковый номер только что удаленной записи. В результате получаем следующее: во-первых, значение флага удаления записи не равно -1 (т.е. теперь запись отмечена как удаленная) и, во-вторых, поле порядкового номера первой удаленной записи служебного заголовка теперь указывает на удаленную запись (т.е. запись, место, занимаемое которой, можно использовать повторно).

При удалении второй записи выполняются все те же операции. После них флаг второй удаленной записи будет содержать порядковый номер первой удаленной записи (не равный -1, что говорит о том, что запись удалена), а поле первой удаленной записи служебного заголовка будет указывать на вторую удаленную запись.

А что происходит при добавлении в файл новой записи? Вместо простого добавления записи в конец файла, как мы делали раньше, проверяем значение поля порядкового номера удаленной записи в служебном заголовке. Если значение не равно -1, значит, существует запись, занимаемое которой место можно использовать повторно. При вставке новой записи потребуется изменить содержащееся в служебном заголовке значение. Если этого не сделать, при последующем добавлении записи она снова будет записана на то же место, а предыдущая запись будет потеряна. В этом случае мы считываем флаг удаления записи, занимаемое которой место будет использоваться повторно, и переносим его в поле первой удаленной записи служебного заголовка данных. Обратите внимание, что при повторном использовании последней удаленной записи в поле первой удаленной записи служебного заголовка будет установлено значение -2, поскольку флаг удаления записи содержал это значение.

Есть еще один вопрос, который проще рассмотреть на примере кода. Было бы довольно глупо ограничить концепцию постоянных (устойчивых) массивов только до файлов на диске. Несмотря на то что в подавляющем большинстве случаев будут использоваться файлы, ничто не мешает нам организовать постоянный массив в памяти или на любом другом устройстве хранения данных. Было бы удобно иметь класс постоянного массива, который пользуется потоками. В Delphi предусмотрен богатый набор классов потоков, включая файловый поток. Таким образом, если мы напишем код, использующий класс **TStream**, его можно будет применять со всеми другими классами, порожденными от **TStream**.

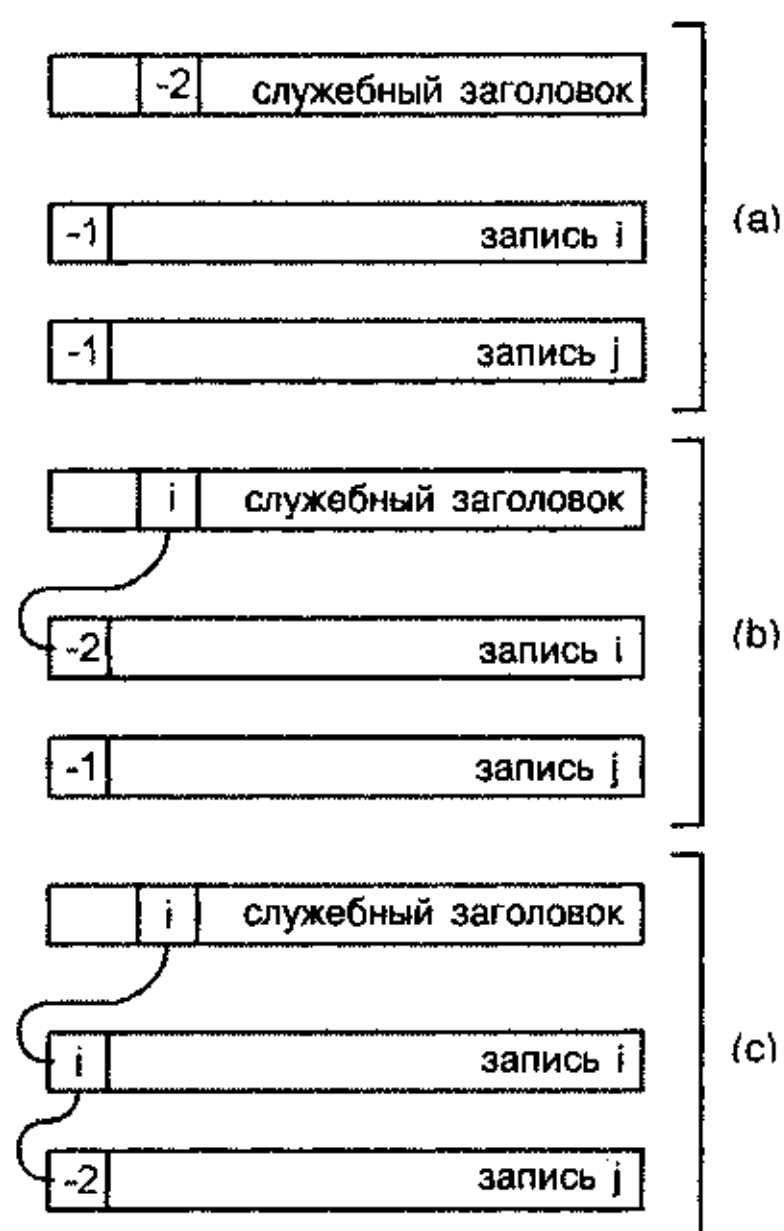


Рисунок 2.3. Удаление записи

Ниже приведен код класса **TtdRecordStream** – класса, предназначенного для постоянного хранения в потоке массива записей.

Листинг 2.18. Класс TtdRecordStream для хранения постоянных массивов.

```

type
  TtdRecordStream = class
  private
    FStream : TStream;
    FCount : longint;
    FCapacity : longint;
    FHeaderRec : PtdRSHeaderRec;
    FName : TtdNameString;
    FRecord : PByteArray;
    FRecordLen : integer;
    FRecordLen4 : integer;
    FZeroPosition : longint;
  protected
    procedure rsSetCapacity(aCapacity : longint);
    procedure rsError(aErrorCode : integer;
      const aMethodName : TtdNameString; aNumValue : longint);
    function rsCalcRecordOffset(aIndex : longint) : longint;
    procedure rsCreateHeaderRec(aRecordLen : integer);
    procedure rsReadHeaderRec;
    procedure rsReadStream(var aBuffer; aBufLen : integer);
    procedure rsWriteStream(var aBuffer; aBufLen : integer);
    procedure rsSeekStream(aOffset : longint);
  public
    constructor Create(aStream : TStream; aRecordLength : integer);
    destructor Destroy; override;
    procedure Flush; virtual;
    function Add(var aRecord) : longint;
    procedure Clear;
    procedure Delete(aIndex : longint);
    procedure Read(aIndex : longint;
      var aRecord;
      var aIsDeleted : boolean);
    procedure Write(aIndex : longint;
      var aRecord);
    property Capacity : longint read FCapacity write rsSetCapacity;
    property Count : longint read FCount;
    property RecordLength : integer read FRecordLen;
    property Name : TtdNameString read FName write FName;
  end;

```

К сожалению, для такого типа постоянных массивов очень сложно перегрузить операцию **[]**, поэтому в классе **TtdRecordStream** свойство **Items** не используется. Вместо него введены простые методы **Read** и **Write**.

Конструктор **Create** может вызываться в двух режимах: для постоянного массива в потоке или не в потоке. Режим определяется самим конструктором и в случае, если используется новый поток, создается служебный блок.

Листинг 2.19. Конструктор класса **TtdRecordStream**

```

constructor TtdRecordStream.Create(aStream : TStream;
                                   aRecordLength : integer);
begin
    inherited Create;
    {сохранить поток и его текущую позицию}
    FStream := aStream;
    FZeroPosition := aStream.Position;
    {если размер потока равен нулю, нужно создать служебный заголовок}
    if (aStream.Size - FZeroPosition = 0) then
        rsCreateHeaderRec(aRecordLength)
    {в противном случае проверить, содержится ли в потоке действительный
     служебный заголовок, считать его и установить значения его полей}
    else
        rsReadHeaderRec;
    {выделить память под запись}
    FRecordLen4 := FRecordLen + sizeof(longint);
    GetMem(FRecord, FRecordLen4);
end;

```

Обратите внимание, что конструктор считывает текущее положение потока и записывает его в **FZeroPosition**. Текущее положение, которое, как правило, равно нулю, будет использоваться для указания положения служебного заголовка для постоянного массива. Это означает, что перед вызовом конструктора **Create** программист может записать в поток свой служебный заголовок, и методы класса не будут его изменять. Тем не менее, класс предполагает, что оставшаяся часть потока, начиная с положения **FZeroPosition**, принадлежит классу и в нее допускается вносить изменения.

Конструктор вызывает либо метод **rsCreateHeaderRec**, который создает новый служебный заголовок для пустого потока (т.е. при необходимости создания нового массива), либо метод **rsReadHeaderRec**, который считывает текущий служебный заголовок (и, кроме того, проверяет его корректность).

И, наконец, конструктор **Create** выделяет из кучи память для записи (память выделяется с учетом размера флага удаления). Деструктор **Destroy** освобождает память, выделенную под запись.

Листинг 2.20. Деструктор класса **TtdRecordStream**

```

destructor TtdRecordStream.Destroy;
begin
    if (FHeaderRec<>nil) then
        FreeMem(FHeaderRec, FHeaderRec^.hrHeaderLen);
end;

```

```

if (FRecord<>nil) then
    FreeMem(FRecord, FRecordLen4);
    inherited Destroy;
end;

```

А теперь давайте рассмотрим два вспомогательных метода, которые соответственно создают новый или считывают существующий служебный заголовок.

Листинг 2.21. Создание и считывание служебного заголовка

```

procedure TtdRecordStream.rsCreateHeaderRec(aRecordLen : integer);
begin
    {выделить память под служебный заголовок}
    if ((aRecordLen + sizeof(longint)) < sizeof(TtdRSHeaderRec)) then
        begin
            FHeaderRec := AllocMem(sizeof(TtdRSHeaderRec));
            FHeaderRec^.hrHeaderLen := sizeof(TtdRSHeaderRec);
        end
    else begin
        FHeaderRec := AllocMem(aRecordLen + sizeof(longint));
        FHeaderRec^.hrHeaderLen := aRecordLen + sizeof(longint);
    end;
    {задать значения остальных стандартных полей}
    with FHeaderRec^ do begin
        hrSignature := cRSSignature;
        hrVersion := $00010000; {Major=1; Minor=0}
        hrRecordLen := aRecordLen;
        hrCapacity := 0;
        hrCount := 0;
        hr1stDelRec := cEndOfDeletedChain;
    end;
    {обновить служебный заголовок}
    rsSeekStream(FZeroPosition);
    rsWriteStream(FHeaderRec^, FHeaderRec^.hrHeaderLen);
    {задать значение поля длины записи}
    FRecordLen := aRecordLen;
end;

procedure TtdRecordStream.rsReadHeaderRec;
var
    StreamSize : longint;
    TempHeaderRec : TtdRSHeaderRec;
begin
    {если размер потока меньше размера служебного заголовка, это неверный поток}
    StreamSize := FStream.Size - FZeroPosition;
    if (StreamSize < sizeof(TtdRSHeaderRec)) then
        rsError(tdeRSNoHeaderRec, 'rsReadHeaderRec', 0);
    {считать служебный заголовок}
    rsSeekStream(FZeroPosition);
    rsReadStream(TempHeaderRec, sizeof(TtdRSHeaderRec));

```



```

{первая санитарная проверка: сигнатура и счетчик/емкость}
with TempHeaderRec do begin
  if (hrSignature<>cRSSignature) or (hrCount > hrCapacity) then
    rsError(tdeRSBadHeaderRec, 'rsReadHeaderRec', 0);
end;
{выделить память под реальный служебный заголовок,
скопировать уже считанные данные}
FHeaderRec := AllocMem(TempHeaderRec.hrHeaderLen);
Move(TempHeaderRec, FHeaderRec^, TempHeaderRec.hrHeaderLen);
{вторая санитарная проверка: проверка данных записи}
with FHeaderRec^ do begin
  FRecordLen4 := hrRecordLen + 4; {for rsCalcRecordOffset}
  if (StreamSize<>rsCalcRecordOffset(hrCapacity)) then
    rsError(tdeRSBadHeaderRec, 'rsReadHeaderRec', 0);
  {установить значения полей класса}
  FCount := hrCount;
  FCapacity := hrCapacity;
  FRecordLen := hrRecordLen;
end;
end;
function TtdRecordStream.rsCalcRecordOffset(aIndex : longint) : longint;
begin
  Result := FZeroPosition + FHeaderRec^.hrHeaderLen + (aIndex * FRecordLen4);
end;

```

Приведенный метод создания служебного заголовка вызывается только в случае, когда поток пуст. Принцип его работы очень прост. Сначала служебный заголовок создается в памяти, а затем записывается в поток. Если длина записи больше, чем нормальный размер служебного заголовка, его размер увеличивает до размера записи. В служебном заголовке содержится семь полей: поле сигнатуры, которое может использоваться для контроля при считывании записи; номер версии служебного заголовка (это позволит в будущем добавлять в заголовок новые поля и сохранять совместимость версий); длина служебного заголовка; длина записи; емкость потока (т.е. количество записей как активных, так и удаленных, которые в данный момент находятся в потоке); количество активных записей; и, наконец, порядковый номер первой удаленной записи (здесь значение этого поля устанавливается равным **cEndOfDetectedChain** или -2).

Метод считывания служебного заголовка должен содержать определенную проверку, которая будет гарантировать, что данный заголовок является действительным. Для этого сначала выполняется проверка сигнатуры, затем проверка того, что количество активных записей меньше или равно емкости потока и имеет ли поток достаточный объем для объявленной емкости. Если все проверки проходят успешно, считается, что служебный блок содержит корректные данные, и поля класса обновляются значениями, считанными из потока.

Метод **rsCalcRecordOffset** просто вычисляет смещение записи, порядковый номер которой передан ему во входном параметре. При этом учитывается начальное положение потока и размер служебного заголовка.

Листинг 2.22. Добавление новой записи в постоянный массив

```
function TtdRecordStream.Add(var aRecord) : longint;
begin
  {если цепочка удаленных записей пуста, в поток добавляется новая запись}
  if (FHeaderRec^.hr1stDelRec = cEndOfDeletedChain) then begin
    Result := FCapacity;
    inc(FCapacity);
    inc(FHeaderRec^.hrCapacity);
  end
  {в противном случае используется первая удаленная запись,
  обновляется значение поля удаленной записи в служебном заголовке для
  указания на следующую удаленную запись}
  else begin
    Result := FHeaderRec^.hr1stDelRec;
    rsSeekStream(rsCalcRecordOffset(FHeaderRec^.hr1stDelRec));
    rsReadStream(FHeaderRec^.hr1stDelRec, sizeof(longint));
  end;
  {определить смещение записи и сохранить новую запись}
  rsSeekStream(rsCalcRecordOffset(Result));
  PLongint(FRecord)^ := cActiveRecord;
  Move(aRecord, FRecord^[sizeof(longint)], FRecordLen);
  rsWriteStream(FRecord^, FRecordLen4);
  {количество записей увеличилось на единицу}
  inc(FCount);
  inc(FHeaderRec^.hrCount);
  {обновить служебный заголовок}
  rsSeekStream(FZeroPosition);
  rsWriteStream(FHeaderRec^, sizeof(TtdRSHeaderRec));
end;
```

Если цепочка удаленных записей не пуста, определить первую удаленную запись (именно поверх нее будет сохраняться новая запись). Мы считываем флаг удаления для этой записи и обновляем поле первой удаленной записи служебного заголовка. Затем мы определяем положение начала удаленной записи, устанавливаем значение флага удаления равным **cActiveRecord** (-1) и сохраняем запись, переданную методу во входном параметре.

При считывании и сохранении записей необходимо учитывать, удалена ли требуемая запись. Записи идентифицируются по их порядковым номерам.

Листинг 2.23. Чтение и обновление записи в постоянном массиве

```

procedure TtdRecordStream.Read(aIndex : longint;
                                var aRecord;
                                var aIsDeleted : boolean);
begin
    {проверить, действителен ли порядковый номер записи}
    if (aIndex < 0) or (aIndex >= Capacity) then
        rsError(tdeRSOutOfBounds, 'Read', aIndex);
    {определить смещение записи и считать ее}
    rsSeekStream(rsCalcRecordOffset(aIndex));
    rsReadStream(FRecord^, FRecordLen4);
    if (PLongint(FRecord)^ = cActiveRecord) then begin
        aIsDeleted := false;
        Move(FRecord^[sizeof(longint)], aRecord, FRecordLen);
    end
    else begin
        aIsDeleted := true;
        FillChar(aRecord, FRecordLen, 0);
    end;
end;

procedure TtdRecordStream.Write(aIndex : longint;
                                var aRecord);

var
    DeletedFlag : longint;
begin
    {проверить, действителен ли порядковый номер записи}
    if (aIndex < 0) or (aIndex >= Capacity) then
        rsError(tdeIndexOutOfBounds, 'Write', aIndex);
    {проверить, что запись не была удалена}
    rsSeekStream(rsCalcRecordOffset(aIndex));
    rsReadStream(DeletedFlag, sizeof(longint));
    if (DeletedFlag <> cActiveRecord) then
        rsError(tdeRSRecIsDeleted, 'Write', aIndex);
    {сохранить запись}
    rsWriteStream(aRecord, FRecordLen);
end;

```

Метод **Read** возвращает флаг, который показывает, была ли удалена запись. Если запись не удалена, буфер записи, переданный во входном параметре, заполняется записью, считанной из потока. Код просто в один прием считывает всю запись и ее флаг удаления и действует в соответствии со значением флага.

Метод **Write**, прежде всего, проверяет, была ли удалена требуемая запись. Если запись удалена, она недоступна для изменения, вследствие чего возникает исключение. В противном случае в поток помещается новое значение записи.

И последний метод, связанный с обработкой записей, – это метод **Delete**.

Листинг 2.24. Чтение и обновление записи в постоянном массиве

```

procedure TtdRecordStream.Delete(aIndex : longint);
var
    DeletedFlag : longint;
begin
    {проверить, действителен ли порядковый номер записи}
    if (aIndex < 0) or (aIndex >= Capacity) then
        rsError(tdeRSOutOfBounds, 'Delete', aIndex);
    {проверить, что запись не была удалена}
    rsSeekStream(rsCalcRecordOffset(aIndex));
    rsReadStream(DeletedFlag, sizeof(longint));
    if (DeletedFlag <> cActiveRecord) then
        rsError(tdeRSAlreadyDeleted, 'Delete', aIndex);
    {записать порядковый номер первой удаленной записи в первые
     4 байта удаляемой записи}
    rsSeekStream(rsCalcRecordOffset(aIndex));
    rsWriteStream(FHeaderRec^.hr1stDelRec, sizeof(longint));
    {обновить значение поля первой удаленной записи служебного
     заголовка, чтобы оно указывало на удаляемую запись}
    FHeaderRec^.hr1stDelRec := aIndex;
    {количество записей уменьшилось на единицу}
    dec(FCount);
    dec(FHeaderRec^.hrCount);
    {обновить служебный заголовок}
    rsSeekStream(FZeroPosition);
    rsWriteStream(FHeaderRec^, sizeof(TtdRSHeaderRec));
end;

```

Метод **Delete**, прежде всего, проверяет, была ли удалена требуемая запись. Если запись была удалена, метод вызывает ошибку. Если все в порядке, текущее значение поля первой удаленной записи служебного заголовка копируется во флаг удаления записи. Затем значение поля первой удаленной записи устанавливается равным порядковому номеру удаляемой записи, количество активных записей в массиве уменьшается на единицу и обновляется служебный заголовок в потоке.

Метод **Clear** аналогичен **Delete**, но он предназначен для удаления всех активных записей постоянного массива.

Листинг 2.25. Очистка содержимого постоянного массива

```

procedure TtdRecordStream.Clear;
var
    Inx : longint;
    DeletedFlag : longint;
begin
    {выполнить цикл по всем записям и объединить их в одну цепочку удаленных
     записей}
    for Inx := 0 to pred(FCapacity) do begin
        rsSeekStream(rsCalcRecordOffset(Inx));

```

```

rsReadStream(DeletedFlag, sizeof(longint));
if (DeletedFlag = cActiveRecord) then begin
    {записать порядковый номер первой удаленной записи в первые 4 байта
    удаляемой записи}
    rsSeekStream(rsCalcRecordOffset(Inx));
    rsWriteStream(FHeaderRec^.hrlstDelRec, sizeof(longint));
    {обновить значение поля первой удаленной записи служебного заголовка,
    чтобы оно указывало на удаляемую запись}
    FHeaderRec^.hrlstDelRec := Inx;
end;
end;
{записей нет}
FCount := 0;
FHeaderRec^.hrCount := 0;
{обновить служебный заголовок}
rsSeekStream(FZeroPosition);
rsWriteStream(FHeaderRec^, sizeof(TtdRSHeaderRec));
end;

```

Этот метод выполняет цикл по всем записям массива и, если запись активна, удаляет ее в соответствии с алгоритмом, используемым в методе **Delete**.

Класс **TtdRecordStream** позволяет также в один прием увеличивать емкость потока на несколько записей, а не добавлять записи по одной с помощью метода **Add**. Такая возможность позволяет зарезервировать место под постоянный массив, если заранее известно количество записей, которое будет в нем храниться. Запись свойства **Capacity** осуществляется через метод **rsSetCapacity**.

Листинг 2.26. Задание емкости постоянного массива

```

procedure TtdRecordStream.rsSetCapacity(aCapacity : longint);
var
    Inx : longint;
begin
    {допускается только увеличение емкости}
    if (aCapacity > FCapacity) then begin
        {заполнить текущую запись нулями}
        FillChar(FRecord^, FRecordLen4, 0);
        {найти конец файла}
        rsSeekStream(rsCalcRecordOffset(FCapacity));
        {создать дополнительные записи и внести их в цепочку удаленных записей}
        for Inx := FCapacity to pred(aCapacity) do begin
            PLongint(FRecord)^ := FHeaderRec^.hrlstDelRec;
            rsWriteStream(FRecord^, FRecordLen4);
            FHeaderRec^.hrlstDelRec := Inx;
        end;
        {сохранить новую емкость}
        FCapacity := aCapacity;
        FHeaderRec^.hrCapacity := aCapacity;
    end;

```



```

    {обновить служебный заголовок}
    rsSeekStream(FZeroPosition);
    rsWriteStream(FHeaderRec^, sizeof(TtdRSHeaderRec));
end;
end;

```

Как видно из приведенного кода, метод **rsSetCapacity** добавляет в конец потока пустые записи и вносит их в цепочку удаленных записей. После этого обновляются поля служебного заголовка, и в массиве появляется несколько удаленных записей, которые можно заполнить с помощью метода **Add**.

Последние методы, которые мы рассмотрим, будут очень простыми. Это низкоуровневые методы, предназначенные для считывания из потока, записи в поток и поиска в потоке. Кроме того, в каждом из них имеется блок проверки результата.

Листинг 2.27. Низкоуровневые методы доступа к потоку

```

procedure TtdRecordStream.rsReadStream(var aBuffer;
                                         aBufLen : integer);

var
    BytesRead : longint;
begin
    BytesRead := FStream.Read(aBuffer, aBufLen);
    if (BytesRead <> aBufLen) then
        rsError(tdeRSReadError, 'rsReadStream', aBufLen);
end;

procedure TtdRecordStream.rsSeekStream(aOffset : longint);
var
    NewOffset : longint;
begin
    NewOffset := FStream.Seek(aOffset, soFromBeginning);
    if (NewOffset <> aOffset) then
        rsError(tdeRSSeekError, 'rsSeekStream', aOffset);
end;

procedure TtdRecordStream.rsWriteStream(var aBuffer;
                                         aBufLen : integer);

var
    BytesWritten : longint;
begin
    BytesWritten := FStream.Write(aBuffer, aBufLen);
    if (BytesWritten <> aBufLen) then
        rsError(tdeRSWriteError, 'rsWriteStream', aBufLen);
    Flush;
end;

```

Как видите, если результат выполнения одного из методов не соответствует ожидаемому, методы вызывают исключения.

Существует еще один метод, о котором мы не говорили, – **rsWriteStream**. Фактически это метод **Flush** – виртуальный метод, предназначенный для сброса содер-

жащихся в потоке данных на связанное с потоком устройство (например, диск). Его реализация для нашего класса представляет собой пустую подпрограмму, поскольку мы не знаем, как сбросить данные из стандартного потока **TStream**. Он существует только для того, чтобы быть перекрытым в дочерних классах, которые имеют дело с потоком, связанным с диском, например, файловым потоком.

Листинг 2.28. Реализация постоянных массивов с помощью файлового потока

```

constructor TtdRecordFile.Create(const aFileName : string;
                                aMode : word;
                                aRecordLength : integer);

begin
    FStream := TFileStream.Create(aFileName, aMode);
    inherited Create(FStream, aRecordLength);
    FFileName := aFileName;
    FMode := aMode;
end;

destructor TtdRecordFile.Destroy;
begin
    inherited Destroy;
    FStream.Free;
end;

procedure TtdRecordFile.Flush;
{ $IFDEF Delphi1}
var
    DosError : word;
    Handle : THandle;
begin
    Handle := FStream.Handle;
    asm
        mov ah, $68
        mov bx, Handle
        call DOS3Call
        jc @@Error
        xor ax, ax
    @@Error:
        mov DosError, ax
    end;
    if (DosError <> 0) then
        rsError(tdeRSFlushError, 'Flush', DosError)
end;
{ $ENDIF}
{ $IFDEF Delphi2Plus}
begin
    if not FlushFileBuffers(FStream.Handle) then
        rsError(tdeRSFlushError, 'Flush', GetLastError)
end;
{ $ENDIF}

```

В приведенном коде присутствует перекрытый метод **Flush**, который сбрасывает данные в дескриптор, связанный с файловым потоком, содержащим постоянный массив. Реализации для Delphi 1 и для 32-битных версий будут отличаться, поскольку процесс сброса данных в дескриптор в этих версиях различен.

Полный код класса **TtdRecordStream** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDRecFil.pas**.

Резюме

Эта глава была посвящена массивам – одной из фундаментальных структур данных. Были описаны их достоинства (доступ к отдельным элементам составляет $O(1)$, поддерживается локальность ссылок) и недостатки (вставка и удаление элементов относятся к операциям класса $O(n)$). Приведена реализация класса массива **TtdRecordList**. Затем был подробно рассмотрен стандартный класс **TList** и его простой дочерний класс **TtdObjectList**.

Кроме того, мы познакомились с реализацией постоянных массивов в форме потока записей. Был приведен пример реализации класса постоянных массивов, **TtdRecordStream**, который позволяет выполнять чтение, запись и удаление отдельных записей.



Глава 3

Связные списки, стеки и очереди

Как и массивы, связные списки представляют собой универсальную структуру данных, широко используемую многими программистами. Однако, в отличие от массивов, связные списки не входят в состав стандартного языка Object Pascal. Тем не менее, в Object Pascal создать связный список достаточно просто. Все что для этого нужно – наличие в составе языка указателя, хотя фактически могут использоваться и классы или объекты.

На основе связных списков можно легко организовать стеки и очереди – еще две простые, но эффективные структуры данных. Несмотря на то что они, на первый взгляд, не имеют ничего общего со связными списками, их можно написать на базе односвязных списков. И, как мы увидим чуть позже, иногда удобнее реализовать стеки и очереди на базе массивов, а не связных списков.

Начнем наше рассмотрение со связного списка и операций, которые такой список должен поддерживать.

Односвязные списки

По своей сути *связный список (linked list)* представляет собой цепочку элементов или объектов с некоторыми описаниями (обычно называемых узлами). При этом каждый элемент содержит указатель, указывающий на следующий элемент в списке. Такая структура данных называется *односвязным списком (singly linked list)* – каждый элемент имеет только одну ссылку или указатель на следующий элемент. Сам список начинается с первого узла, от которого путем последовательных переходов по ссылкам можно обойти все остальные узлы. Обратите внимание, что определение связного списка отличается от определения массива, для которо-



Рисунок 3.1. Односвязный список

го следующий элемент находится в памяти рядом с предыдущим. В связном списке элементы могут быть разбросаны по разным местам памяти, а их порядок определяется ссылками.

А каким образом помечается конец списка? Самый простой способ – установить указатель ссылки в последнем элементе списка равным **nil**. Это будет означать, что следующий элемент отсутствует. Второй способ – ввести специальный узел, называемый конечным узлом, и установить так, чтобы ссылка последнего узла указывала на этот узел. И третий способ – установить так, чтобы ссылка последнего узла указывала на первый элемент. В этом случае мы получим круговой связный список.

А теперь рассмотрим, чем же связный список отличается от массива. Первое, что нужно отметить, – размер связного списка можно не устанавливать. Для массива нам всегда было нужно заранее знать, сколько элементов будет в нем храниться (чтобы можно было статически распределить непрерывный участок памяти) или разработать некоторую схему расширения массива (или его сокращения), чтобы массив мог разместить большее (или меньшее) количество элементов. В связном списке каждый узел является отдельным элементом. И в простых случаях распределение памяти под каждый узел выполняется отдельно. При необходимости добавления в список нового элемента под него распределяется память, а затем элемент просто на него устанавливается ссылка из списка. При удалении узла нужно всего лишь удалить ссылки на него и освободить занимаемую им память.

Хорошо. Если связный список настолько удобен, почему бы его не использовать вместо массива? В чем состоят его недостатки? Первый, хотя и незначительный, состоит в том, что каждый элемент связного списка должен содержать указатель на следующий элемент. Таким образом, чтобы вставить элемент в список, его реальный размер необходимо увеличить на размер указателя (в настоящее время это 4 байта).

Хуже то, что память под каждый узел распределяется отдельно. Сравним эту ситуацию с аналогичной ситуацией для массива. Распределение памяти под n элементов массива, фактически, представляет собой операцию класса $O(1)$: все элементы должны находиться в одном непрерывном блоке памяти, поэтому одновременно распределяется целый блок. (Нужно помнить, что память для элементов массивов не обязательно должна распределяться из кучи. Массивы могут представлять собой, например, локальные переменные в стеке.) Для связного списка память под узлы распределяется отдельно, следовательно, это операция класса $O(n)$. Даже если не учитывать быстродействие, подобное поведение может привести к фрагментации кучи.

Самым большим недостатком связного списка является получение доступа к некоторому элементу n . В массиве доступ к n -ному элементу требует проведения простых арифметических вычислений, поскольку все элементы содержатся в одном непрерывном блоке памяти. С другой стороны, в списке получение доступа к элементу n требует прохождения по ссылкам от первого элемента до n -ного. Другого метода доступа не существует, мы всегда должны следовать по ссылкам. (Обратите внимание, что можно применить определенные хитрости, например, хранить элемент и его позицию в рамках списка в кэш-памяти. В таком случае можно определять целесообразность начала прохождения списка с его первого элемента или с элемента, хранящегося в кэш-памяти.)

Узлы связного списка

Перед началом описания операций со связным списком давайте рассмотрим, как каждый узел списка будет представляться в памяти. Знание структуры узла позволит нам более детально рассматривать основные операции со связными списками. Структура узла списка, не использующего классы и объекты, выглядит следующим образом:

```
type
  PSimpleNode = ^TSimpleNode;
  TSimpleNode = record
    Next : PSimpleNode;
    Data : SomeDataType;
  end;
```

Тип **PSimpleNode** представляет собой указатель на запись **TSimpleNode**, поле **Next** которой содержит ссылку на точно такой же узел, а поле **Data** – сами данные. В приведенном примере тип данных узла задан как **SomeDataType**. Для перехода по ссылке нужно написать примерно следующий код:

```
var
  NextNode, CurrentNode : PSimpleNode;
begin
  ...
  NextNode := CurrentNode^.Next;
```

Создание односвязного списка

Это тривиальная задача. В самом простом случае первый узел в связном списке описывает весь список. Первый узел иногда называют *головой* списка.

```
var
  MyLinkedList : PSimpleNode;
```

Если **MyLinkedList** содержит **nil**, списка еще нет. Таким образом, это начальное значение связного списка.

```
{ инициализация связного списка }
MyLinkedList := nil;
```

Вставка и удаление элементов в односвязном списке

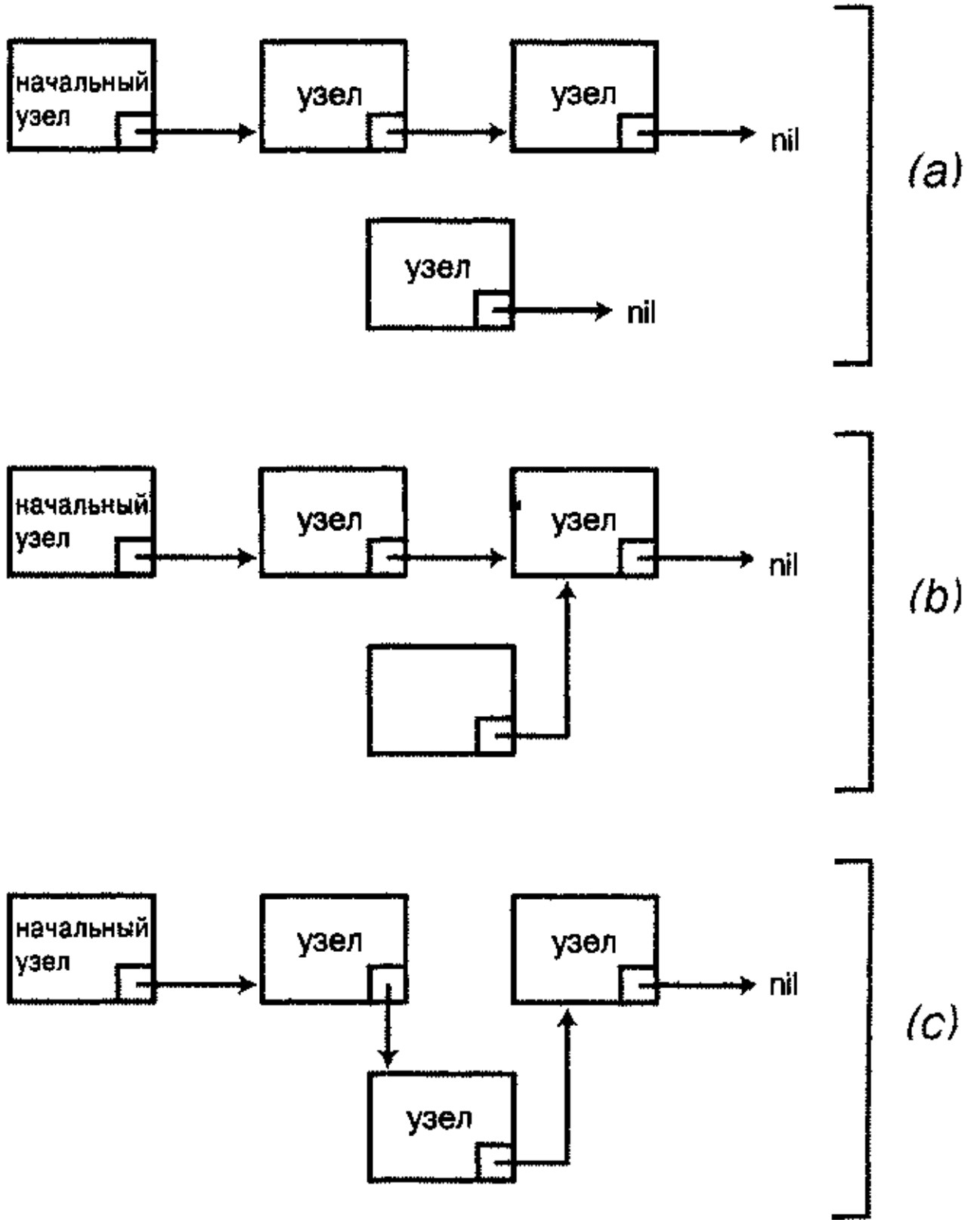
А каким образом можно вставить новый элемент в связный список? Или удалить? Оказывается, что для выполнения этих операций требуется выполнить небольшую работу с указателями.

Для односвязного списка существует только один вариант вставки – после заданного элемента списка. Нужно установить так, чтобы указатель **Next** нашего нового узла указывал на узел после заданного, а указатель **Next** заданного узла – на наш новый узел. В коде это выглядит следующим образом:

```
var
  GivenNode, NewNode : PSimpleNode;
begin
  ...
  New(NewNode);
  ..задать значение поля Data..
  NewNode^.Next := GivenNode^.Next;
  GivenNode^.Next := NewNode;
```

Аналогично, для удаления простейшим вариантом является удаление элемента, находящегося после заданного узла. В этом случае мы устанавливаем, чтобы ука-

Рисунок 3.2.
Вставка нового узла
в односвязный список



затель **Next** заданного узла указывал на узел, расположенный после удаляемого. После этого удаляемый узел уже выделен из списка и может быть освобожден. В коде это выглядит следующим образом:

```
var
  GivenNode, NodeToGo : PSimpleNode;
begin
  ...
  NodeToGo := GivenNode^.Next;
  GivenNode^.Next := NodeToGo^.Next;
  Dispose(NodeToGo);
```

Тем не менее, для обеих операций существует специальный случай: вставка перед первым элементом списка (т.е. новый элемент становится первым) и удаление первого элемента списка (т.е. первым становится другой элемент). Поскольку в наших рассуждениях первый элемент считается определяющим узлом всего списка, код для этих случаев нужно написать отдельно. Вставка перед первым узлом будет выглядеть следующим образом:

```
var
  GivenNode, NewNode : PSimpleNode;
begin
  ...
  New(NewNode);
  ..задать значение поля Data..
  NewNode^.Next := MyLinkedList;
  MyLinkedList := NewNode;
```

а удаление будет выглядеть так:

```
var
  GivenNode, NodeToGo : PSimpleNode;
begin
  ...
  NodeToGo := GivenNode^.Next;
  MyLinkedList := NodeToGo^.Next;
  Dispose(NodeToGo);
```

Обратите внимание, что код вставки элемента будет работать даже в случае, когда исходный список пуст, т.е. содержит **nil**, а код удаления элемента правильно установит содержимое связного списка в случае удаления из него последнего узла.

Прохождение связного списка

Прохождение связного списка также не представляет никаких трудностей. Фактически мы переходим от узла к узлу по указателям **Next** до достижения указателя **nil**, который свидетельствует об окончании списка.

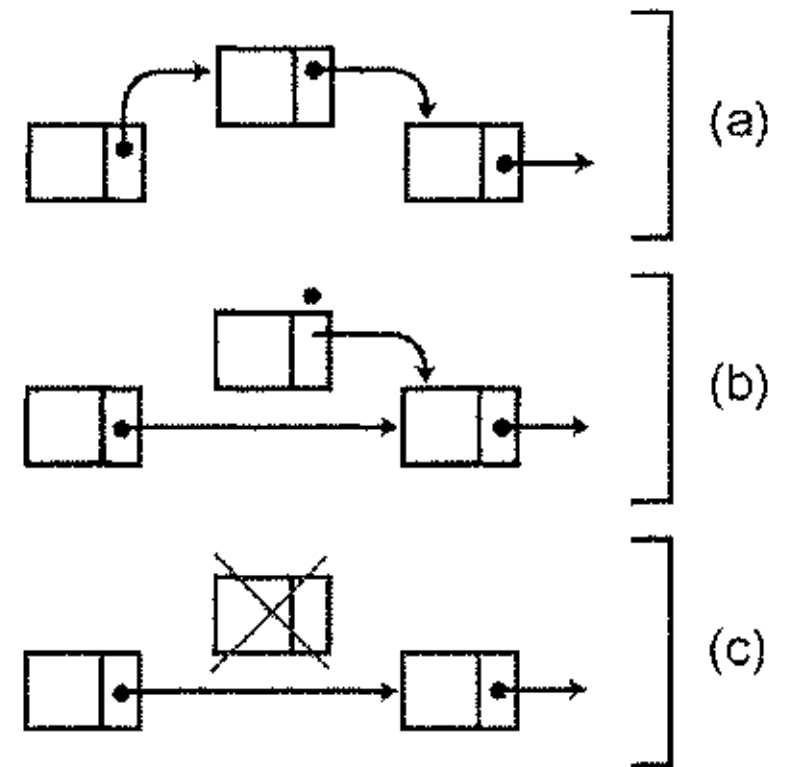


Рисунок 3.3. Удаление узла из односвязного списка

```
var
  FirstNode, TempNode : PSimpleNode;
begin
  . . .
  TempNode := FirstNode;
  while TempNode <> nil do begin
    Process(TempNode^.Data);
    TempNode := TempNode^.Next;
  end;
```

В этом простом цикле процедура **Process** (определенная в другом месте) выполняет обработку поля **Data** переданного ей узла. Очистка связного списка требует небольшого изменения алгоритма, чтобы гарантировать, что мы не ссылаемся на поле **Next** после освобождения узла (довольно-таки частая ошибка).

```
var
  MyLinkedList, TempNode, NodeToGo : PSimpleNode;
begin
  NodeToGo := MyLinkedList;
  while NodeToGo <> nil do begin
    TempNode := NodeToGo^.Next;
    Dispose(NodeToGo);
    NodeToGo := TempNode;
  end;
  MyLinkedList := nil;
```

Теперь, когда мы научились проходить по узлам связного списка, давайте вернемся к вопросу, который, наверное, появился у вас пару абзацев назад. А что если нам нужно вставить узел перед заданным узлом? Как это сделать? Единственным решением такой задачи для односвязного списка является прохождение списка и поиск узла, перед которым мы должны вставить новый узел. При прохождении будут использоваться две переменных: одна будет указывать на текущий, а вторая на предыдущий узел (родительский узел, если можно так сказать). Когда будет найден заданный узел, у нас будет указатель на предыдущий узел, что позволит использовать алгоритм вставки после заданного узла. В коде это выглядит следующим образом:

```
var
  FirstNode, GivenNode,
  TempNode, ParentNode : PSimpleNode;
begin
  ParentNode := nil;
  TempNode := FirstNode;
  while TempNode <> GivenNode do begin
    ParentNode := TempNode;
    TempNode := ParentNode^.Next;
  end;
```

```

if TempNode = GivenNode then begin
  if (ParentNode = nil) then begin
    NewNode^.Next := FirstNode;
    FirstNode := NewNode;
  end
  else begin
    NewNode^.Next := ParentNode^.Next;
    ParentNode^.Next := NewNode;
  end;
end;

```

Обратите внимание на специальный код для случая вставки нового узла перед первым узлом (в этом случае родительский узел **nil**). Код для вставки перед заданным узлом медленнее кода вставки после заданного узла, поскольку он требует прохождения списка с целью обнаружения родительского узла заданного узла. В общем случае, при необходимости вставки нового узла перед заданным мы будем использовать двухсвязный список, который будет подробно рассмотрен немного ниже.

Соображения по поводу эффективности

Если бы это было все, что можно сказать о связных списках, то глава оказалась бы очень короткой. До сих пор была представлена только реализация класса, инкапсулирующего односвязный список. Но перед написанием класса связного списка нужно рассмотреть еще несколько вопросов, касающихся, в частности, эффективности.

Использование начального узла

Еще раз просмотрите код вставки и удаления элемента связного списка. Не кажется ли вам неудобным наличие двух случаев для обеих операций? Отдельные специальные случаи нужны для обработки вставки и удаления первого узла – операция, которая, возможно, будет выполняться не очень часто. Может быть, существует другой способ? Другой способ действительно есть, он предусматривает использование фиктивного начального узла. Фиктивный начальный узел – это узел, который нужен только в качестве заполнителя, в нем не будут храниться данные. Первым реальным узлом будет тот, на который указывает указатель **Next** фиктивного узла. Связный список, как и раньше, заканчивается узлом, указатель **Next** которого равен **nil**. При создании такого списка его нужно правильно инициализировать, выделив память под начальный узел и установив его указатель **Next** равным **nil**.

```

var
  HeadNode : PSimpleNode;
begin
  ...
  New(HeadNode);
  HeadNode^.Next := nil;

```


После этой небольшой подготовительной части все вставки и удаления можно будет выполнять с помощью операций "вставить после" и "удалить после". Операция "вставить после первого узла" сводится к вставке нового элемента после фиктивного узла, а операция "удалить первый узел" превращается в удаление элемента после начального узла. За счет использования фиктивного начального узла нам удалось избежать специальных случаев.

Конечно, введение фиктивного начального узла усложнило реализацию класса: теперь при создании нового связного списка нам нужно распределить и инициализировать дополнительный узел, а при удалении списка – уничтожить этот узел.

Использование диспетчера узлов

Перед написанием класса связного списка нужно рассмотреть еще один вопрос. Мы начали с того, что объявили тип узла как запись (тип **TSimpleNode**), в которой хранятся (1) данные и (2) указатель на следующий узел списка. Второе поле записи удалить нельзя – оно требуется для организации связного списка. Но первое зависит от приложения или конкретного использования списка в данный момент. Фактически поле данных может представлять собой не одно, а несколько полей в отдельной записи или даже объект. Очень сложно написать общий класс связного списка, если неизвестен тип данных, который будет в нем содержаться.

Однако решение этой задачи существует, даже целых два. Первое – объявить класс родительского узла, который бы содержал только указатель **Next**, а пользователь сам в дочернем классе объявит элементы, содержащие данные узла. В таком случае реализация базового класса будет распределять и освобождать память для узла, поскольку все, что нужно для организации связного списка – это выделенные узлы, указателями **Next** которых можно манипулировать. Такое решение одновременно является как элегантным, так и неэлегантным. Оно соответствует парадигме объектно-ориентированного программирования, но для хранения данных приходится объявлять дочерний класс (а что если элементы, которые необходимо поместить в список, являются экземплярами класса, над которым вы не имеете контроля?).

Второе решение, которое можно считать более удачным, – представить данные в форме нетипизированного указателя. (Для этого имеются все предпосылки – в Delphi подобным образом работает стандартный класс **TList**.) При добавлении элемента в связный список классу списка передается только значение указателя (скажем, указателя на данные или объект в куче), а все остальное делает сам список: распределяет память под узел, устанавливает значения данных и манипулирует ссылками. Это решение обходит проблему незнания типа данных, поскольку пользователь класса может не беспокоиться об указателях **Next**, не должен распределять для них память, создавать специальные дочерние классы и т.д.

Но это второе решение имеет свой недостаток. Размер используемых классом узлов всегда будет равен 8 байтам – по 4 байта для указателя и данных.

Обратите внимание, что в приведенных выше рассуждениях считается, что размер указателей составляет 4 байта. Можно предположить, что последующие версии Delphi будут написаны для 64-разрядных операционных систем. В таком случае длина указателей будет составлять 8 байт. Поэтому не основывайтесь на предположении, что длина указателя всегда 4 байта, пользуйтесь функцией `sizeof(pointer)`. В будущем это существенно упростит перенос кода в новые версии Delphi. Тем не менее, в настоящей главе считается, что длина указателя составляет 4 байта, даже несмотря на то, что в коде используется `sizeof(pointer)`. Такое допущение позволяет упростить выкладки, поскольку можно просто сказать "8 байт", а не "удвоенное значение `sizeof(pointer)`".

Что дает нам постоянный размер узла? При необходимости записи данных в связный список нужно предварительно распределить память под узел. Для этого пришлось бы использовать очень сложный диспетчер кучи Delphi, всего лишь, чтобы распределить 8 байт памяти. Диспетчер кучи содержит код, который может манипулировать кусками памяти, а также распределять и освобождать память любого размера. Все операции выполняются быстро и эффективно. Но мы знаем, что будут использоваться только 8-байтные блоки, и нам нужны только такие блоки. Можно ли, учитывая постоянство размеров блоков, увеличить скорость распределения памяти для новых узлов и освобождения памяти, занимаемой удаляемыми узлами? Конечно же, ответ "да". Мы с помощью диспетчера кучи одновременно распределяем память для нескольких узлов, а затем используем ее по мере необходимости. Таким образом, память распределяется не для одного узла, а, скажем, для 100 узлов. Если потребуется большее количество, будет выделен еще один блок из 100 узлов.

А вот теперь начинается самое интересное. Массивы узлов хранятся во внутреннем связном списке, а узлы, которые берутся из этих массивов, попадают в свободный список, который также представляет собой связный список. Таким образом, для увеличения эффективности работы класс связного списка будет использовать массивы и собственно связные списки.

Что в предыдущем абзаце понимается под понятием "свободный список"? Это цепная конструкция в программировании. У нас имеется набор некоторых элементов, которые "распределяются" и "освобождаются". При освобождении элемента он, скорее всего, в какой-то момент времени будет использоваться повторно. Поэтому вместо возвращения занимаемой элементом памяти поддерживается *свободный список*, т.е. список свободных элементов. Диспетчер кучи Delphi использует свободный список освобожденных блоков памяти различного размера. Многие базы данных поддерживают скрытый свободный список удаленных записей, которые можно использовать повторно. Массив записей, описанный в главе 2, использует цепочку удаленных записей, которая является ни чем иным, как другим названием свободного списка. При необходимости распределения памяти под элемент приложение обращается к свободному списку и выбирает один из свободных элементов.

Давайте разработаем *диспетчер распределения узлов*. Он будет содержать свободный список узлов, который вначале устанавливается равным **nil**, что означает, что список пуст. При распределении узла диспетчер будет просматривать свой свободный список. Если он пуст (равен **nil**), диспетчер распределяет большой блок памяти, обычно называемый страницей, при помощи диспетчера кучи Delphi. Затем страница разделяется на отдельные куски с размером, равным размеру узла, и куски записываются в свободный список. После этого узел извлекается из списка и передается пользователю. При освобождении узла он записывается в свободный список. Под "записью" здесь понимается вставка узла в начало списка, а под "извлечением" – считывание узла с начала списка.

Листинг 3.1. Класс TtdNodeManager

```
TtdNodeManager = class
  private
    FNodeSize : cardinal;
    FFreeList : pointer;
    FNodesPerPage : cardinal;
    FPageHead : pointer;
    FPageSize : cardinal;
  protected
    procedure nmAllocNewPage;
  public
    constructor Create(aNodeSize : cardinal);
    destructor Destroy; override;

    function AllocNode : pointer;
    procedure FreeNode(aNode : pointer);
end;
```

Как видно из приведенного кода, реализация интерфейса класса достаточно проста. Класс позволяет создавать и уничтожать экземпляры и распределять и освобождать узлы. Конструктор **Create** в качестве единственного входного параметра принимает размер узла, а затем на его основании вычисляет несколько значений: количество узлов на странице и размер страницы. Класс пытается распределять страницы размером 1024 байта. Но если размер узла настолько велик, что на одной такой странице может поместиться только один узел, размер страницы выбирается равным размеру узла. Для увеличения эффективности размер узла округляется до ближайшей границы 4 байт (фактически округление выполняется до ближайшего значения **sizeof(pointer)**).

Листинг 3.2. Конструктор TtdNodeManager.Create

```
constructor TtdNodeManager.Create(aNodeSize : cardinal);
begin
  inherited Create;
```

```

{сохранить размер узла, округленный до ближайших 4 байт}
if (aNodeSize <= sizeof(pointer)) then
  aNodeSize := sizeof(pointer)
else
  aNodeSize := ((aNodeSize + 3) shr 2) shl 2;
FNodeSize := aNodeSize;
{вычислить размер страницы (по умолчанию используется 1024 байта) и
 количество узлов на странице; если размер страницы по умолчанию недостато-
 чен для размещения двух или большего количества узлов, выбрать размер стра-
 ницы равным размеру узла}
FNodesPerPage := (PageSize - sizeof(pointer)) div aNodeSize;
if (FNodesPerPage > 1) then
  FPageSize := PageSize
else begin
  FNodesPerPage := 1;
  FPagesize := aNodeSize + sizeof(pointer);
end;
end;
end;

```

Код метода **AllocNode** очень прост. Если свободный список пуст, вызывается метод **nmAllocNewPage**, который распределяет новую страницу и вносит все ее узлы в свободный список. Если свободный список не пуст, из него выбирается первый узел (фактически с помощью процедуры удаления первого узла).

Листинг 3.3. Распределение памяти для узла в классе **TtdNodeManager**

```

function TtdNodeManager.AllocNode : pointer;
begin
  {если свободный список пуст, распределить память для новой страницы; это
   приведет к заполнению свободного списка}
  if (FFreeList = nil) then
    nmAllocNewPage;
  {вернуть первый элемент свободного списка}
  Result := FFreeList;
  FFreeList := PGenericNode(FFreeList)^.gnNext;
end;

```

Тип **PGenericNode** представляет собой запись с одним полем – полем для ссылки **gnNext**. Этот тип и преобразование типов в коде позволяет работать с узлами в списке свободных узлов на основании общего алгоритма – нечто похожее на тип **TSimpleNode**, который мы рассматривали ранее. Обратите внимание, конструктор гарантирует, что размер узлов, отслеживаемых диспетчером узлов, составляет, по крайней мере, 4 байта, т.е. размер указателя.

Следующий метод – **FreeNode** – ничуть не сложнее предыдущего. Он просто вставляет новый узел в начало свободного списка (используется код вставки перед первым узлом).

Листинг 3.4. Освобождение узла в классе TtdNodeManager

```

procedure TtdNodeManager.FreeNode(aNode : pointer);
begin
  {вставить узел (если он не nil) в начало свободного списка}
  if (aNode <> nil) then begin
    PGenericNode(aNode)^.gnNext := FFreeList;
    FFreeList := aNode;
  end;
end;

```

Следующий метод, заслуживающий внимания, – **nmAllocNewPage**. Этот метод предназначен для распределения памяти объемом **FPageSize**, вычисленным в конструкторе **Create**, для новой страницы. Каждая страница содержит указатель и узлы **FNodesPerPage**. Указатель используется для создания связанного списка страниц (именно поэтому конструктор учитывает в своих вычислениях значение **sizeof(pointer)**). Находящиеся на странице узлы вставляются в свободный список с помощью простого вызова **FreeNode**. Поскольку переменная **NewPage** объявлена как **PAnsiChar**, при выполнении простых операций с указателем для идентификации отдельных узлов на странице преобразование указателя в тип **integer** не требуется.

Листинг 3.5. Распределение новой страницы в классе TtdNodeManager

```

procedure TtdNodeManager.nmAllocNewPage;
var
  NewPage : PAnsiChar;
  i : integer;
begin
  {распределить новую страницу и вставить ее в начало списка страниц}
  GetMem(NewPage, FPageSize);
  PGenericNode(NewPage)^.gnNext := FPageHead;
  FPageHead := NewPage;
  {разделить новую страницу на узлы и вставить их в начало свободного списка;
   обратите внимание, что первые 4 байта страницы представляют собой указатель
   на следующую страницу, поэтому не забудьте их пропустить}
  inc(NewPage, sizeof(pointer));
  for i := pred(FNodesPerPage) downto 0 do begin
    FreeNode(NewPage);
    inc(NewPage, FNodeSize);
  end;
end;

```

И, наконец, деструктор **Destroy** удаляет все страницы в списке страниц. Он ничего не делает со свободным списком, поскольку все узлы в нем находятся на освобождаемых страницах и будут освобождены в любом случае.

Листинг 3.6. Удаление экземпляра класса `TtdNodeManager`

```

destructor TtdNodeManager.Destroy;
var
  Temp : pointer;
begin
  {освободить все имеющиеся страницы}
  while (FPageHead <> nil) do begin
    Temp := PGenericNode(FPageHead)^.gnNext;
    FreeMem(FPageHead, FPageSize);
    FPageHead := Temp;
  end;
  inherited Destroy;
end;

```

Рекомендации на будущее. Уже в недалеком будущем мы получим новую версию Windows, использующую 64-битные указатели и написанную для 64-разрядных процессоров Intel. Понятно, что подобная же участь ждет и операционную систему Linux. Вскоре после выхода в свет 64-разрядных систем на рынке появятся версии Delphi и Kylix, поддерживающие длинные указатели. Весь код в настоящей книге основан на предположении, что длина указателей может составлять и не 4 байта, или 32 бита. Для определения длины указателей используется функция `sizeof(pointer)`. Нигде в коде мы не считаем значения `sizeof(pointer)` и `sizeof(longint)` равными – простая хитрость, которая может оказаться полезной при работе с будущими версиями Delphi. Примером описанного принципа программирования может служить диспетчер узлов.

Полный код класса **`TtdNodeManager`** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **`TDNdeMgr.pas`**.

Перед тем как вернуться к дальнейшим исследованиям односвязных списков, с которых мы начали наши рассуждения, отметим несколько недостатков класса **`TtdNodeManager`**. Первое, что следует отметить – это то, что метод **`FreeNode`** не проверяет, принадлежит ли освобождаемый узел к требуемому классу, т.е. находится ли он на странице, контролируемой классом. Это основной вопрос правильного функционирования класса. Если у класса есть узел, который не принадлежит к данному классу, он может иметь неверную длину (что в итоге может привести к перезаписи памяти) или может принадлежать к другому классу, который, возможно, очистит страницу, содержащую узел, и т.д. При отладке имеет смысл проводить проверку принадлежности к классу всех освобождаемых узлов. Реализация, содержащаяся в рамках сопровождающих книгу материалов, включает специальный код проверки при условии, что модель будет компилироваться с использованием утверждений.

Вторая проблема вызвана тем, что мы легко можем удалить экземпляр диспетчера узлов до удаления объектов, которые эти узлы используют. Это приведет к возникновению неизвестных ошибок. Только достаточная внимательность во время программирования может нас избавить от такого рода ошибок.

(Кстати, в качестве простого доказательства того, что мы не зря потеряли время на реализацию диспетчера узлов, можно сказать, что тесты по распределению и освобождению одного миллиона узлов показали, что диспетчер узлов работает в 3–4 раза быстрее, чем диспетчер кучи Delphi.)

Класс односвязного списка

Перед тем как приступить к реализации класса **TtdSingleLinkedList** для представления односвязного списка, рассмотрим несколько вводных замечаний.

Начнем с самого начала. Как уже упоминалось, было бы очень удобно использовать связный список, не беспокоясь о его узлах. Хотелось бы, чтобы класс связного списка мог работать с любыми типами указателей, подобно классу **TList**. Для получения доступа к элементам связного списка было бы желательно использовать индекс (несмотря на то что это может негативно сказаться на быстродействии), но еще лучше было бы использовать терминологию баз данных. Так, в связном списке можно использовать курсор, который указывает на "текущий" элемент списка. Тогда можно было бы написать методы для позиционирования курсора перед любым элементом списка, перемещения курсора на следующий элемент, вставки и удаления элемента в позиции курсора и т.д. Поскольку мы создаем связный список в виде класса, мы можем работать с родительским объектом текущего элемента, что позволит запрограммировать метод **Insert** так, как он реализован в **TList** (т.е. за счет перемещения текущего элемента и всех последующих элементов на одну позицию и вставки в освободившееся место нового элемента). Аналогично можно реализовать и метод **Delete**.

Интерфейс класса **TtdSingleLinkedList** выглядит следующим образом:

Листинг 3.7. Класс **TtdSingleLinkedList**

```
TtdSingleLinkedList = class
private
    FCount : longint;
    FCursor : PslNode;
    FCursorIx: longint;
    FDispose : TtdDisposeProc;
    FHead : PslNode;
    FName : TtdNameString;
    FParent : PslNode;
protected
    function sllGetItem(aIndex : longint) : pointer;
    procedure sllSetItem(aIndex : longint; aItem : pointer);
    procedure sllError(aErrorCode : integer;
        const aMethodName : TtdNameString);
```

```

class procedure sllGetNodeManager;
procedure sllPositionAtNth(aIndex : longint);
public
  constructor Create(aDispose : TtdDisposeProc);
  destructor Destroy; override;
  function Add(aItem : pointer) : longint;
  procedure Clear;
  procedure Delete(aIndex : longint);
  procedure DeleteAtCursor;
  function Examine : pointer;
  function First : pointer;
  function IndexOf(aItem : pointer) : longint;
  procedure Insert(aIndex : longint; aItem : pointer);
  procedure InsertAtCursor(aItem : pointer);
  function IsAfterLast : boolean;
  function IsBeforeFirst : boolean;
  function IsEmpty : boolean;
  function Last : pointer;
  procedure MoveBeforeFirst;
  procedure MoveNext;
  procedure Remove(aItem : pointer);
  procedure Sort(aCompare : TtdCompareFunc);
  property Count : longint read FCount;
  property Items[aIndex : longint] : pointer
  read sllGetItem write sllSetItem; default;
  property Name : TtdNameString read FName write FName;
end;

```

Несмотря на то что названия методов соответствуют стандарту **TList**, появилось несколько новых методов. Метод **MoveBeforeFirst** помещает курсор перед всеми элементами связного списка. **IsBeforeFirst** и **IsAfterLast** возвращают **True**, если курсор находится, соответственно, перед всеми элементами или после всех элементов списка. Метод **MoveNext** перемещает курсор на следующий элемент списка. Свойство **Items** аналогично соответствующему свойству списка **TList**: элементы нумеруются от 0 до **Count-1**.

Конструктор **Create** проверяет, создан ли экземпляр диспетчера узлов, а затем распределяет память для узла, который будет фиктивным начальным узлом. Затем курсор помещается перед всеми узлами (поскольку в списке еще нет узлов, это совсем несложно). Деструктор **Destroy** очищает связный список и освобождает фиктивный начальный узел, выделенный конструктором **Create**.

Листинг 3.8. Конструктор и деструктор класса **TtdSingleLinkedList**

```

constructor TtdSingleLinkedList.Create(aDispose : TtdDisposeProc);
begin
  inherited Create;

```

```

    {сохранить процедуру удаления}
    FDispose := aDispose;
    {получить диспетчер узлов}
    sl1GetNodeManager;
    {распределить память под начальный узел}
    FHead := PslNode(SLNodeManager.AllocNode);
    FHead^.slnNext := nil;
    FHead^.slnData := nil;
    {установить курсор}
    MoveBeforeFirst;
end;

destructor TtdSingleLinkList.Destroy;
begin
    {удалить все узлы, включая начальный фиктивный узел}
    Clear;
    SLNodeManager.FreeNode(FHead);
    inherited Destroy;
end;

```

Особый интерес здесь представляет тот факт, что связный список организован таким образом, что для всех экземпляров класса **TtdSingleLinkList** создается только один диспетчер узлов. Все экземпляры пользуются одним и тем же диспетчером. Можно было бы запрограммировать, чтобы каждый класс создавал свой диспетчер, но это бы означало использование большого дополнительного объема для экземпляра класса. Таким образом, учитывая то, что в приложении, в котором имеется один связный список, как правило, есть несколько списков, было решено ввести переменную класса. Но во всех этих рассуждениях присутствует один недостаток: Delphi не поддерживает переменные класса. Поэтому в коде мы имитируем такую переменную, объявив ее как глобальную в разделе **implementation** модуля. Если вы просмотрите содержимое файла **TDLnkLst.pas**, то найдете следующее объявление:

```

var
    SLNodeManager : TtdNodeManager;

```

Все методы класса односвязного списка можно разбить на две категории: методы, действующие по последовательной схеме (**MoveBeforeFirst**, **InsertAtCursor** и т.д.), и методы, которые работают со списком как с массивом (свойство **Items**, методы **Delete**, **IndexOf** и т.д.). Рассмотрим сначала методы первой группы, поскольку мы уже говорили о принципе их работы в начале главы при описании связных списков. Для упрощения реализации мы не только храним курсор (т.е. указатель на текущий узел) в объекте, но и родительский объект курсора (т.е. указатель на родительский объект текущего курсора). Такая схема позволяет упростить методы вставки и удаления элементов.

Листинг 3.9. Стандартные операции со связным списком для класса TtdSingleLinkedList

```
procedure TtdSingleLinkedList.Clear;
var
    Temp : PslNode;
begin
    {удалить все узлы, за исключением начального;
     при возможности освободить все данные}
    Temp := FHead^.slnNext;
    while (Temp <> nil) do begin
        FHead^.slnNext := Temp^.slnNext;
        if Assigned(FDispose) then
            FDispose(Temp^.slnData);
        SLNodeManager.FreeNode(Temp);
        Temp := FHead^.slnNext;
    end;
    FCount := 0;
    MoveBeforeFirst;
end;

procedure TtdSingleLinkedList.DeleteAtCursor;
begin
    if (FCursor = nil) or (FCursor = FHead) then
        slError(tdeListCannotDelete, 'Delete');
    {удалить все элементы}
    if Assigned(FDispose) then
        FDispose(FCursor^.slnData);
    {удалить ссылки на узел и удалить сам узел}
    FParent^.slnNext := FCursor^.slnNext;
    SLNodeManager.FreeNode(FCursor);
    FCursor := FParent^.slnNext;
    dec(FCount);
end;

function TtdSingleLinkedList.Examine : pointer;
begin
    if (FCursor = nil) or (FCursor = FHead) then
        slError(tdeListCannotExamine, 'Examine');
    {вернуть данные с позиции курсора}
    Result := FCursor^.slnData;
end;

procedure TtdSingleLinkedList.InsertAtCursor(aItem : pointer);
var
    NewNode : PslNode;
begin
    {убедиться, что вставка производится не перед первой позицией; если курсор
     находится перед первой позицией, переместить его на одну позицию вперед}
    if (FCursor = FHead) then
        MoveNext;
```

```

    {распределить новый узел и вставить его в позицию курсора}
    NewNode := PslNode(SLNodeManager.AllocNode);
    NewNode^.slnData := aItem;
    NewNode^.slnNext := FCursor;
    FParent^.slnNext := NewNode;
    FCursor := NewNode;
    inc(FCount);
end;

function TtdSingleLinkedList.IsAfterLast : boolean;
begin
    Result := FCursor = nil;
end;

function TtdSingleLinkedList.IsBeforeFirst : boolean;
begin
    Result := FCursor = FHead;
end;

function TtdSingleLinkedList.IsEmpty : boolean;
begin
    Result := (Count = 0);
end;

procedure TtdSingleLinkedList.MoveBeforeFirst;
begin
    {установить курсор на начальный узел}
    FCursor := FHead;
    FParent := nil;
    FCursorIx := -1;
end;

procedure TtdSingleLinkedList.MoveNext;
begin
    {переместить курсор по его указателю Next, игнорировать попытку выхода за
    конечный узел списка}
    if (FCursor <> nil) then begin
        FParent := FCursor;
        FCursor := FCursor^.slnNext;
        inc(FCursorIx);
    end;
end;

```

Вы, возможно, обратили внимание, что некоторые из приведенных методов пользуются полем объекта **FCursorIx**. Именно это поле позволяет обеспечить высокую эффективность методов, основанных на использовании индекса, поскольку в нем хранится индекс курсора (при этом первый узел имеет индекс 0, точно так же как в **TList**). Значение поля используется методом **slPositionAtNth**, который оптимальным образом перемещает курсор в позицию с указанным индексом.

Листинг 3.10. Метод sllPositionAtNth

```

procedure TtdSingleLinkList.sllPositionAtNth(aIndex : longint);
var
    WorkCursor : PslNode;
    WorkParent : PslNode;
    WorkCursorIx : longint;
begin
    {проверить, корректно ли задан индекс}
    if (aIndex < 0) or (aIndex >= Count) then
        sllError(tdeListInvalidIndex, 'sllPositionAtNth');
    {обработать наиболее простой случай}
    if (aIndex = FCursorIx) then
        Exit;
    {--для повышения быстродействия использовать локальные переменные--}
    {если заданный индекс меньше индекса курсора, переместить рабочий курсор в
    позицию перед всеми узлами}
    if (aIndex < FCursorIx) then begin
        WorkCursor := FHead;
        WorkParent := nil;
        WorkCursorIx := -1;
    end
    {в противном случае поставить рабочий курсор в позицию текущего курсора}
    else begin
        WorkCursor := FCursor;
        WorkParent := FParent;
        WorkCursorIx := FCursorIx;
    end;
    {пока индекс рабочего курсора меньше заданного индекса, передвинуть его на
    одну позицию вперед}
    while (WorkCursorIx < aIndex) do begin
        WorkParent := WorkCursor;
        WorkCursor := WorkCursor^.slnNext;
        inc(WorkCursorIx);
    end;
    {установить реальный курсор равным рабочему курсору}
    FCursor := WorkCursor;
    FParent := WorkParent;
    FCursorIx := WorkCursorIx;
end;

```

Метод **sllPositionAtNth** для увеличения быстродействия использует локальные переменные. Вначале метод определяет, больше ли заданный индекс индекса курсора (в этом случае поиск узла начинается с позиции курсора) или же он меньше (поиск узла начинается с начала списка). Без знания позиции курсора мы всегда бы начинали поиск с начала списка.

Реализация остальных методов, основанных на использовании индекса, после написания кода метода **sllPositionAtNth** не представляет особых трудностей.

Листинг 3.11. Методы класса TtdSingleLinkList, основанные на использовании индекса

```
procedure TtdSingleLinkList.Delete(aIndex : longint);  
begin  
    {установить курсор в позицию с заданным индексом}  
    sllPositionAtNth(aIndex);  
    {удалить элемент в позиции курсора}  
    DeleteAtCursor;  
end;  
  
function TtdSingleLinkList.First : pointer;  
begin  
    {установить курсор на первый узел}  
    sllPositionAtNth(0);  
    {вернуть данные с позиции курсора}  
    Result := FCursor^.slnData;  
end;  
  
procedure TtdSingleLinkList.Insert(aIndex : longint;  
                                   aItem : pointer);  
begin  
    {установить курсор в позицию с заданным индексом}  
    sllPositionAtNth(aIndex);  
    {вставить элемент в позицию курсора}  
    InsertAtCursor(aItem);  
end;  
  
function TtdSingleLinkList.Last : pointer;  
begin  
    {установить курсор в позицию с заданным индексом}  
    sllPositionAtNth(pred(Count));  
    {вернуть данные с позиции курсора}  
    Result := FCursor^.slnData;  
end;  
  
function TtdSingleLinkList.sllGetItem(aIndex : longint) : pointer;  
begin  
    {установить курсор в позицию с заданным индексом}  
    sllPositionAtNth(aIndex);  
    {вернуть данные с позиции курсора}  
    Result := FCursor^.slnData;  
end;  
  
procedure TtdSingleLinkList.sllSetItem(aIndex : longint;  
                                       aItem : pointer);  
begin  
    {установить курсор в позицию с заданным индексом}  
    sllPositionAtNth(aIndex);
```

```

{если возможно удалить заменяемые данные, удалить их}
if Assigned(FDispose) and (aItem <> FCursor^.slnData) then
  FDispose(FCursor^.slnData);
{заменить данные}
FCursor^.slnData := aItem;
end;

```

Теперь нам осталось рассмотреть еще несколько методов, которые по разным причинам реализованы в соответствии с главными принципами. Метод **Add** добавляет элемент в конец связанного списка. Код поиска последнего узла достаточно прост и имеет смысл реализовать его в коде самого метода. В эту группу входит и метод **IndexOf**. Поиск заданного элемента с помощью этого метода можно организовать только в коде самого метода. После написания кода метода **IndexOf** реализация **Remove** становится предельно простой.

Листинг 3.12. Методы Add, IndexOf и Remove

```

function TtdSingleLinkList.Add(aItem : pointer) : longint;
var
  WorkCursor : PslNode;
  WorkParent : PslNode;
begin
  {для увеличения быстродействия используются локальные переменные}
  WorkCursor := FCursor;
  WorkParent := FParent;
  {перейти в конец связанного списка}
  while (WorkCursor <> nil) do begin
    WorkParent := WorkCursor;
    WorkCursor := WorkCursor^.slnNext;
  end;
  {перенести реальный курсор}
  FParent := WorkParent;
  FCursor := nil;
  FCursorIx := Count;
  Result := Count;
  {вставить элемент в позицию курсора}
  InsertAtCursor(aItem);
end;

function TtdSingleLinkList.IndexOf(aItem : pointer) : longint;
var
  WorkCursor : PslNode;
  WorkParent : PslNode;
  WorkCursorIx : longint;
begin
  {установить рабочий курсор на первый узел (если таковой существует)}
  WorkParent := FHead;
  WorkCursor := WorkParent^.slnNext;
  WorkCursorIx := 0;

```

```
{идти по списку в поисках требуемого элемента}
while (WorkCursor <> nil) do begin
  if (WorkCursor^.slnData = aItem) then begin
    {требуемый элемент найден; записать результат; установить реальный
    курсор в позицию рабочего курсора}
    Result := WorkCursorIx;
    FCursor := WorkCursor;
    FParent := WorkParent;
    FCursorIx := WorkCursorIx;
    Exit;
  end;
  {перейти к следующему узлу}
  WorkParent := WorkCursor;
  WorkCursor := WorkCursor^.slnNext;
  inc(WorkCursorIx);
end;
{требуемый элемент не найден}
Result := -1;
end;

procedure TtdSingleLinkList.Remove(aItem : pointer);
begin
  if (IndexOf(aItem) <> -1) then
    DeleteAtCursor;
end;
```

Полный код класса **TtdSingleLinkList** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDLn1Lst.pas**.

Только что написанный нами класс обладает максимально возможной эффективностью. Узлы распределяются блоками. Определяющим фактором эффективности перехода от одного узла к другому, в общем случае, является скорость работы операционной системы по листанию страниц виртуальной памяти, но очевидно, что она будет зависеть от схемы использования связного списка. Если вставки и удаления элементов выполняются в случайном порядке, узлы будут разбросаны по различным страницам памяти. Как и в случае с классом **TList**, данные, на которые указывают ссылки каждого узла, будут находиться в разных участках памяти. Но здесь, к сожалению, мы ничего поделать не можем.

Двухсвязные списки

После достаточно подробных исследований односвязных списков можно переходить к рассмотрению двухсвязных списков. Как и в случае с односвязными списками, здесь имеется набор связанных между собой узлов, но помимо ссылки на следующий узел существует ссылка и на предыдущий узел:

type

PSimpleNode = ^TSimpleNode;

TSimpleNode = **record**

Next : PSimpleNode;

Prior : PSimpleNode;

Data : SomeDataType;

end;

Таким образом, двухсвязный список позволяет двигаться по узлам не только вперед, по ссылкам **Next**, но и назад, по ссылкам **Prior**. Схематично двухсвязный список показан на рис. 3.4.

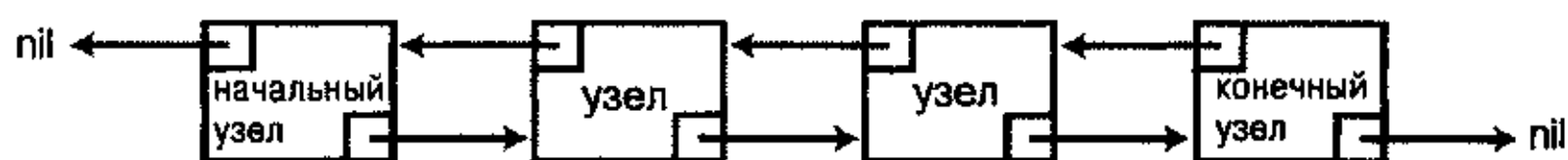


Рисунок 3.4. Двухсвязный список

Вставка и удаление элементов в двухсвязном списке

Каким образом вставлять новый узел в двухсвязный список? В односвязном списке для этого нужно было разорвать одну ссылку и вставить две новых, а для двухсвязного списка потребуется разорвать две ссылки и вставить четыре новых. Причем вставку можно выполнять как перед, так и после определенного элемента, поскольку указатель **Prior** позволяет проходить список в обратном направлении. Фактически операцию "вставить перед" можно запрограммировать как "перейти к предыдущему узлу и вставить после". Поэтому в главе мы рассмотрим только операцию "вставить после".

Ссылка **Next** нового узла устанавливается на узел, расположенный после заданного узла, а ссылка **Next** заданного узла устанавливается на новый узел. Для установки обратных ссылок ссылка **Prior** нового узла устанавливается на заданный узел, а ссылка **Prior** узла, следующего за новым, устанавливается на новый узел. В коде это выглядит следующим образом:

var

GivenNode, NewNode : PSimpleNode;

begin

...

New(NewNode);

..задать значение поля Data..

NewNode^.Next := GivenNode^.Next;

GivenNode^.Next := NewNode;

NewNode^.Prior := GivenNode;

NewNode^.Next^.Prior := NewNode;

В случае с удалением проще всего удалить узел, находящийся после заданного узла. Необходимо установить ссылку **Next** заданного узла на узел, находящийся после удаляемого, а ссылку **Prior** узла, находящегося после удаляемого, на заданный узел.

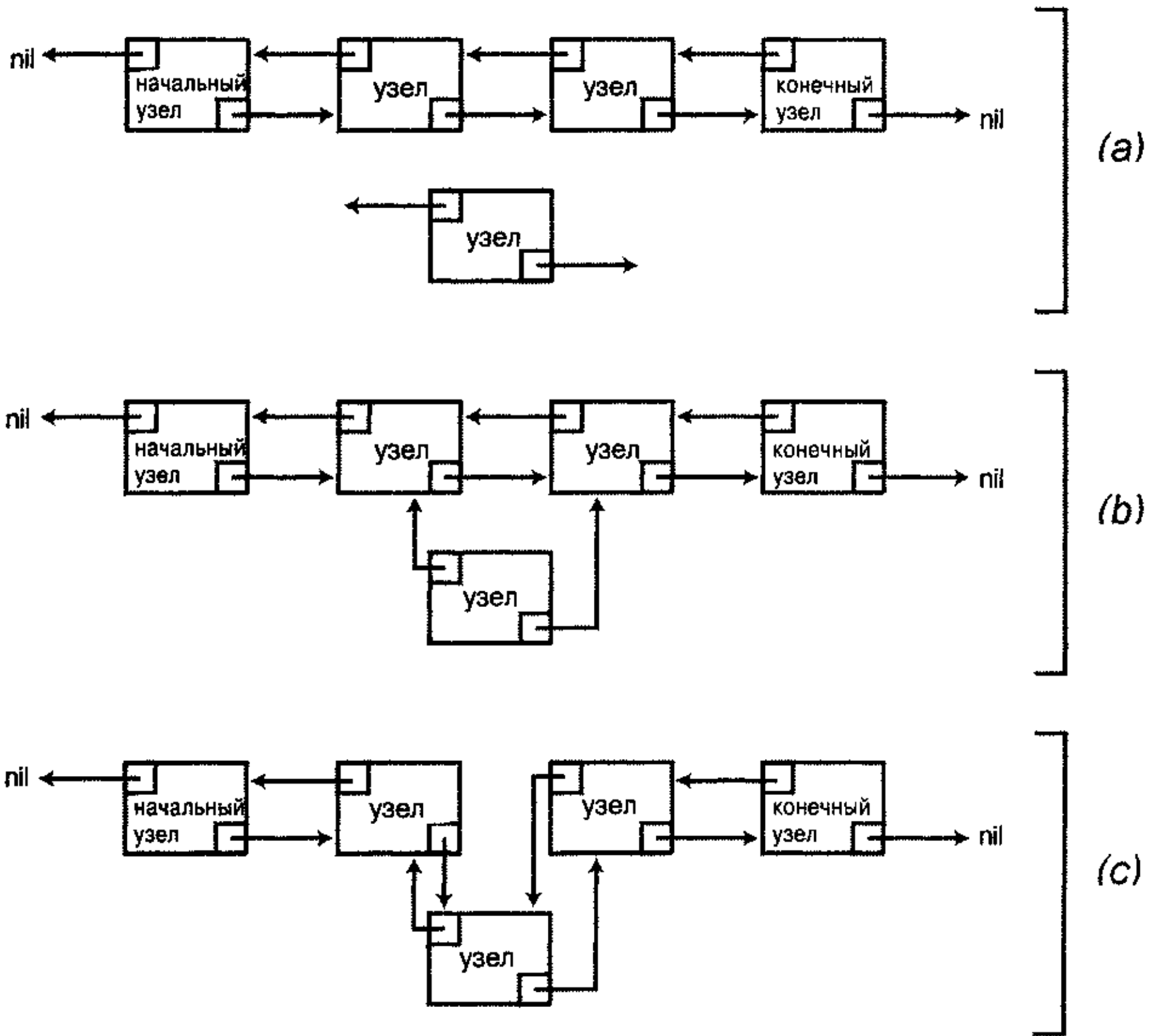


Рисунок 3.Б. Вставка нового узла в двухсвязный список

После этих операций удаляемый узел исключен из списка, и его можно удалить. В коде это выглядит следующим образом:

```

var
  GivenNode, NodeToGo : PSimpleNode;
begin
  . . .
  NodeToGo := GivenNode^.Next;
  GivenNode^.Next := NodeToGo^.Next;
  NodeToGo^.Next^.Prior := GivenNode;
  Dispose(NodeToGo);

```

Как и для односвязных списков, здесь для обеих операций существуют специальные случаи: вставка перед первым элементом списка (т.е. новый элемент становится первым) и удаление первого элемента списка (т.е. первым становится другой элемент). Поскольку в наших рассуждениях первый элемент считается определяющим узлом всего списка, код для этих случаев потребует написать отдельно.

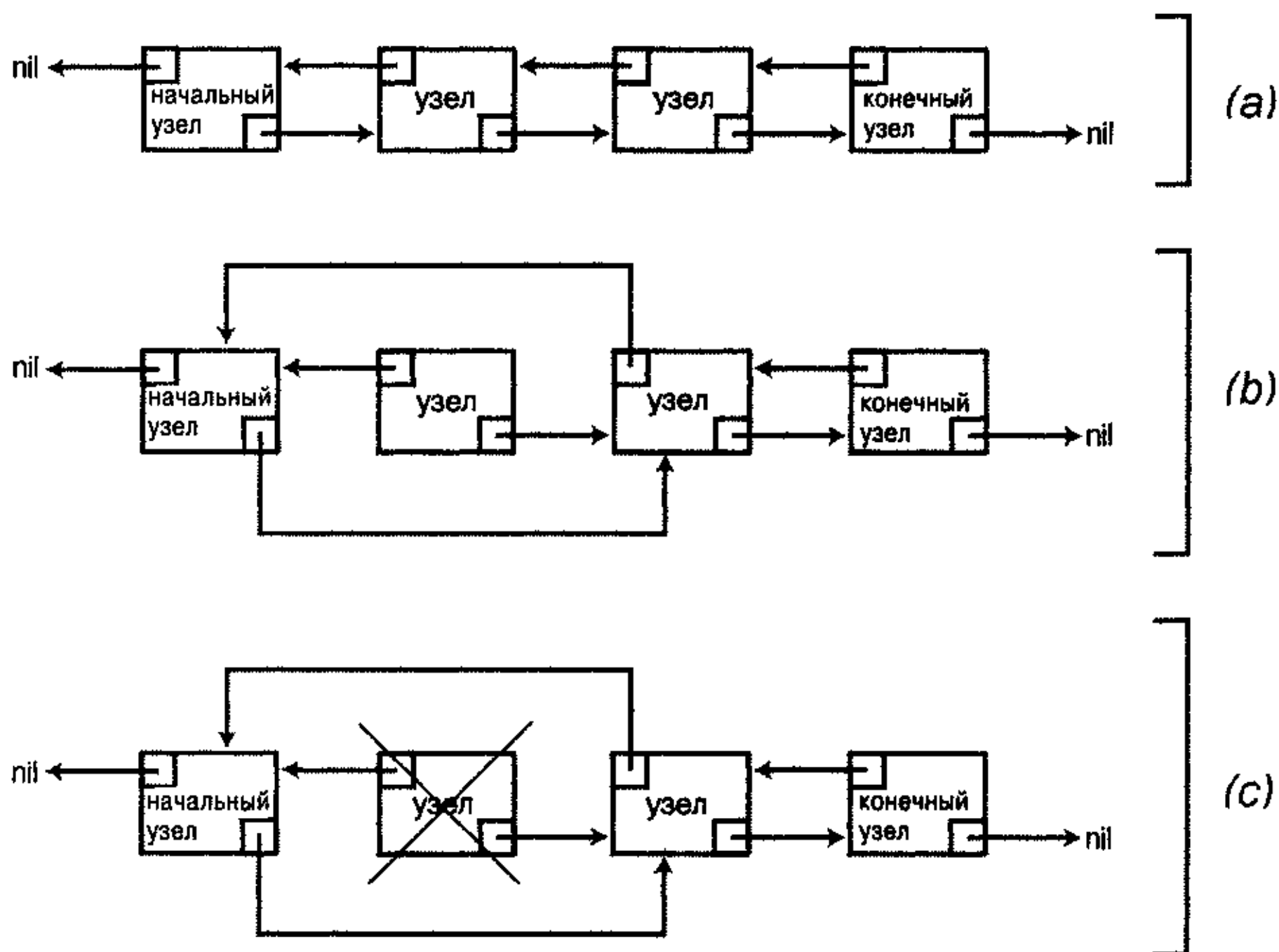


Рисунок 3.6. Удаление узла в двухсвязном списке

Вставка:

```

var
  FirstNode, NewNode : PSimpleNode;
begin
  ...
  New(NewNode);
  ..задать значение поля Data..
  NewNode^.Next := FirstNode;
  NewNode^.Prior := nil;
  FirstNode^.Prior := NewNode;
  FirstNode := NewNode;

```

Удаление:

```

var
  FirstNode, NodeToGo : PSimpleNode;
begin
  ...
  NodeToGo := FirstNode;
  FirstNode := NodeToGo^.Next;
  FirstNode^.Prior := nil;
  Dispose(NodeToGo);

```


Соображения по поводу эффективности

Как и для односвязных списков, предварительно приведем некоторые соображения, касающиеся проблемы эффективности.

Использование начального и конечного узлов

Для односвязного списка было показано, что наличие начального узла существенно упрощало операции вставки и удаления. Соответствующий случай для двухсвязного списка – наличие двух фиктивных узлов: начального и конечного. Они позволяют очень легко выполнять прохождение списка от первого узла к последнему, равно как от последнего к первому. Специальные случаи при этом исключаются.

Использование диспетчера узлов

Как и для односвязного списка, данные в списке удобно хранить в виде указателей. Это позволяет написать общий класс двухсвязного списка. В двухсвязном списке в каждом узле будет находиться прямой указатель, обратный указатель и указатель на данные. Общий размер узла составит 12 байт (т.е. $3 * \text{sizeof}(\text{pointer})$). Все узлы одинаковы, таким образом, можно реализовать диспетчер узлов и для двухсвязного списка.

Класс двухсвязного списка

Интерфейс класса двухсвязного списка выглядит следующим образом:

Листинг 3.13. Класс TtdDoubleLinkedList

```
TtdDoubleLinkedList = class
  private
    FCount : longint;
    FCursor : PdlNode;
    FCursorIx: longint;
    FDispose : TtdDisposeProc;
    FHead : PdlNode;
    FName : TtdNameString;
    FTail : PdlNode;
  protected
    function dllGetItem(aIndex : longint) : pointer;
    procedure dllSetItem(aIndex : longint; aItem : pointer);
    procedure dllError(aErrorCode : integer;
                      const aMethodName : TtdNameString);
    class procedure dllGetNodeManager;
    procedure dllPositionAtNth(aIndex : longint);
  public
    constructor Create(aDispose : TtdDisposeProc);
    destructor Destroy; override;
    function Add(aItem : pointer) : longint;
    procedure Clear;
    procedure Delete(aIndex : longint);
```

```

procedure DeleteAtCursor;
function Examine : pointer;
function First : pointer;
function IndexOf(aItem : pointer) : longint;
procedure Insert(aIndex : longint; aItem : pointer);
procedure InsertAtCursor(aItem : pointer);
function IsAfterLast : boolean;
function IsBeforeFirst : boolean;
function IsEmpty : boolean;
function Last : pointer;
procedure MoveAfterLast;
procedure MoveBeforeFirst;
procedure MoveNext;
procedure MovePrior;
procedure Remove(aItem : pointer);
procedure Sort(aCompare : TtdCompareFunc);
property Count : longint read FCount;
property Items[aIndex : longint] : pointer
    read dllGetItem write dllSetItem; default;
property Name : TtdNameString read FName write FName;
end;

```

Как видите, этот интерфейс очень похож на интерфейс класса **TtdSingleLinkedList**. Собственно, так и должно быть. Для пользователя должно быть безразлично, какой класс он выбирает, оба они должны работать одинаково. Сам выбор односвязного или двухсвязного списка должен зависеть от назначения. Если большая часть перемещений курсора направлена вперед и доступ к случайным элементам выполняется редко, эффективнее использовать односвязный список. Если же высока вероятность того, что список будет проходиться как в прямом, так и в обратном направлениях, то, несмотря на большие требования к памяти, лучше выбрать двухсвязный список. Если же ожидается, что доступ к элементам списка будет осуществляться, в основном, в случайном порядке, выберите класс **TList**, несмотря на то, что он требует несколько большего времени на вставку и удаление элемента.

Поскольку в двухсвязном списке присутствует обратный указатель, реализация методов класса проще, нежели для односвязного списка. Теперь у нас имеется возможность перейти к предыдущему элементу, если это будет необходимо.

Конструктор **Create** распределяет при помощи диспетчера узлов еще один дополнительный фиктивный узел – **FTail**. Как упоминалось во введении к двухсвязным спискам, он предназначен для обозначения конца списка. Начальный и конечный фиктивные узлы вначале будут связаны друг с другом, т.е. ссылка **Next** начального узла указывает на конечный узел, а ссылка **Prior** конечного узла – на начальный узел. Естественно, деструктор **Destroy** будет удалять фиктивный конечный узел и возвращать его вместе с начальным узлом в диспетчер узлов.

Листинг 3.14. Конструктор Create и деструктор Destroy класса TtdDoubleLinkedList

```

constructor TtdDoubleLinkedList.Create;
begin
    inherited Create;
    {сохранить процедуру удаления}
    FDispose := aDispose;
    {получить диспетчер узлов}
    dllGetNodeManager;
    {распределить и связать начальный и конечный узлы}
    FHead := PdlNode(DLNodeManager.AllocNode);
    FTail := PdlNode(DLNodeManager.AllocNode);
    FHead^.dlnNext := FTail;
    FHead^.dlnPrior := nil;
    FHead^.dlnData := nil;
    FTail^.dlnNext := nil;
    FTail^.dlnPrior := FHead;
    FTail^.dlnData := nil;
    {установить курсор на начальный узел}
    FCursor := FHead;
    FCursorIx := -1;
end;

destructor TtdDoubleLinkedList.Destroy;
begin
    if (Count <> 0) then
        Clear;
        DLNodeManager.FreeNode(FHead);
        DLNodeManager.FreeNode(FTail);
    inherited Destroy;
end;

```

Методы последовательного доступа, т.е. традиционные для связанных списков методы, реализуются для двухсвязного списка очень просто. Нам уже не требуется сохранять родительский узел, что упрощает реализацию, однако при вставке и удалении элементов приходится работать с четырьмя указателями, а не с двумя, как это имело место для односвязного списка.

Листинг 3.15. Стандартные для связного списка операции для класса TtdDoubleLinkedList

```

procedure TtdDoubleLinkedList.Clear;
var
    Temp : PdlNode;
begin
    {удалить все узлы, за исключением начального и конечного; если возможно их освободить, то сделать это}
    Temp := FHead^.dlnNext;

```

```

while (Temp <> FTail) do begin
    FHead^.dlnNext := Temp^.dlnNext;
    if Assigned(FDispose) then
        FDispose(Temp^.dlnData);
    DLNodeManager.FreeNode(Temp);
    Temp := FHead^.dlnNext;
end;
    {устранить "дыру" в связанном списке}
    FTail^.dlnPrior := FHead;
    FCount := 0;
    {установить курсор на начальный узел}
    FCursor := FHead;
    FCursorIx := -1;
end;

procedure TtdDoubleLinkedList.DeleteAtCursor;
var
    Temp : PdlNode;
begin
    {запустить в Temp удаляемый узел}
    Temp := FCursor;
    if (Temp = FHead) or (Temp = FTail) then
        dllError(tdeListCannotDelete, 'Delete');
    {избавиться от его содержимого}
    if Assigned(FDispose) then
        FDispose(Temp^.dlnData);
    {удалить ссылки на узел и освободить его; курсор перемещается на следующий узел}
    Temp^.dlnPrior^.dlnNext := Temp^.dlnNext;
    Temp^.dlnNext^.dlnPrior := Temp^.dlnPrior;
    FCursor := Temp^.dlnNext;
    DLNodeManager.FreeNode(Temp);
    dec(FCount);
end;

function TtdDoubleLinkedList.Examine : pointer;
begin
    if (FCursor = nil) or (FCursor = FHead) then
        dllError(tdeListCannotExamine, 'Examine');
    {вернуть данные узла в позиции курсора}
    Result := FCursor^.dlnData;
end;

procedure TtdDoubleLinkedList.InsertAtCursor(aItem : pointer);
var
    NewNode : PdlNode;
begin
    {если курсор находится на начальном узле, не генерировать исключение, а перейти на следующий узел}
    if (FCursor = FHead) then
        MoveNext;

```

```
{распределить новый узел и вставить его перед позицией курсора}
NewNode := PdlNode(DLNodeManager.AllocNode);
NewNode^.dlnData := aItem;
NewNode^.dlnNext := FCursor;
NewNode^.dlnPrior := FCursor^.dlnPrior;
NewNode^.dlnPrior^.dlnNext := NewNode;
FCursor^.dlnPrior := NewNode;
FCursor := NewNode;
inc(FCount);
end;
function TtdDoubleLinkedList.IsAfterLast : boolean;
begin
    Result := FCursor = FTail;
end;
function TtdDoubleLinkedList.IsBeforeFirst : boolean;
begin
    Result := FCursor = FHead;
end;
function TtdDoubleLinkedList.IsEmpty : boolean;
begin
    Result := (Count = 0);
end;
procedure TtdDoubleLinkedList.MoveAfterLast;
begin
    {установить курсор на конечный узел}
    FCursor := FTail;
    FCursorIx := Count;
end;
procedure TtdDoubleLinkedList.MoveBeforeFirst;
begin
    {установить курсор на начальный узел}
    FCursor := FHead;
    FCursorIx := -1;
end;
procedure TtdDoubleLinkedList.MoveNext;
begin
    {переместить курсор по его прямому указателю}
    if (FCursor <> FTail) then begin
        FCursor := FCursor^.dlnNext;
        inc(FCursorIx);
    end;
end;
procedure TtdDoubleLinkedList.MovePrior;
begin
    {переместить курсор по его обратному указателю}
    if (FCursor <> FHead) then begin
        FCursor := FCursor^.dlnPrior;
        dec(FCursorIx);
    end;
end;
end;
```

Если сравнить приведенный код с его эквивалентом для односвязных списков (листинг 3.9), можно понять, каким образом дополнительные обратные связи влияют на реализацию методов. С одной стороны, методы стали немного проще. Так, например, в случае двухсвязных списков для метода **MoveNext** не нужно вводить переменную **FParent**. С другой стороны, требуется дополнительный код для обработки обратных ссылок. Примером могут служить методы **InsertAtCursor** и **DeleteAtCursor**.

Методы, основанные на использовании индекса, в случае двухсвязного списка реализуются проще, чем в случае односвязного. Единственную сложность представляет метод **dllPositionAtNth**, предназначенный для установки курсора в позицию с заданным индексом. Вспомните алгоритм для односвязного списка: если заданный индекс соответствует позиции после курсора, начать с позиции курсора и идти вперед, вычисляя индекс. В двухсвязном списке при необходимости можно двигаться и в обратном направлении. Таким образом, алгоритм поиска можно немного изменить. Как и ранее, мы определяем, где по отношению к курсору находится узел с заданным индексом. После этого выполняется еще одно вычисление – ближе к какому узлу находится узел с заданным индексом: к начальному, конечному или к текущему? Далее мы начинаем прохождение с ближайшего узла, при необходимости двигаясь вперед или назад.

Листинг 3.16. Установка курсора на узел с индексом n для класса **TtdDoubleLinkedList**

```
procedure TtdDoubleLinkedList.dllPositionAtNth(aIndex : longint);
var
    WorkCursor : PdllNode;
    WorkCursorIx : longint;
begin
    {проверить, корректно ли задан индекс}
    if (aIndex < 0) or (aIndex >= Count) then
        dllError(tdeListInvalidIndex, 'dllPositionAtNth');
    {для увеличения быстродействия используются локальные переменные}
    WorkCursor := FCursor;
    WorkCursorIx := FCursorIx;
    {обработать наиболее простой случай}
    if (aIndex = WorkCursorIx) then
        Exit;
    {заданный индекс либо перед курсором, либо после него; в любом случае,
    заданный индекс ближе либо к курсору, либо к соответствующему концу списка;
    определить самый короткий путь}
    if (aIndex < WorkCursorIx) then begin
        if ((aIndex - 0) < (WorkCursorIx - aIndex)) then begin
            {начать с начального узла и двигаться вперед до индекса aIndex}
            WorkCursor := FHead;
            WorkCursorIx := -1;
        end;
    end
end
```



```

else (aIndex > FCursorIx) begin
  if ((aIndex - WorkCursorIx) < (Count - aIndex)) then begin
    {начать с конечного узла и двигаться назад до индекса aIndex}
    WorkCursor := FTail;
    WorkCursorIx := Count;
  end;
end;
{пока индекс рабочего курсора меньше заданного индекса, перемещать рабочий
курсor на одну позицию вперед}
while (WorkCursorIx < aIndex) do begin
  WorkCursor := WorkCursor^.dlnNext;
  inc(WorkCursorIx);
end;
{пока индекс рабочего курсора больше заданного индекса, перемещать рабочий
курсor на одну позицию назад}
while (WorkCursorIx > aIndex) do begin
  WorkCursor := WorkCursor^.dlnPrior;
  dec(WorkCursorIx);
end;
{установить реальный курсор равным рабочему курсору}
FCursor := WorkCursor;
FCursorIx := WorkCursorIx;
end;

```

Теперь, когда мы умеем находить узел по заданному индексу, можно перейти к реализации остальных методов: все они очень похожи на соответствующие методы для односвязных списков.

Листинг 3.17. Методы класса TtdDoubleLinkedList, основанные на использовании индекса

```

function TtdDoubleLinkedList.Add(aItem : pointer) : longint;
begin
  {перейти к концу связного списка}
  FCursor := FTail;
  FCursorIx := Count;
  {вернуть индекс нового узла}
  Result := Count;
  {вставить элемент в позицию курсора}
  InsertAtCursor(aItem);
end;

procedure TtdDoubleLinkedList.Delete(aIndex : longint);
begin
  {установить курсор в позицию с заданным индексом}
  dllPositionAtNth(aIndex);
  {удалить элемент в позиции курсора}
  DeleteAtCursor;
end;

```

```

function TtdDoubleLinkedList.dllGetItem(aIndex : longint) : pointer;
begin
    {установить курсор в позицию с заданным индексом}
    dllPositionAtNth(aIndex);
    {вернуть данные из позиции курсора}
    Result := FCursor^.dlnData;
end;

procedure TtdDoubleLinkedList.dllSetItem(aIndex : longint;
                                         aItem : pointer);
begin
    {установить курсор в позицию с заданным индексом}
    dllPositionAtNth(aIndex);
    {если возможно удалить заменяемые данные, удалить их}
    if Assigned(FDispose) and (aItem <> FCursor^.dlnData) then
        FDispose(FCursor^.dlnData);
    {заменить данные}
    FCursor^.dlnData := aItem;
end;

function TtdDoubleLinkedList.First : pointer;
begin
    {установить курсор на первый узел}
    dllPositionAtNth(0);
    {вернуть данные из позиции курсора}
    Result := FCursor^.dlnData;
end;

function TtdDoubleLinkedList.IndexOf(aItem : pointer) : longint;
var
    WorkCursor : PdllNode;
    WorkCursorIx : longint;
begin
    {установить рабочий курсор на первый узел (если он существует)}
    WorkCursor := FHead^.dlnNext;
    WorkCursorIx := 0;
    {идти по списку в поисках требуемого элемента}
    while (WorkCursor <> FTail) do begin
        if (WorkCursor^.dlnData = aItem) then begin
            {требуемый элемент найден; записать результат; установить реальный
            курсор в позицию рабочего курсора}
            Result := WorkCursorIx;
            FCursor := WorkCursor;
            FCursorIx := WorkCursorIx;
            Exit;
        end;
        {перейти к следующему узлу}
        WorkCursor := WorkCursor^.dlnNext;
        inc(WorkCursorIx);
    end;
    {требуемый элемент не найден}
    Result := -1;
end;

```

```
procedure TtdDoubleLinkedList.Insert(aIndex : longint;  
                                     aItem : pointer);  
begin  
    {установить курсор в позицию с заданным индексом}  
    dllPositionAtNth(aIndex);  
    {вставить элемент в позицию курсора}  
    InsertAtCursor(aItem);  
end;  
  
function TtdDoubleLinkedList.Last : pointer;  
begin  
    {установить курсор на последний узел}  
    dllPositionAtNth(pred(Count));  
    {вернуть данные из позиции курсора}  
    Result := FCursor^.dllData;  
end;  
  
procedure TtdDoubleLinkedList.Remove(aItem : pointer);  
begin  
    if (IndexOf(aItem) <> -1) then  
        DeleteAtCursor;  
end;
```

Полный код класса **TtdDoubleLinkedList** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDLnkLst.pas**.

Достоинства и недостатки связных списков

Связные списки обладают одним очень важным преимуществом: для них операции вставки и удаления принадлежат к классу $O(1)$. Независимо от текущего элемента списка и его емкости, для вставки или удаления элемента всегда требуется одно и то же время.

Основным недостатком связных списков является то, что получение доступа к их элементам принадлежит к классу $O(n)$. В этом случае важно количество элементов в списке: при поиске n -ного элемента мы начинаем с некоторой позиции в списке и переходим по ссылкам вплоть до искомого элемента. Чем больше элементов в списке, тем больше переходов придется совершить. Для увеличения быстродействия в реализации классов списков мы воспользовались небольшими хитростями, но, тем не менее, операция все равно принадлежит к классу $O(n)$.

По сравнению с классом **TList** связные списки требуют большего объема памяти. В качестве ссылки на элемент в **TList** используется один указатель, т.е. в массиве **TList** для каждого элемента требуется, по крайней мере, **sizeof(pointer)** байт. С другой стороны, односвязный список содержит два указателя: указатель на данные и указатель на следующий элемент. Таким образом, для каждого элемента в односвязном списке нужно, по меньшей мере, **2*sizeof(pointer)** байт.

Очевидно, что для каждого элемента в двухсвязном списке требуется не менее $3 \cdot \text{sizeof}(\text{pointer})$ байт.

Но это еще не все. Если неэффективно использовать массив **TList** (другими словами, не использовать свойство **Capacity** для установки размера массива), будут распределяться несколько блоков памяти, каждый из которых больше предыдущего, и потребуются большой объем работ, связанный с копированием данных массива. Если элементы вставляются только в начало, быстродействие массива **TList** существенно уменьшается. В настоящей книге будут приведены несколько реализаций алгоритмов и структур данных, которые позволяют достичь для связанных списков гораздо большей эффективности, нежели это показывает **TList**, однако в общем случае массив **TList** лучше, быстрее и эффективнее связанных списков.

Стеки

Еще одной известной и широко используемой структурой данных является стек. Стек представляет собой структуру, которая позволяет выполнять две основных операции: заталкивание для вставки элемента в стек и выталкивание с целью считывания данных из стека. Структура устроена таким образом, что операция выталкивания всегда возвращает элемент, вставленный в стек последним (самый "новый" элемент в стеке). Другими словами, элементы в стеке считываются в порядке, обратном порядку их записи в стек. Благодаря такому устройству стек известен как контейнер магазинного типа.

Написание кода стека не представляет никаких трудностей. Причем существуют два варианта реализации: первый – на основе односвязного списка, второй – на основе массива. Как и в случае со списками, будем считать, что записываться и считываться из стека будут указатели на элементы. Сначала рассмотрим организацию стека на базе связного списка.

Стеки на основе односвязных списков

В реализации стеков на основе односвязных списков операция заталкивания представляет собой вставку элемента в начало списка, а операция выталкивания – удаление элемента из начала списка и считывание его данных. Обе операции не зависят от количества элементов в списке, следовательно, их можно отнести к классу $O(1)$. Вот и все, что касается организации стека.

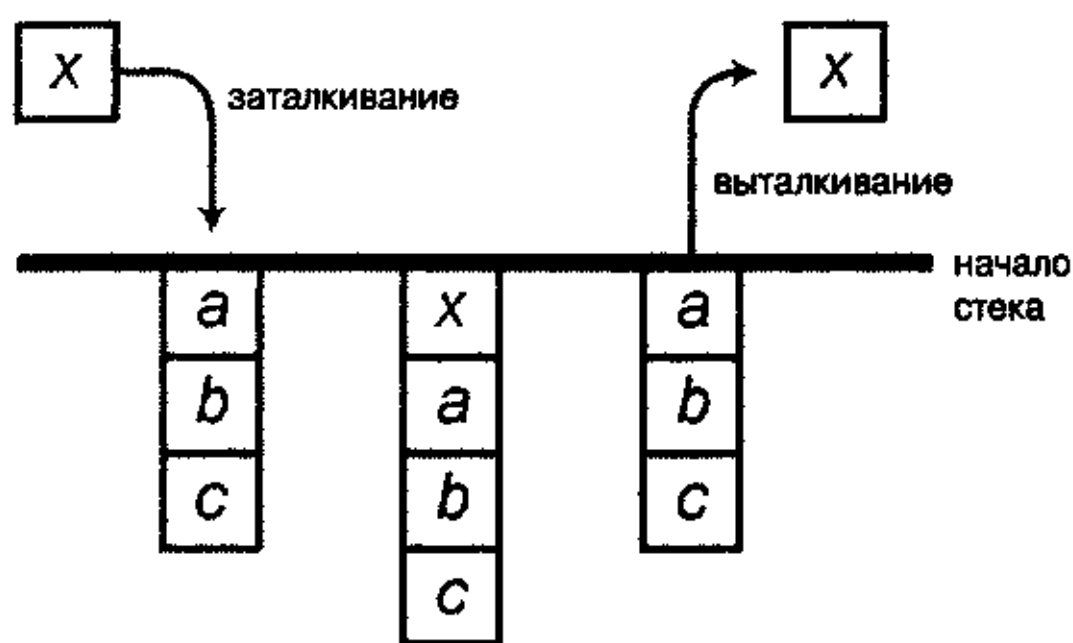


Рисунок 3.7. Операции заталкивания и выталкивания для стека

Конечно, реализация описанной организации требует большего объема в плане принятия решений. Класс стека можно реализовать как дочерний класса односвязного списка или делегировать операции заталкивания и выталкивания внутреннему экземпляру класса связного списка. Первый вариант не особенно эффективен: мы приходим к реализации класса с методами **Push** и **Pop**, но при этом у нас остаются и другие методы связного списка (**Insert**, **Delete** и т.д.). Понятно, что это не самое лучшее решение.

Второй вариант реализации, делегирование, – чисто в духе Delphi. Класс стека можно организовать именно таким образом. Конструктор **Create** будет создавать новый экземпляр класса **TtdSingleLinkedList** и устанавливать курсор после начального узла, деструктор **Destroy** будет уничтожать созданный конструктором экземпляр. Метод **Push** будет пользоваться экземпляром класса для вставки элемента в позицию курсора, а метод **Pop** будет удалять элемент в позиции курсора, предварительно сохранив его значение. Вполне реализуемое решение.

Тем не менее, мы будем писать класс **TtdStack**, исходя из первых принципов. **TtdStack** – простой класс, и за счет этого мы попытаемся увеличить его быстродействие и эффективность.

Листинг 3.18. Класс **TtdStack**

```

TtdStack = class
  private
    FCount : longint;
    FDispose : TtdDisposeProc;
    FHead : PslNode;
    FName : TtdNameString;
  protected
    procedure sError(aErrorCode : integer;
                     const aMethodName : TtdNameString);
    class procedure sGetNodeManager;
  public
    constructor Create(aDispose : TtdDisposeProc);
    destructor Destroy; override;
    procedure Clear;
    function Examine : pointer;
    function IsEmpty : boolean;
    function Pop : pointer;
    procedure Push(aItem : pointer);
    property Count : longint read FCount;
    property Name : TtdNameString read FName write FName;
end;
```

Метод **Examine** возвращает первый элемент стека, не выталкивая его из стека. Он бывает очень удобным в использовании, поскольку не требует выталкивания элемента с последующим заталкиванием. Метод **IsEmpty** возвращает значение **True**, если стек пуст, что эквивалентно проверке равенства нулю свойства **Count**.

Листинг 3.19. Методы Examine и IsEmpty для класса TtdStack

```

function TtdStack.Examine : pointer;
begin
    if (Count = 0) then
        sError(tdeStackIsEmpty, 'Examine');
    Result := FHead^.slnNext^.slnData;
end;

function TtdStack.IsEmpty : boolean;
begin
    Result := (Count = 0);
end;

```

Конструктор **Create** работает аналогично конструктору класса односвязного списка. Он проверяет, существует ли диспетчер узлов, а затем с помощью диспетчера распределяет фиктивный начальный узел, который, естественно, ни на что не указывает. Деструктор **Destroy** очищает стек и освобождает фиктивный начальный узел, **FHead**, возвращая его диспетчеру узлов.

Листинг 3.20. Конструктор и деструктор класса TtdStack

```

constructor TtdStack.Create(aDispose : TtdDisposeProc);
begin
    inherited Create;
    {сохранить процедуру удаления}
    FDispose := aDispose;
    {получить диспетчер узлов}
    sGetNodeManager;
    {распределить начальный узел}
    FHead := PslNode(SlNodeManager.AllocNode);
    FHead^.slnNext := nil;
    FHead^.slnData := nil;
end;

destructor TtdStack.Destroy;
begin
    {удалить все оставшиеся узлы; очистить начальный фиктивный узел}
    if (Count <> 0) then
        Clear;
    SlNodeManager.FreeNode(FHead);
    inherited Destroy;
end;

```

Заталкивание элемента в стек и выталкивание его из стека представляют собой короткие процедуры. **Push** распределяет новый узел при помощи диспетчера узлов и вставляет его после фиктивного начального узла. Метод **Pop** перед удалением связей узла с фиктивным узлом с помощью алгоритма "удалить после" проверяет, существует ли в стеке хотя бы один узел. Затем он возвращает элемент и освобождает узел, возвращая его диспетчеру узлов.

Листинг 3.21. Методы Push и Pop класса TtdStack

```

procedure TtdStack.Push(aItem : pointer);
var
    Temp : PslNode;
begin
    {распределить новый узел и поместить его в начало стека}
    Temp := PslNode(SLNodeManager.AllocNode);
    Temp^.slnData := aItem;
    Temp^.slnNext := FHead^.slnNext;
    FHead^.slnNext := Temp;
    inc(FCount);
end;

function TtdStack.Pop : pointer;
var
    Temp : PslNode;
begin
    if (Count = 0) then
        sError(tdeStackIsEmpty, 'Pop');
    {обратите внимание, что даже если это возможно, мы не удаляем данные узла;
     этот метод должен возвращать данные}
    Temp := FHead^.slnNext;
    Result := Temp^.slnData;
    FHead^.slnNext := Temp^.slnNext;
    SLNodeManager.FreeNode(Temp);
    dec(FCount);
end;

```

Полный код класса **TtdStack** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStkQue.pas**.

Стеки на основе массивов

После написания класса стека, основанного на связном списке, давайте перейдем к исследованию стеков, реализованных на базе массивов. Причина для организации такого класса заключается в том, что во многих случаях реализация стека на одном из простых типов (например, **char** или **double**) гораздо проще в случае применения массивов.

Ради простоты, в качестве базового массива возьмем класс **TList**. Другими словами, мы создадим класс стека указателей. В предыдущей версии стека операция **Push** вставляла узел в начало списка, а операция **Pop** выбирала узел из начала списка. Это не самый эффективный метод работы с массивами. Вставка в начало списка принадлежит к классу операций $O(n)$, а нам желательно разработать операцию класса $O(1)$, как в ситуации со связными списками. Поэтому при заталкивании и выталкивании элемента мы будем вставлять и удалять элемент в конце списка.

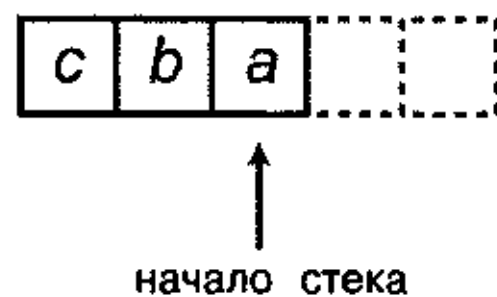


Рисунок 3.8.
Использование массива
для организации стека

Рассмотрим интерфейс класса **TtdArrayStack**. Как видите, его раздел **public** полностью соответствует разделу **public** класса **TtdStack**.

Листинг 3.22. Класс **TtdArrayStack**

```

TtdArrayStack = class
  private
    FCount : longint;
    FDispose : TtdDisposeProc;
    FList : TList;
    FName : TtdNameString;
  protected
    procedure asError(aErrorCode : integer;
                      const aMethodName : TtdNameString);
    procedure asGrow;
  public
    constructor Create(aDispose : TtdDisposeProc;
                      aCapacity : integer);
    destructor Destroy; override;
    procedure Clear;
    function Examine : pointer;
    function IsEmpty : boolean;
    function Pop : pointer;
    procedure Push(aItem : pointer);
    property Count : longint read FCount;
    property Name : TtdNameString read FName write FName;
end;
```

Конструктор и деструктор, соответственно, создает и удаляет экземпляр класса **TList**. Конструктор в качестве входного параметра принимает емкость стека. Это только начальное значение для количества элементов в экземпляре массива, предназначенное только для повышения эффективности класса, а не для установки каких-либо ограничений.

Листинг 3.23. Конструктор и деструктор класса **TtdArrayStack**

```

constructor TtdArrayStack.Create(aDispose : TtdDisposeProc;
                                aCapacity : integer);
begin
  inherited Create;
  {сохранить процедуру удаления}
  FDispose := aDispose;
  {создать внутренний экземпляр класса TList
   и установить его емкость равной aCapacity}
  FList := TList.Create;
  if (aCapacity <= 1) then
    aCapacity := 16;
  FList.Count := aCapacity;
end;
```

```
destructor TtdArrayStack.Destroy;  
begin  
    FList.Free;  
    inherited Destroy;  
end;
```

Методы **Push** и **Pop** содержат довольно-таки интересный код. Внутреннее поле **FCount** используется для двух целей. Первая цель связана с хранением количества элементов в стеке, а вторая предполагает его использование в качестве указателя стека. Для заталкивания элемента в стек мы записываем его в позицию с индексом **FCount** и увеличиваем **FCount** на единицу. Для выталкивания элемента из стека мы выполняем обратную операцию: уменьшаем значение **FCount** на единицу и возвращаем элемент с индексом **FCount**.

Листинг 3.24. Методы **Push** и **Pop** класса **TtdArrayStack**

```
procedure TtdArrayStack.asGrow;  
begin  
    FList.Count := (FList.Count * 3) div 2;  
end;  
  
function TtdArrayStack.Pop : pointer;  
begin  
    {убедиться, что стек не пуст}  
    if (Count = 0) then  
        asError(tdeStackIsEmpty, 'Pop');  
    {уменьшить значение счетчика на единицу}  
    dec(FCount);  
    {выталкиваемый элемент находится в конце списка}  
    Result := FList[FCount];  
end;  
  
procedure TtdArrayStack.Push(aItem : pointer);  
begin  
    {проверить, полон ли стек; если стек полон, увеличить емкость списка}  
    if (FCount = FList.Count) then  
        asGrow;  
    {добавить элемент в конец стека}  
    FList[FCount] := aItem;  
    {увеличить значение счетчика на единицу}  
    inc(FCount);  
end;
```

Полный код класса **TtdArrayStack** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStkQue.pas**.

Пример использования стека

Стеки используются в случае, когда требуется вычислить элементы в обратном порядке, а затем перестроить их в прямой порядок. Одним из самых простых примеров может служить изменение порядка символов в строке. При наличии стека символов задание становится очень простым: затолкнуть символы из строки в стек, а затем вытолкнуть их в обратном порядке. (Разумеется, существуют и другие методы изменения порядка символов в строке.)

Интересной вариацией этой темы является преобразование целого значения в строку. В языке Object Pascal имеются функции **Str** и **IntToStr**, которые позволяют решать поставленную задачу далеко не с нуля, но, тем не менее, задача остается достаточно интересной.

Давайте четко запишем условия задачи. Необходимо написать функцию, которая в качестве параметра принимала бы значение типа **longint** и возвращала бы значение в форме строки.

Внутри функции нужно будет вычислять цифры, соответствующие целочисленному значению. Простейший метод таких вычислений – вычислить остаток от деления значения на 10 (это будут числа от 0 до 9 включительно), сохранить его где-нибудь, поделить значение на 10 (чтобы избавиться от только что вычисленного нами значения) и повторить процесс. Цикл вычислений продолжается до тех пор, пока не будет получено значение 0.

Давайте применим описанный алгоритм (да-да, это алгоритм!) к числу 123. Остаток от деления 123 на 10 равен 3. Записываем остаток. Делим 123 на 10. Получаем 12. Остаток от деления 12 на 10 равен 2. Записываем остаток. Делим 12 на 10. Получаем 1. Остаток от деления 1 на 10 равен 1. Записываем остаток. Делим 1 на 10. Получаем 0. Завершаем вычисления. Цифры были вычислены в следующем порядке: 3, 2, 1. Однако в строке они должны находиться в обратном порядке. Мы не можем записывать цифры в строку по мере их вычисления (какой длины должна быть строка?).

Можно предложить заталкивать цифры в стек по мере их вычисления, а после выполнения вычислений определить количество элементов в стеке (т.е. длину строки) и постепенно выталкивать их в строку. Соответствующий код приведен в листинге 3.25.

Листинг 3.25. Преобразование целочисленного значения в строку

```
function tdIntToStr(aValue : longint) : string;
var
  ChStack : array [0..10] of char;
  ChSP : integer;
  IsNeg : boolean;
  i : integer;
```

```
begin
  {установить нулевую длину стека}
  ChSP := 0;
  {установить, чтобы значение было положительным}
  if (aValue < 0) then begin
    IsNeg := true;
    aValue := -aValue;
  end
  else
    IsNeg := false;
  {если значение равно нулю, записать в стек символ '0'}
  if (aValue = 0) then begin
    ChStack[ChSP] := '0';
    inc(ChSP);
  end
  {в противном случае вычислить цифры значения в обратном порядке с помощью
  описанного алгоритма и затолкнуть их в стек}
  else begin
    while (aValue <> 0) do begin
      ChStack[ChSP] := char((aValue mod 10) + ord('0'));
      inc(ChSP);
      aValue := aValue div 10;
    end;
  end;
  {если исходное значение было отрицательным, затолкнуть в стек знак минус}
  if IsNeg then begin
    ChStack[ChSP] := '-';
    inc(ChSP);
  end;
  {теперь выталкиваем значения из стека (их количество равно ChSP) в результи-
  рующую строку}
  SetLength(Result, ChSP);
  for i := 1 to ChSP do begin
    dec(ChSP);
    Result[i] := ChStack[ChSP];
  end;
end;
```

В приведенной функции присутствует несколько особенностей, о которых стоит упомянуть. Первая особенность состоит в том, что функция требует, чтобы исходное значение перед выполнением алгоритма было положительным. Если мы изменяем знак значения, то устанавливаем флаг **IsNeg**, который позволит в дальнейшем записать в строку знак минус. Вторая особенность – отдельно обрабатывается случай, когда значение равно 0. Без этого при нулевом входном значении мы бы получили пустую строку.

Следующий аспект. Стек символов был написан "с нуля". Почему? Мы уже имеем два класса стеков. Разве мы не можем использовать их?

Ответы на эти вопросы возвращают нас к тому, о чем уже ранее говорилось в книге: иногда эффективнее написать простой контейнер (в нашем случае стек) с самого начала. При написании кода преобразования целочисленного значения в строку максимальная длина значения будет составлять 10 цифр (поскольку максимальное значение типа `longint` – 2 147 483 648 – 10-значное число). Эту длину нужно увеличить на 1 – возможный знак минус. Столь короткую строку вполне можно поместить в стек.

Очереди

И, наконец, последним моментом, который мы рассмотрим в этой главе, будут очереди – последняя базовая структура данных. В то время как извлечение элементов из стека происходит в порядке, обратном тому, в котором они вносились, в очереди элементы выбираются в порядке их добавления. Таким образом, очередь относится к структурам типа "первый пришел, первый вышел" (FIFO – first in, first out). С очередью связаны две основные операции: постановка в очередь (т.е. добавление нового элемента в очередь) и снятие с очереди (т.е. извлечение из нее самого старого элемента).

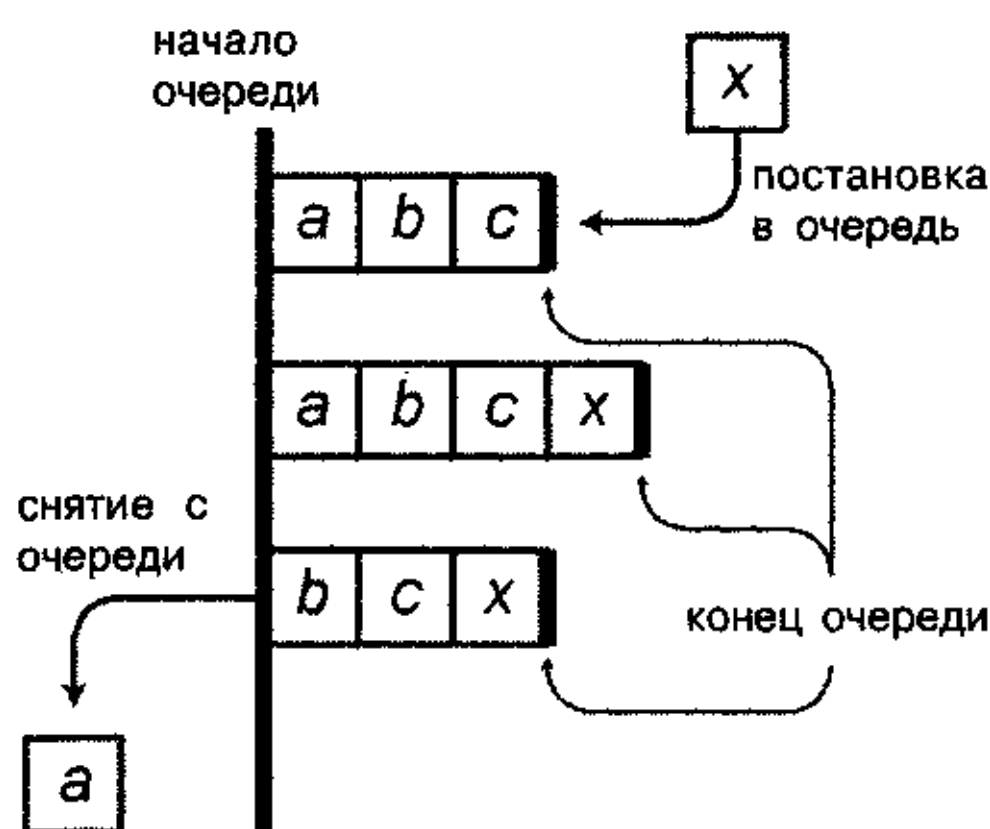


Рисунок 3.9. Постановка в очередь и снятие с очереди

Иногда эти операции ошибочно называют заталкиванием и выталкиванием. Это абсолютно неверные термины для очереди. Ближе к истине будут слова *включение* и *исключение*.

Как и стеки, очереди можно реализовать на основе односвязных списков или массивов. Тем не менее, в отличие от стеков, очень трудно добиться высокой эффективности реализации на основе массивов. К тому же организация очередей на базе связных списков ничуть не сложнее. Поэтому давайте для начала рассмотрим построение очереди на базе односвязных списков.

Очереди на основе односвязных списков

Фактически мы должны смоделировать обычную очередь в универмаге. С помощью списков это можно сделать очень легко, поскольку сами списки по своей сути являются очередями. Просто для моделирования очереди элементы должны

добавляться с одной стороны и удаляться с другой. При использовании односвязного списка снятие с очереди будет выполняться с начала списка, а постановка в очередь – в конец списка. Для двухсвязных списков для постановки или снятия с очереди может выбираться как начало, так и конец. Но в этом случае очередь будет требовать больший объем памяти. Очевидно, что обе операции с очередью не зависят от количества элементов в ней, т.е. они принадлежат к классу $O(1)$.

Как и для класса **TtdStack**, код класса **TtdQueue** будет разрабатываться на основе главных принципов. Аргументы за использование такой схемы мы рассматривали во время написания кода для класса стека.

Листинг 3.26. Класс **TtdQueue**

```

TtdQueue = class
  private
    FCount : longint;
    FDispose : TtdDisposeProc;
    FHead : PslNode;
    FName : TtdNameString;
    FTail : PslNode;
  protected
    procedure qError(aErrorCode : integer;
                    const aMethodName : TtdNameString);
    class procedure qGetNodeManager;
  public
    constructor Create(aDispose : TtdDisposeProc);
    destructor Destroy; override;
    procedure Clear;
    function Dequeue : pointer;
    procedure Enqueue(aItem : pointer);
    function Examine : pointer;
    function IsEmpty : boolean;
    property Count : longint read FCount;
    property Name : TtdNameString read FName write FName;
end;

```

Как и ранее, конструктор **Create** проверяет, существует ли экземпляр диспетчера узлов, а затем распределяет с его помощью фиктивный начальный узел. Затем инициализируется специальный указатель **FTail**, который при создании указывает на начальный узел. Его содержимое будет меняться, чтобы он всегда указывал на последний узел связного списка. Это позволит легко вставлять новые элементы после конечного узла.

Листинг 3.27. Конструктор и деструктор для класса **TtdQueue**

```

constructor TtdQueue.Create(aDispose : TtdDisposeProc);
begin
  inherited Create;

```

```

    {сохранить процедуру удаления}
    FDispose := aDispose;
    {получить диспетчер узлов}
    qGetNodeManager;
    {распределить и связать начальный и конечный узлы}
    FHead := PslNode(SLNodeManager.AllocNode);
    FHead^.slnNext := nil;
    FHead^.slnData := nil;
    {установить указатель конечного узла на начальный узел}
    FTail := FHead;
end;

destructor TtdQueue.Destroy;
begin
    {удалить все оставшиеся узлы; очистить начальный фиктивный узел}
    if (Count <> 0) then
        Clear;
    SLNodeManager.FreeNode(FHead);
    inherited Destroy;
end;

```

А теперь перейдем к методу **Enqueue**. Он посредством диспетчера узлов распределяет новый узел и устанавливает его указатель данных на вставляемый элемент. Затем используется указатель **FTail**. Учитывая, что он указывает на последний узел, мы вставляем новый узел за ним, после чего перемещаем указатель на одну позицию вперед – на новый узел, который теперь стал последним.

Листинг 3.28. Метод **Enqueue** класса **TtdQueue**

```

procedure TtdQueue.Enqueue(aItem : pointer);
var
    Temp : PslNode;
begin
    Temp := PslNode(SLNodeManager.AllocNode);
    Temp^.slnData := aItem;
    Temp^.slnNext := nil;
    {добавить новый узел в конец списка и переместить указатель конечного узла на
    только что вставленный узел}
    FTail^.slnNext := Temp;
    FTail := Temp;
    inc(FCount);
end;

```

Метод **Dequeue** ничуть не сложнее. Сначала он проверяет список на наличие в нем элементов, а затем, пользуясь алгоритмом "удалить после" фиктивного начального узла **FHead**, удаляет из списка первый узел. Перед освобождением узла с помощью диспетчера узлов метод **Dequeue** возвращает данные. После выполнения метода количество элементов в списке уменьшается на единицу. Вот здесь и начи-

нается самое интересное. Представьте себе, что из очереди снимается один единственный имеющийся в ней элемент. До выполнения операции **Dequeue** указатель **FTail** указывал на последний узел списка, который был одновременно и первым. После снятия элемента с очереди первый узел будет отсутствовать, но указатель **FTail** все еще указывает на него. Нам нужно сделать так, чтобы после удаления узла в списке **FTail** указывал на фиктивный начальный элемент. Если же в списке до удаления присутствовало несколько элементов, указатель будет указывать на действительный последний узел.

Листинг 3.29. Метод **Dequeue** класса **TtdQueue**

```
function TtdQueue.Dequeue : pointer;
var
  Temp : PslNode;
begin
  if (Count = 0) then
    qError(tdeQueueIsEmpty, 'Dequeue');
  Temp := FHead^.slnNext;
  Result := Temp^.slnData;
  FHead^.slnNext := Temp^.slnNext;
  SLNodeManager.FreeNode(Temp);
  dec(FCount);
  {если после удаления элемента очередь опустела, переместить указатель после-
   днего элемента на фиктивный начальный узел}
  if (Count = 0) then
    FTail := FHead;
end;
```

Остальные методы, **Clear**, **Examine** и **IsEmpty**, еще проще.

Листинг 3.30. Методы **Clear**, **Examine** и **IsEmpty** класса **TtdQueue**

```
procedure TtdQueue.Clear;
var
  Temp : PslNode;
begin
  {удалить все узлы за исключением начального; при возможности освободить все
   данные}
  Temp := FHead^.slnNext;
  while (Temp <> nil) do begin
    FHead^.slnNext := Temp^.slnNext;
    if Assigned(FDispose) then
      FDispose(Temp^.slnData);
    SLNodeManager.FreeNode(Temp);
    Temp := FHead^.slnNext;
  end;
  FCount := 0;
```

```

    {теперь очередь пуста, установить указатель последнего элемента на начальный
      узел}
    FTail := FHead;
end;

function TtdQueue.Examine : pointer;
begin
    if (Count = 0) then
        qError(tdeQueueIsEmpty, 'Examine');
    Result := FHead^.slnNext^.slnData;
end;

function TtdQueue.IsEmpty : boolean;
begin
    Result := (Count = 0);
end;

```

Полный код класса **TtdQueue** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStkQue.pas**.

Очереди на основе массивов

А теперь давайте рассмотрим реализацию очереди на основе массива. Как и раньше, для простоты воспользуемся массивом **TList**. По крайней мере, в этом случае нам не придется беспокоиться о распределении памяти и увеличении размера массива.

Зная, как реализуется очередь на основе связанного списка, первым желанием может быть, для имитации операции постановки в очередь, добавлять элементы в конец экземпляра массива **TList** с помощью метода **Add**, а для имитации снятия с очереди – удалять первый элемент с помощью метода **Delete** (или наоборот, вставлять в начало массива, а удалять с конца). Тем не менее, давайте посмотрим, что при этом будет происходить с массивом. При выполнении метода **Add** ничего интересного не происходит, за исключением тех случаев, когда приходится увеличивать размер массива. Это операция класса $O(1)$ – как раз то, что требуется. Что же касается **Delete**, то здесь все не так безоблачно. Для реализации операции снятия с очереди из массива **TList** потребуется удалить первый элемент, что приведет к тому, что все элементы массива переместятся на одну позицию вперед. Такая операция зависит от количества элементов в массиве, т.е. принадлежит к классу $O(n)$. Вот и дождались плохих новостей. Мы не можем поменять местами операции постановки в очередь и снятия с очереди, т.е. мы добавляем только в начало списка и удаляем с его конца. Другими словами, мы все равно получаем операцию класса $O(n)$ при добавлении в начало списка.

В некоторых источниках описанный принцип все же используется для реализации очереди. Более того, класс **TQueue** в модуле **Contnrs**, возможно, основан на таком принципе.

Каким образом можно реализовать очередь на основе массива, чтобы обе базовых операции принадлежали к классу $O(1)$?

Решение заключается в использовании *кольцевой очереди*. Представьте себе приемную у стоматолога. Как правило, это комната со стульями вдоль стен. В отличие от очереди в супермаркете, где вы подходите к началу очереди, толкая тележку, в приемной вы сидите на стуле. При вызове очередного пациента все остальные не встают и не переходят на соседний стул. Просто начало очереди – какой-то тяжело объяснимый атрибут (ага, такое впечатление, что в Америке нет очередей к стоматологу...) – переходит к другому человеку. При вызове пациента этот атрибут передается следующему пациенту, и он становится "началом" очереди. Таким образом, никто не встает со стульев, просто некоторым образом (возможно, с помощью ассистента стоматолога) определяется первый пациент в очереди. Подобного рода организация называется *круговой очередью*.

Для реализации круговой очереди на основе массива введем переменную, которая будет содержать индекс первого элемента в очереди. Кроме того, введем еще одну переменную, которая будет указывать на конец очереди. Начнем с массива с некоторым определенным количеством элементов (размер будем определять на основе максимально возможного количества элементов в очереди) и установим индекс начального элемента равным индексу конечного элемента. Фактически, это равенство означает, что очередь пуста.

Постановка элемента в очередь эквивалентна установке значения элемента, на который указывает индекс конца очереди, равным значению записываемого в очередь элемента. После этого значение индекса конца очереди нужно увеличить на 1. Если после увеличения индекса он будет превышать размер массива, необходимо установить его равным 0, т.е. индексу первого элемента.

Снятие элемента с очереди означает возврат значения элемента, на который указывает индекс начала очереди. После этого значение индекса начала очереди увеличивается на 1. Если после увеличения индекса он будет превышать размер массива, установить его равным 0. Очевидно, что перед снятием элемента с очереди необходимо убедиться, что очередь не пуста. Для этого следует проверить, равны ли индексы начала и конца очереди (в случае равенства индексов, очередь пуста).

И нам осталось рассмотреть еще одну проблему: при постановке элемента в очередь необходимо убедиться, что новое значение индекса конца очереди не равно значению индекса начала очереди. Если равенство соблюдается, значит, очередь полностью заполнена элементами. К сожалению, такая ситуация также озна-

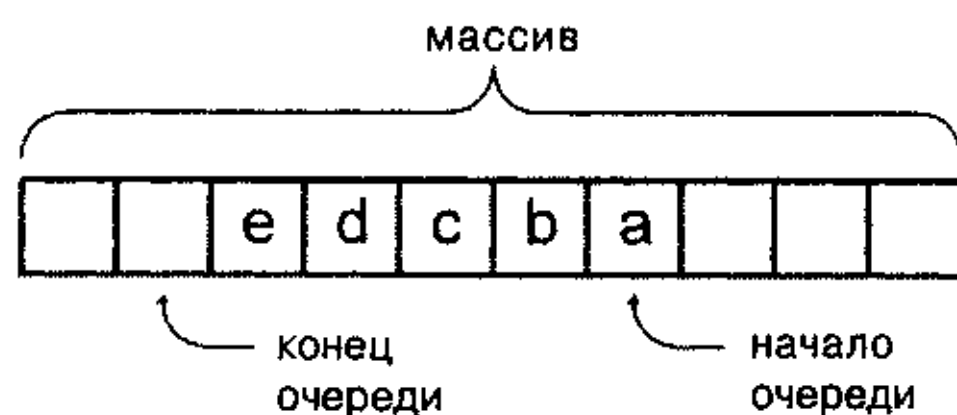


Рисунок 3.10. Использование массива для организации очереди

чает (по крайней мере, для процедуры снятия с очереди), что очередь пуста. Таким образом, если такая достаточно абсурдная ситуация возникает – пустая очередь эквивалентна заполненной – необходимо увеличить размер массива, перемещая все имеющиеся в массиве элементы и изменяя значения индексов начала и конца очереди.

Интерфейс класса **TtdArrayQueue** выглядит точно так же, как и интерфейс класса **TtdQueue**.

Листинг 3.31. Класс TtdArrayQueue

```

TtdArrayQueue = class
  private
    FCount : integer;
    FDispose : TtdDisposeProc;
    FHead : integer;
    FList : TList;
    FName : TtdNameString;
    FTail : integer;
  protected
    procedure aqError(aErrorCode : integer;
                     const aMethodName : TtdNameString);
    procedure aqGrow;
  public
    constructor Create(aDispose : TtdDisposeProc;
                      aCapacity : integer);
    destructor Destroy; override;
    procedure Clear;
    function Dequeue : pointer;
    procedure Enqueue(aItem : pointer);
    function Examine : pointer;
    function IsEmpty : boolean;
    property Count : integer read FCount;
    property Name : TtdNameString read FName write FName;
end;

```

Конструктор и деструктор мало чем отличаются от соответствующих методов класса **TtdArrayStack**.

Листинг 3.32. Конструктор и деструктор класса TtdArrayQueue

```

constructor TtdArrayQueue.Create( aDispose : TtdDisposeProc;
                                   aCapacity : integer);

begin
  inherited Create;
  {сохранить процедуру удаления}
  FDispose := aDispose;
  {создать внутренний массив TList и установить его размер равным aCapacity
   элементов}
  FList := TList.Create;

```



```

if (aCapacity <= 1) then
  aCapacity := 16;
  FList.Count := aCapacity;
end;

destructor TtdArrayQueue.Destroy;
begin
  FList.Free;
  inherited Destroy;
end;

```

Самое интересное происходит в методах **Enqueue** и **Dequeue**.

ЛИСТИНГ 3.33. Методы **Enqueue** и **Dequeue** класса **TtdArrayQueue**

```

function TtdArrayQueue.Dequeue : pointer;
begin
  {убедиться, что очередь не пуста}
  if (Count = 0) then
    aqError(tdeQueueIsEmpty, 'Dequeue');
  {элемент, снимаемый с очереди, находится в ее начале}
  Result := FList[FHead];
  {переместить индекс начала очереди и убедиться, что он все еще действителен;
   уменьшить количество элементов на 1}
  FHead := (FHead + 1) mod FList.Count;
  dec(FCount);
end;

procedure TtdArrayQueue.Enqueue(aItem : pointer);
begin
  {добавить элемент в конец очереди}
  FList[FTail] := aItem;
  {переместить индекс конца очереди и убедиться, что он все еще действителен;
   увеличить количество элементов на 1}
  FTail := (FTail + 1) mod FList.Count;
  inc(FCount);
  {если после добавления очередного элемента мы обнаруживаем, что значения
   индексов начала и конца очереди равны, увеличить размер массива}
  if (FTail = FHead) then
    aqGrow;
end;

```

Как видите, снятие элемента с очереди включает возврат элемента, находящегося в позиции с индексом начала очереди, а затем увеличение индекса на 1. Постановка в очередь включает запись элемента в позицию с индексом конца очереди и увеличение индекса на 1. Если конец очереди достигает ее начала, размер массива увеличивается с помощью метода **aqGrow**.

Листинг 3.34. Расширение размера экземпляра класса `TtdArrayQueue`

```
procedure TtdArrayQueue.aqGrow;  
var  
    i : integer;  
    ToInx : integer;  
begin  
    {увеличить размер списка}  
    FList.Count := (FList.Count * 3) div 2;  
    {теперь элементы находятся в конце списка, необходимо восстановить  
    корректный порядок элементов в кольцевой очереди}  
    if (FHead = 0) then  
        FTail := FCount  
    else begin  
        ToInx := FList.Count;  
        for i := pred(Count) downto FHead do begin  
            dec(ToInx);  
            FList[ToInx] := FList[i];  
        end;  
        FHead := ToInx;  
    end;  
end;
```

Приведенный метод является наиболее сложным методом во всем классе. При его вызове очередь заполнена, индекс конца очереди временно равен индексу начала (не забывают, что это также означает, что очередь пуста), причем необходимо увеличить размер массива **TList**. Первое, что мы делаем, – увеличиваем размер массива на 50%. После этого нужно исправить кольцевую очередь таким образом, чтобы она правильно учитывала свободное место. Если значение индекса начала очереди равно 0, кольцевая очередь была не круговой, и все что требуется сделать – изменить значение индекса конца очереди. Если же значение индекса начала не равно 0, очередь была "закольцована" внутри массива. Чтобы переходить по элементам в правильном порядке, мы начинаем с индекса начала очереди, доходим до старого конца массива, переходим к началу массива и идем до индекса конца очереди (который равен индексу начала очереди). Теперь у нас имеются дополнительные элементы, которые находятся между старым и новым концом массива. Следовательно, мы должны поместить элементы, находящиеся между началом очереди и старым концом массива таким образом, чтобы они занимали место до нового конца массива. После этого мы получим правильный порядок элементов в кольцевой очереди.

Полный код класса **TtdArrayQueue** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStkQue.pas**.

Резюме

Эта глава была посвящена связным спискам: как односвязным, так и двухсвязным. Были описаны некоторые проблемы, касающиеся работы стандартного связного списка, и показано, что использование диспетчера узлов повысило быстродействие обеих версий списков. В конце главы мы рассмотрели стеки и очереди и реализовали их на основе связных списков и массивов.

После изучения стеков и очередей, основанных на связных списках и массивах, вы, наверное, задали себе вопрос: "Какой тип лучше использовать?" Тесты на контроль времени в различных версиях Delphi (16- и 32-разрядных) показали, что в большинстве случаев быстрее оказывается версия, основанная на массиве. Ее и лучше использовать. Исключением является случай, когда в Delphi 1 количество элементов в стеке или очереди превышает 16000 – это максимальное значение для Delphi 1. Поэтому, если в вашем стеке или очереди будет больше 16000 элементов, в Delphi 1 потребуется работать со связными списками.



Глава 4

Поиск

Поиск – это действие, заключающееся в просмотре набора элементов и выделении из этого набора интересующего элемента. Наверное, все вы знакомы с одной из функций поиска – **Pos** из модуля **SysUtils**, которая предназначена для поиска подстроки в строке.

Эта и следующая главы, посвященные поиску, довольно-таки тесно связаны между собой. Часто поиск элемента приходится осуществлять в уже отсортированном контейнере. И если контейнер отсортирован, можно воспользоваться эффективным алгоритмом для поиска позиции вставки нового элемента, чтобы и после вставки контейнер оказался отсортированным. Тем не менее, поиск не ограничивается просмотром отсортированных списков. Мы, помимо прочих, рассмотрим простейший тип поиска – алгоритмы, которые кажутся почти очевидными и не заслуживают специального названия.

Кроме того, настоящая глава служит мостом между простыми фундаментальными контейнерами, массивами и связными списками, и более сложными, например, бинарными деревьями, списками пропусков и хеш-таблицами. Эффективный поиск зависит от сложности контейнера, в котором находятся элементы, поэтому мы приводим алгоритмы как для массивов, так и для связных списков. В последующих главах при рассмотрении более сложных контейнеров мы всегда будем говорить об оптимальной стратегии поиска для обсуждаемых структур данных.

Процедуры сравнения

Само действие поиска элемента в наборе элементов требует возможности отличать элементы друг от друга. Если мы не можем различить два элемента, то не

имеет смысла искать один из таких элементов. Таким образом, первая трудность, которую нам потребуется преодолеть, – это *сравнение* двух элементов, находящихся в одном наборе. Существует два типа сравнения. Первый из них предназначен для несортированных списков элементов, когда все, что нам нужно знать, так это равны ли два элемента. Вторым тип используется в отсортированных списках элементов, когда можно добиться повышения эффективности поиска, если имеется возможность определить отношение одного элемента к другому (меньше, равен или больше). (Фактически, операция сравнения определяет, в каком порядке элементы находятся в списке. При поиске в отсортированном списке необходимо выполнять то же самое сравнение, на основе которого был построен список.)

Очевидно, что если элементы принадлежат к целочисленному типу, операция сравнения не представляет никаких трудностей: все мы можем взять два целых числа и определить, отличаются они или нет. В случае строк сравнение усложняется. Можно выполнять сравнение, чувствительное к регистру (т.е. строчные символы будут отличаться от прописных), и сравнение, нечувствительное к регистру (т.е. строчные символы не будут отличаться от прописных), сравнение по локальным таблицам символов (сравнение на основе алгоритмов, специфических для определенной страны или языка) и т.д. Тип **set** в Delphi, несмотря на то, что он позволяет сравнивать два набора, все же не имеет четко определенного способа определения того, что один набор больше другого (фактически выражение "один набор больше другого" не имеет смысла, если речь не идет о количестве элементов). А что касается объектов, то здесь даже нет метода, который бы позволил сказать, что объект А равен или не равен объекту В (за исключением сравнения указателей на объекты).

Лучше всего на данном этапе рассматривать процедуру сравнения в виде "черного ящика" – функции с четко определенным интерфейсом или синтаксисом, которая в качестве входного параметра принимает два элемента и возвращает результат сравнения – первый элемент меньше второго, первый элемент равен второму или первый элемент больше второго. Для тех типов элементов, которые не имеют определенного порядка (т.е. даже если известно, что два элемента не равны, мы не можем определить, меньше элемент А элемента В или больше), нужно предусмотреть, чтобы функция сравнения возвращала значение, которое трактуется как "не равно".

В книге все функции сравнения принадлежат к типу **TtdCompareFunc** (этот тип объявлен в файле **TDBasics.pas**, который можно найти на Web-сайте издательства, в разделе материалов; там же находятся и примеры функций сравнения):

Листинг 4.1. Прототип функции **TtdCompareFunc**

type

```
TtdCompareFunc = function(aData1, aData2 : pointer) : integer;
```

Другими словами, функция сравнения в качестве входных параметров принимает два указателя и возвращает целочисленное значение. Возвращаемое значение будет равно 0, если два сравниваемых элемента *равны*, меньше нуля, если первый элемент *меньше* второго, и больше нуля, если первый элемент *больше* второго. Тип параметров **aData1** и **aData2** определяет сама функция, и она же решает, что делать с переданными данными: привести к определенному классу или просто к типу, который не является указателем.

Приведем пример функции сравнения, которая предполагает, что входные параметры принадлежат к типу **longint**, а не представляют собой указатели. (Будем считать, что значение **sizeof(longint)** равно **sizeof(pointer)**. На сегодняшний день это справедливо для всех версий Delphi.)

Листинг 4.2. Функция **TDCompareLongint**

```
function TDCompareLongint(aData1, aData2 : pointer) : integer;
var
  L1 : longint absolute aData1;
  L2 : longint absolute aData2;
begin
  if (L1 < L2) then
    Result := -1
  else if (L1 = L2) then
    Result := 0
  else
    Result := 1
end;
```

Перед тем как в ужасе сказать, что вы бы никогда не вызвали такую функцию сравнения двух значений типа **longint**, обратите внимание, что этого и не требуется. Приведенная функция предназначена для использования структурами данных, которые принимают элементы в виде указателей (например, список **TtdSingleLinkList** или стандартный массив **TList**), и подпрограммами, которые используют такие структуры данных. Если вы разрабатываете функцию поиска, исходя из главных принципов, имеет смысл написать и процедуру сравнения. Остается надеяться, что все мы сможем написать функцию для сравнения двух целых чисел.

Давайте рассмотрим пример функции **TDCompareNullStr**, предназначенной для сравнения двух строк, завершающихся нулем, не привязываясь к алфавиту определенной страны:

Листинг 4.3. Функция **TDCompareNullStr**

```
function TDCompareNullStr(aData1, aData2 : pointer) : integer;
begin
  Result := StrComp(PAnsiChar(aData1), PAnsiChar(aData2));
end;
```

(В Delphi 1 в модуле **TDBasics** объявлено, что тип **PAnsiChar** соответствует типу **PChar**.) К счастью, для данного примера стандартная функция **StrComp** возвращает значение того же типа, что и требуется для нашей функции сравнения.

В качестве последнего примера приведем функцию **TDCompareNullStrANSI**, предназначенную для сравнения двух строк, завершающихся нулем, с учетом локальных таблиц символов:

Листинг 4.4. Функция **TDCompareNullStrANSI**

```
function TDCompareNullStrANSI(aData1, aData2 : pointer) : integer;  
begin  
    {$IFDEF Delphi1}  
    Result := lstrcmp(PAnsiChar(aData1), PAnsiChar(aData2));  
    {$ENDIF}  
    {$IFDEF Delphi2Plus}  
    Result := CompareString(LOCALE_USER_DEFAULT, 0,  
                           PAnsiChar(aData1), -1,  
                           PAnsiChar(aData2), -1) - 2;  
    {$ENDIF}  
    {$IFDEF Kylix1Plus}  
    Result := strcoll(PAnsiChar(aData1), PAnsiChar(aData2));  
    {$ENDIF}  
end;
```

В приведенной функции для Delphi 1 и 32-разрядных версий Delphi используются разные коды. Кроме того, обратите внимание, что функция **lstrcmp** возвращает значения в том виде, который нужен нам. К сожалению, функция **CompareString** этого не делает. Она возвращает 1, если первая строка меньше второй, 2, если строки равны, и 3, если первая строка больше второй. Поэтому для получения требуемого значения необходимо просто вычесть 2 из результата, возвращаемого функцией **CompareString**. В Kylix для сравнения строк нужно воспользоваться функцией **strcoll** из модуля **Libc**.

Последовательный поиск

Теперь, когда мы определились с функцией сравнения, можно перейти к рассмотрению алгоритмов поиска элемента в массивах и связанных списках.

Массивы

Массивы представляют собой простейшую реализацию набора элементов, для которой можно использовать алгоритм последовательного поиска. Возможны два случая: первый – элементы массива расположены в произвольном порядке и второй – элементы отсортированы. Сначала рассмотрим случай несортированного массива.

Если массив не отсортирован, для поиска определенного элемента может использоваться только один единственный алгоритм: выбирать каждый элемент мас-

сива и сравнивать его с искомым. Как правило, такой алгоритм реализуется с помощью цикла **For**. В качестве примера давайте выполним поиск значения 42 в массиве из 100 целых чисел:

```
var
  MyArray : array[0..99] of integer;
  Inx : integer;
begin
  for Inx := 0 to 99 do
    if MyArray[Inx] = 42 then
      Break;
  if (Inx = 100) then
    .. значение 42 не было найдено ..
  else
    .. значение 42 было найдено в элементе с индексом Inx ..
```

Довольно просто, не правда ли? Код выполняет цикл по всем элементам массива, начиная с первого и заканчивая последним, используя **Break** для выхода из цикла при обнаружении первого элемента, значение которого равно искомому 42. (Оператор **Break** очень удобно использовать, здесь он ничем не отличается от оператора **goto**.) После цикла, для того чтобы определить, найден ли элемент, проверяется значение счетчика цикла **Inx**.

Интересно, сколько читателей в приведенном выше коде нашли ошибку? Проблема заключается в том, что в языке Object Pascal при успешном завершении цикла значение переменной цикла будет не определено. С другой стороны, в случае преждевременного завершения цикла, скажем, с помощью оператора **Break**, значение переменной цикла будет определено.

В коде предполагается, что переменная цикла **Inx** после завершения цикла будет на 1 больше конечного значения для цикла **For**, даже если цикл будет выполнен успешно. Оказывается, что в 32-разрядных компиляторах (в версиях Delphi от 2 до 7) ошибки не возникает: значение переменной цикла после завершения цикла будет на 1 больше, чем при последнем выполнении цикла. В Delphi 1 код будет работать неправильно: после завершения выполнения цикла переменная цикла будет содержать значение, равное своему значению при последнем выполнении цикла (в нашем примере **Inx** после полного выполнения цикла будет содержать 99). Кто знает, что будет в следующих версиях Delphi? Вполне возможно, что в будущих версиях Delphi будет изменен оптимизатор компилятора, и переменная цикла после завершения цикла будет получать другое значение. В конце концов, разработчики, описав поведение переменной цикла, оставили за собой право изменения ее значения после выхода из цикла.

Тогда каким образом можно реализовать алгоритм последовательного поиска? Цикл **For** можно использовать (это самый быстрый метод организации последовательного поиска), однако потребуется ввести флаг, который будет указывать, найден ли искомый элемент. Код несколько усложнится, но зато становится корректным с точки зрения языка программирования:

```
var
  MyArray : array[0..99] of integer;
  Inx      : integer;
  FoundIt : boolean;
begin
  FoundIt := false;
  for Inx := 0 to 99 do
    if MyArray[Inx] = 42 then begin
      FoundIt := true;
      Break;
    end;
  if not FoundIt then
    .. значение 42 не было найдено ..
  else
    .. значение 42 было найдено в элементе с индексом Inx ..
```

А теперь рассмотрим функцию поиска элемента в массиве **TList** с помощью функции сравнения (ее реализацию можно найти в файле **TDList.pas** на Web-сайте издательства, в разделе сопровождающих материалов). Если искомый элемент не найден, функция возвращает -1, в противном случае возвращается индекс элемента.

Листинг 4.5. Последовательный поиск в несортированном массиве TList

```
function TDListIndexOf(aList : TList; aItem : pointer;
                      aCompare : TtdCompareFunc) : integer;

var
  Inx : integer;
begin
  for Inx := 0 to pred(aList.Count) do
    if (aCompare(aList.List^[Inx], aItem) = 0) then begin
      Result := Inx;
      Exit;
    end;
  {если мы попали сюда, значит искомый элемент не найден}
  Result := -1;
end;
```

Эта функция работает не так как метод **TList.IndexOf**, который предназначен для поиска элемента в массиве путем сравнения значений указателей. Фактически он в своем внутреннем списке указателей осуществляет поиск элемента как указателя. С другой стороны, функция **TDListIndexOf** осуществляет поиск самого элемента, вызывая для сравнения искомого и текущего элемента функцию сравнения. Функция сравнения может сравнивать просто значения указателей или преобразовывать указатели во что-нибудь более значимое, например, в класс или запись, а затем сравнивать поля.

Обратите внимание, что в реализации функции с целью повышения эффективности применяется небольшая хитрость. Вместо сравнения **aItem** с **aList[Index]** выполняется сравнение с **aList.List^[Index]**. Зачем? Компилятор преобразовывает первое сравнение в вызов функции, а затем вызываемая функция, **TList.Get**, перед возвратом указателя из внутреннего массива указателей проверяет переданный ей индекс на предмет попадания в диапазон от 0 до количества элементов (вызывая исключение, если условие не соблюдается). Но мы знаем, что индекс находится в требуемом диапазоне, поскольку используется цикл от 0 до количества элементов минус 1. Поэтому нам не нужно считывать значение свойства **Items** и вызывать метод **TList.Get**. Можно получить доступ непосредственно к массиву указателей (свойство **List** экземпляра **TList**).

Эта хитрость (использование свойства **List** экземпляра **TList**) вполне корректна. Если вы уверены, что значения индекса не выходят за пределы допустимого диапазона, можно исключить проверку на предмет попадания в диапазон за счет непосредственного доступа к массиву **List**, а не с помощью свойства **Items**. Тем не менее, ее применение при итерации по массиву **TList** или в коде, который может привести к выходу индекса за пределы допустимого диапазона, не желательно. Лучше обезопасить себя, нежели потом сожалеть.

В классе **TtdRecordList** (который описан в главе 2) для организации последовательного поиска можно пользоваться методом **IndexOf** (см. листинг 4.6).

Листинг 4.6. Последовательный поиск с помощью метода **TtdRecordList.IndexOf**

```
function TtdRecordList.IndexOf(aItem : pointer;
                             aCompare : TtdCompareFunc) : integer;
var
  ElementPtr : PAnsiChar;
  i : integer;
begin
  ElementPtr := FArray;
  for i := 0 to pred(Count) do begin
    if (aCompare(aItem, ElementPtr) = 0) then begin
      Result := i;
      Exit;
    end;
    inc(ElementPtr, FElementSize);
  end;
  Result := -1;
end;
```

Как видите, время выполнения алгоритма последовательного поиска напрямую зависит от количества элементов в массиве. В лучшем случае мы можем найти требуемый элемент с первой попытки (если он будет первым в массиве), но вполне вероятно, что мы обнаружим его в самом конце, после просмотра всех элементов.

В среднем для массива размером n для обнаружения искомого элемента придется пройти $n/2$ элементов. В любом случае, если искомого элемента нет в массиве, будут просмотрены все n элементов. Таким образом, операция последовательного поиска принадлежит к классу $O(n)$.

А что можно сказать о сортированном массиве? Первое, что следует отметить, – простой алгоритм последовательного поиска в отсортированном массиве будет работать ничуть не хуже (или не лучше, в зависимости от вашей точки зрения), чем в несортированном. Операция поиска будет принадлежать к классу $O(n)$.

Тем не менее, алгоритм поиска можно улучшить. Если искомого элемента нет в массиве, поиск можно выполнить намного быстрее. Фактически мы выполняем итерации по массиву, как и раньше, но теперь только до тех пор, пока не будет найден элемент, больший или равный искомому. Если обнаружен элемент, равный искомому, поиск завершается успешно. Если же обнаружен элемент больше искомого, значит, искомый элемент в массиве отсутствует, поскольку массив отсортирован, а мы дошли до элемента большего, чем искомый. Все последующие элементы также будут больше искомого. Следовательно, поиск можно прекратить.

Листинг 4.7. Последовательный поиск в отсортированном массиве TList

```
function TDTListSortedIndexOf(aList : TList; aItem : pointer;
                             aCompare : TtdCompareFunc) : integer;

var
  Inx, CompareResult : integer;
begin
  {искать первый элемент больший или равный элементу aItem}
  for Inx := 0 to pred(aList.Count) do begin
    CompareResult := aCompare(aList.List^[Inx], aItem);
    if (CompareResult >= 0) then begin
      if (CompareResult = 0) then
        Result := Inx
      else
        Result := -1;
      Exit;
    end;
  end;
  {если мы попали сюда, значит искомый элемент не найден}
  Result := -1;
end;
```

Обратите внимание, что функция сравнения вызывается только один раз при каждом выполнении цикла. Мы не знаем, что делает функция **aCompare** – для нас это "черный ящик". Следовательно, желательно ее вызывать как можно реже. Поэтому при каждом выполнении цикла мы вызываем ее только один раз и сохраняем полученный результат в переменной целого типа. После этого переменную можно использовать сколько угодно раз, не вызывая функцию.

Как уже говорилось, приведенная функция поиска нисколько не увеличивает скорость обнаружения искомого элемента, если искомый элемент присутствует в массиве (в среднем, как и ранее, для этого потребуется провести $n/2$ сравнений). Единственным ее преимуществом перед предыдущей функцией является то, что при отсутствии искомого элемента в массиве результат будет получен быстрее. Скоро мы рассмотрим алгоритм бинарного поиска, который позволит повысить быстродействие в обоих случаях.

Связные списки

В связанных списках последовательный поиск выполняется точно так же, как и в массивах. Тем не менее, элементы проходятся не по индексу, а по указателю **Next**. Для класса **TtdSingleLinkedList**, описанного в главе 3, можно разработать две следующих функции: первая – для выполнения поиска по несортированному связному списку, и вторая – по отсортированному. Функции просто указывают, найден ли искомый элемент. В случае, если элемент найден, список будет установлен в позицию искомого элемента. В функции для отсортированного списка курсор будет установлен в позицию, где должен находиться искомый элемент, чтобы список оставался отсортированным.

Листинг 4.8. Последовательный поиск в однонаправленном связном списке

```
function TDSLLSearch(aList : TtdSingleLinkedList;
                    aItem : pointer;
                    aCompare : TtdCompareFunc) : boolean;
begin
  with aList do begin
    MoveBeforeFirst;
    MoveNext;
    while not IsAfterLast do begin
      if (aCompare(Examine, aItem) = 0) then begin
        Result := true;
        Exit;
      end;
      MoveNext;
    end;
  end;
  Result := false;
end;

function TDSLLSortedSearch(aList : TtdSingleLinkedList;
                           aItem : pointer;
                           aCompare : TtdCompareFunc) : boolean;
var
  CompareResult : integer;
begin
  with aList do begin
    MoveBeforeFirst;
    MoveNext;
```



```
while not IsAfterLast do begin
  CompareResult := aCompare(Examine, aItem);
  if (CompareResult >= 0) then begin
    Result := (CompareResult = 0);
    Exit;
  end;
  MoveNext;
end;
end;
Result := false;
end;
```

Соответствующие функции для класса **TtdDoubleLinkedList** будут точно такими же.

Бинарный поиск

В случае отсортированного списка можно использовать более эффективный алгоритм бинарного поиска. Сначала рассмотрим его на примере массива, а затем покажем, как его изменить для связанных списков.

Алгоритм бинарного поиска применим только для отсортированных контейнеров.

Массивы

Предположим, что у нас имеется отсортированный массив. Как было показано ранее, алгоритм последовательного поиска даже при использовании выхода из цикла в случае отсутствия в списке искомого элемента принадлежит к классу $O(n)$. Каким образом можно улучшить быстродействие?

Ответом может служить бинарный поиск. Он основан на стратегии "разделяй и властвуй": начинаем с большой проблемы, разбиваем ее на маленькие проблемы, которые легче решить, а, затем, следовательно, решаем всю большую проблему.

Бинарный поиск работает следующим образом. Берем средний элемент массива. Равен ли он искомому элементу? Если да, то поиск успешно завершен. В противном случае, если искомый элемент меньше среднего, то можно сказать, что, если элемент присутствует в массиве, он находится в первой половине. С другой стороны, если искомый элемент больше среднего, он должен находиться во второй половине. Таким образом, одним сравнением мы разбили нашу проблему на две части. Теперь мы применяем тот же алгоритм к выбранной части массива: находим средний элемент и определяем, в какой половине (точнее уже в четвертой части) находится искомый элемент. Мы снова делим проблему на две части. Описанные операции продолжаются до тех пор, пока искомый элемент не будет найден (разумеется, если он присутствует в массиве).

Это и есть алгоритм бинарного поиска. Поскольку размер массива при каждом выполнении цикла уменьшается в два раза, быстродействие алгоритма будет вы-

ражаться как $O(\log(n))$, т.е. скорость работы алгоритма примерно пропорциональна функции двоичного логарифма $\log_2$ от количества элементов в массиве (таким образом, возведение количества элементов массива во вторую степень приведет к увеличению времени поиска только в два раза).

Ниже приведен пример выполнения бинарного поиска в массиве **TList** (функцию можно найти в файле **TDList.pas** на Web-сайте издательства, в разделе сопровождающих материалов).

Листинг 4.9. Бинарный поиск в отсортированном массиве TList

```
function TDListSortedIndexOf(aList : TList; aItem : pointer;
                             aCompare : TtdCompareFunc) : integer;

var
  L, R, M : integer;
  CompareResult : integer;
begin
  { задать значения для индексов первого и последнего элементов }
  L := 0;
  R := pred(aList.Count);
  while (L <= R) do begin
    { вычислить индекс среднего элемента }
    M := (L + R) div 2;
    { сравнить значение среднего элемента с искомым значением }
    CompareResult := aCompare(aList.List^[M], aItem);
    { если значение среднего элемента меньше искомого значения, переместить
      левый индекс на позицию до среднего индекса }
    if (CompareResult < 0) then
      L := succ(M)
    { если значение среднего элемента больше искомого значения, переместить
      правый индекс на позицию после среднего индекса }
    else if (CompareResult > 0) then
      R := pred(M)
    { в противном случае искомый элемент найден }
    else begin
      Result := M;
      Exit;
    end;
  end;
  Result := -1;
end;
```

Для описания подмассива, рассматриваемого в текущий момент, используются две переменных — **L** и **R**, которые хранят, соответственно, левый и правый индексы. Первоначально значения этих переменных устанавливаются равными 0 (первый элемент массива) и **Count-1** (последний элемент массива). Затем мы входим в цикл **While**, из которого выйдем после обнаружения в массиве искомого элемента или когда значение переменной **L** превысит значение переменной **R**, что означает, что искомый элемент в массиве отсутствует. При каждом выполнении цикла вы-

числяется индекс среднего элемента (фактически это среднее значение между L и R). Затем значение элемента со средним индексом сравнивается с искомым значением. Если значение среднего элемента меньше, чем искомое, мы переносим левый индекс на позицию после среднего. В противном случае мы переносим правый индекс на позицию перед средним. Таким образом, мы определяем новый подмассив для поиска. Если же значение среднего элемента равно искомому, поиск завершен.

Для примера на рис. 4.1 приведены шаги, выполняемые при бинарном поиске буквы d в отсортированном массиве, содержащем буквы от a до k . На шаге (а) переменная L указывает на первый элемент (индекс 0), а R – на последний (индекс 10). Это означает, что значение переменной M будет составлять 5. Далее мы выполняем сравнение: значение элемента с индексом 5 равно f , а это больше искомого значения d .

Согласно алгоритму, мы устанавливаем значение R равным $M-1$ (таким образом, правая граница подмассива теперь находится слева от среднего элемента). Это означает, что значение R теперь равно 4. Новое значение среднего индекса будет равно 2, как показано на шаге (b). Выполняем сравнение: буква c (значение элемента с индексом 2) меньше, чем d .

Теперь, в соответствии с алгоритмом, необходимо установить индекс L за индексом M (т.е. $M+1$ или 3). Новое значение переменной M на шаге (c) равно 3. Выполняем сравнение: элемент с индексом 3 содержит букву d , а это и есть наше искомое значение. Поиск завершен.

Связные списки

Изучая код листинга 4.9, можно прийти к выводу, что маловероятно, чтобы бинарный поиск использовался для связанных списков, если, конечно, не воспользоваться индексным доступом к элементам списка, который, как уже упоминалось в главе 3, приводит к снижению быстродействия.

Но, тем не менее, реализация бинарного поиска для связанных списков оказывается не такой уж и неразрешимой проблемой. Во-первых, нужно понимать, что в общем случае переход по ссылке выполняется гораздо быстрее, нежели вызов функции сравнения. Следовательно, можно сказать, что переход по ссылке – это "хорошо", а вызов функции сравнения – "плохо". Это означает, что следует стремиться к минимизации вызовов функции сравнения. (Поскольку для нас функция

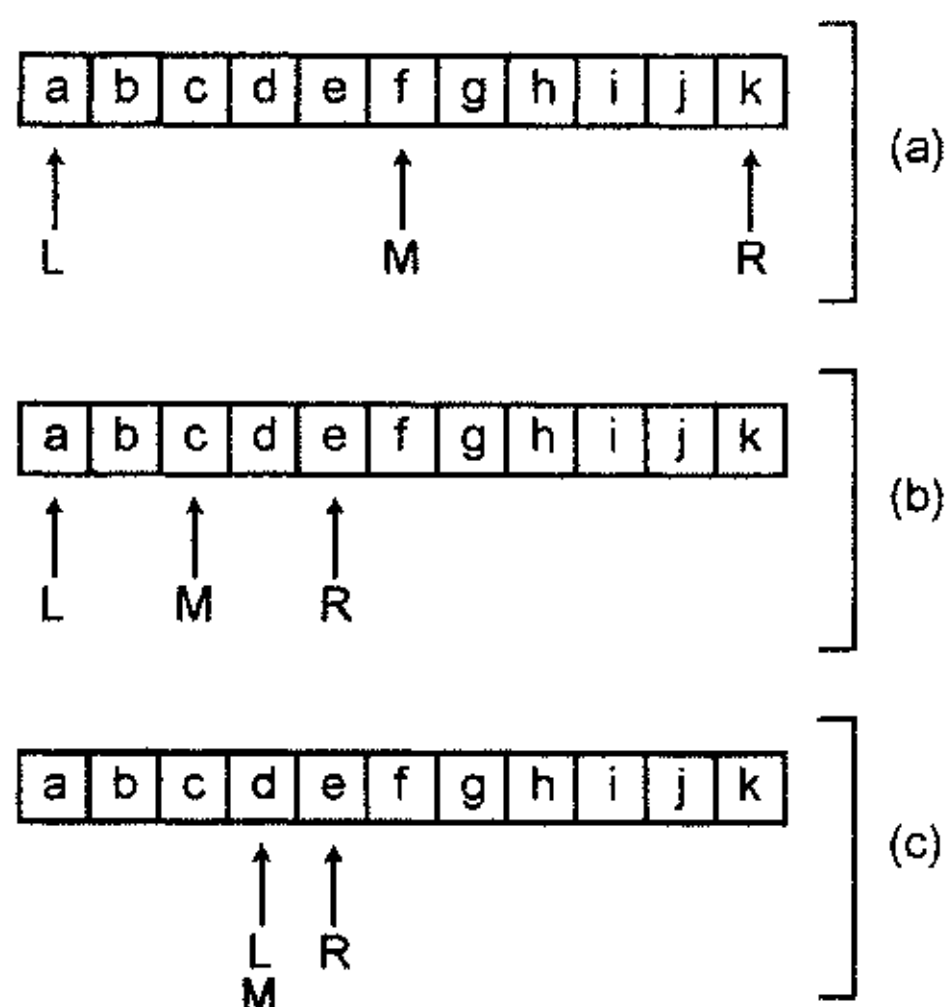


Рисунок 4.1. Бинарный поиск в массиве

сравнения – "черный ящик", мы не можем сказать, сколько времени требуется на ее выполнение: много или мало, по крайней мере, по сравнению со временем, требуемым на переход по ссылке.) Во-вторых, необходимо иметь доступ к "внутренностям" связного списка.

Давайте рассмотрим принцип организации бинарного поиска на примере обобщенного связного списка, а затем рассмотрим код для классов **TtdSingleLinkedList** и **TtdDoubleLinkedList**. Для нашего обобщенного связного списка должно быть известно количество содержащихся в нем элементов, поскольку оно понадобится при реализации алгоритма бинарного поиска. Кроме того, будем считать, что связный список содержит фиктивный начальный узел.

А теперь сам алгоритм.

1. Сохранить фиктивный начальный узел в переменной **BeforeCount**.
2. Сохранить количество элементов в списке в переменной **ListCount**.
3. Если значение **ListCount** равно нулю, искомого элемента нет в списке, и поиск завершается. В противном случае вычислить половину значения **ListCount**, при необходимости округлить его и сохранить в переменной **MidPoint**.
4. Переместить **BeforeCount** по ссылкам **Next** на **MidPoint** узлов.
5. Сравнить значение элемента в узле, где остановилась переменная **BeforeCount**, с искомым значением. Если значения равны, искомый элемент найден и поиск завершается.
6. Если значение в узле меньше, чем искомое, записать узел в переменную **BeforeCount**, вычесть значение **MidPoint** из значения **ListCount** и перейти к шагу 3.
7. Если значение в узле больше, чем искомое, записать значение **MidPoint-1** в переменную **ListCount** и перейти к шагу 3.

Давайте рассмотрим работу этого алгоритма на примере. Предположим, что имеется следующий связный список из пяти узлов, в котором необходимо найти узел В:

Начальный узел → A → B → C → D → E → nil

На первом шаге переменной **BeforeList** присваивается значение начального узла, а на втором переменной **ListCount** присваивается значение 5. Делим **ListCount** на два, округляем до целого, и присваиваем полученное значение (3) переменной **MidPoint** (шаг 3). По ссылкам от узла **BeforeList** отсчитываем три узла: A, B, C (шаг 4). Сравниваем текущий узел с искомым (шаг 5). Его значение больше искомого B, следовательно, устанавливаем значение переменной **ListCount** равным 2 (шаг 7). Еще раз выполняем цикл. Делим **ListCount** на два, округляем до целого и получаем 1 (шаг 3). По ссылкам от узла **BeforeList** отсчи-

тываем один узел: А (шаг 4). Сравниваем значение текущего узла с искомым значением (шаг 5). Оно меньше значения В, следовательно, записываем в **BeforeList** значение узла В, а переменной **ListCount** присваиваем значение 1 (шаг 6) и снова выполняем цикл. В этот раз **MidPoint** получит значение 1 (т.е. значение **ListCount**, деленное на два и округленное до целого). Переходим по ссылке от узла **BeforeList** на один шаг и находим искомый узел.

Если вы считаете, что в процессе выполнения алгоритма искомый узел был пройден несколько раз, то вы совершенно правы. Но следует иметь в виду, что вызов функции сравнения может быть намного медленнее, чем переход по ссылкам (например, если элементы списка представляют собой строки длиной 1000 символов, то для определения соотношения между строками функции сравнения придется сравнить в среднем 500 символов). Если бы связный список содержал целые числа, а мы отказались бы от частого использования функции сравнения, то быстрее всех оказался бы алгоритм последовательного поиска.

Ниже приведена функция бинарного поиска для класса **TtdSingleLinkedList**.

Листинг 4.10. Бинарный поиск в отсортированном однонаправленном связном списке

```
function TtdSingleLinkedList.SortedFind(aItem : pointer;
                                         aCompare : TtdCompareFunc) : boolean;
var
  BLCursor : PslNode;
  BLCursorIx : longint;
  WorkCursor : PslNode;
  WorkParent : PslNode;
  WorkCursorIx : longint;
  ListCount : longint;
  MidPoint : longint;
  i : integer;
  CompareResult : integer;
begin
  {подготовительные операции}
  BLCursor := FHead;
  BLCursorIx := -1;
  ListCount := Count;
  {пока в списке имеются узлы..}
  while (ListCount <> 0) do begin
    {вычислить положение средней точки; оно будет не менее 1}
    MidPoint := (ListCount + 1) div 2;
    {переместиться вперед до средней точки}
    WorkCursor := BLCursor;
    WorkCursorIx := BLCursorIx;
    for i := 1 to MidPoint do begin
      WorkParent := WorkCursor;
      WorkCursor := WorkCursor^.slnNext;
      inc(WorkCursorIx);
    end;
```

```
{сравнить значение узла с искомым значением}
CompareResult := aCompare(WorkCursor^.slnData, aItem);
{если значение узла меньше искомого, уменьшить размер списка и повторить
цикл}
if (CompareResult < 0) then begin
    dec(ListCount, MidPoint);
    BLCursor := WorkCursor;
    BLCursorIx := WorkCursorIx;
end
{если значение узла больше искомого, уменьшить размер списка и повторить
цикл}
else if (CompareResult > 0) then begin
    ListCount := MidPoint - 1;
end
{в противном случае искомое значение найдено; установить реальный курсор на
найденный узел}
else begin
    FCursor := WorkCursor;
    FParent := WorkParent;
    FCursorIx := WorkCursorIx;
    Result := true;
    Exit;
end;
end;
Result := false;
end;
```

Функция бинарного поиска для класса **TtdDoubleLinkedList** аналогична приведенной функции.

Вставка элемента в отсортированный контейнер

Если необходимо создать отсортированный массив или связный список, у нас существует выбор того или иного метода поддержания порядка элементов. Можно сначала вставлять элементы в контейнер, а затем их сортировать и сортировать содержимое контейнера при вставке каждого нового элемента, или же при выполнении вставки находить позицию, вставив новый элемент в которую контейнер останется отсортированным. Если предполагается, что контейнер будет часто использоваться в отсортированном виде, тогда имеет смысл при вставке сохранять правильный порядок элементов.

В таком случае наша задача сводится к вычислению положения нового элемента в отсортированном списке. После определения позиции мы просто вставляем в нее новый элемент. Ранее говорилось, что последовательный поиск может помочь определить точку вставки, но, к сожалению, быстродействие последовательного поиска достаточно низкое. Можно ли для определения точки вставки воспользоваться бинарным поиском?

Оказывается, можно. Посмотрите внимательно на реализацию бинарного поиска для массива, приведенную в листинге 4.9. Когда выполнение цикла завершается, и искомый элемент не найден, *что* можно определить на основании значений переменных **L**, **R** и **M**? Во-первых, очевидно, что $L > R$. Рассмотрим, что происходит при выполнении цикла в последний раз. В начале цикла мы должны были иметь $L = R$ или $L = R - 1$. При этом вычисление даст, что $M = L$. Если бы разница между **L** и **R** была больше, скажем, $L = R - 2$, тогда значение **M** попало бы в диапазон между **L** и **R**, и цикл был бы выполнен, по крайней мере, еще один раз.

Если при выполнении цикла в последний раз искомый элемент был меньше, чем элемент в позиции **M**, то переменная **R** получила бы значение $M - 1$, и цикл завершился бы. Мы уже знаем, что искомого значения не было до элемента **M**, поэтому можно сделать вывод, что новый элемент должен быть вставлен между элементами $M - 1$ и **M**. Другими словами, мы вставляем элемент в позицию **M**.

С другой стороны, если бы искомый элемент был больше элемента в позиции **M**, то переменная **L** получила бы значение $M + 1$. В этом случае можно принять, что в начале цикла $L = R$. В противном случае цикл был бы выполнен еще один раз. Мы уже знаем, что искомого значения не было после элемента **M**, поэтому можно сделать вывод, что новый элемент должен быть вставлен между элементами **M** и $M + 1$. Другими словами, мы вставляем элемент в позицию $M + 1$.

Таким образом, новый элемент должен вставляться в позицию **M** или $M + 1$ в зависимости от того, что произошло при последнем выполнении цикла. Но давайте подумаем еще раз. Разве между описанными двумя случаями нет ничего общего? Оказывается, что на место вставки в обоих случаях указывает значение переменной **L**. Таким образом, вставка выполняется в позицию **L**.

В приведенном ниже листинге показано, каким образом можно вставить новый элемент в массив **TList**. В коде предполагается, что если вновь вставляемый элемент уже присутствует в массиве, вставка будет игнорироваться (другими словами, повторение элементов не допускается). Функция возвращает индекс вставленного элемента. Легко проверить, что приведенная функция будет работать даже в случае, когда список перед вставкой пуст.

Листинг 4.11. Вставка элемента в отсортированный массив **TList** с помощью алгоритма бинарного поиска

```
function TDTListSortedInsert(aList : TList; aItem : pointer;
                             aCompare : TtdCompareFunc) : integer;
var
  L, R, M : integer;
  CompareResult : integer;
begin
  { задать значения левого и правого индексов }
  L := 0;
  R := pred(aList.Count);
```

```
while (L <= R) do begin
  {вычислить индекс среднего элемента}
  M := (L + R) div 2;
  {сравнить значение среднего элемента с заданным значением}
  CompareResult := aCompare(aList.List^[M], aItem);
  {если значение среднего элемента меньше заданного значения, переместить
   левый индекс на позицию после среднего элемента}
  if (CompareResult < 0) then
    L := succ(M)
  {если значение среднего элемента больше заданного значения, переместить
   правый индекс на позицию перед средним элементом}
  else if (CompareResult > 0) then
    R := pred(M)
  {в противном случае элемент найден, выйти из функции}
  else begin
    Result := M;
    Exit;
  end;
end;
Result := L;
aList.Insert(L, aItem);
end;
```

Для связанного списка функция будет еще проще, поскольку нам не нужно решать, каким образом вычислять индекс для вставки нового элемента. Поиск сам указывает на точку вставки элемента.

Резюме

Эта глава была посвящена поиску. Было показано, каким образом выполняется последовательный поиск и как можно улучшить алгоритм поиска для отсортированных массивов и связанных списков. Было доказано, что для отсортированных контейнеров гораздо быстрее будет алгоритм бинарного поиска. И, наконец, мы рассмотрели использование алгоритма бинарного поиска для вставки нового элемента в требуемое место отсортированного массива.



Глава 5

Сортировка

Сортировка при повседневном программировании встречается очень часто. Когда на форме выводится поле со списком, его удобнее и легче использовать, если элементы в списке отсортированы в алфавитном порядке. Мы, как люди, при изучении данных предпочитаем просматривать их в определенном порядке, который помогает визуально отображать распределение данных. Представьте себе, как сложно было бы пользоваться телефонной книгой, если бы она была отсортирована не по фамилиям в алфавитном порядке, а в каком-нибудь другом порядке. Главы в этой книге, как и в любой другой, расположены в соответствие с их номерами. Что касается разработки, то с программами удобнее работать, если данные отсортированы. Например, алгоритм двоичного поиска быстрее алгоритма последовательного поиска при большом количестве элементов в контейнере, поэтому чтобы выиграть в скорости поиска, имеет смысл поддерживать отсортированный порядок элементов.

Существуют десятки различных алгоритмов сортировки. Каждый со своими характеристиками, своими достоинствами и недостатками. При этом каждый алгоритм оптимизирован для использования для определенных наборов данных.

Алгоритмы сортировки

Алгоритмы сортировки являются одним из наиболее изученных направлений теории вычислительных систем. В общем случае определить характеристики выполнения сортировки достаточно просто. Можно доказать, что любой алгоритм сортировки, основанный на сравнении, принадлежит к классу $O(n \log(n))$. Ниже мы рассмотрим несколько таких алгоритмов.

Кроме того, изучение и реализация алгоритмов сортировки изобилует большим количеством разного рода хитростей. Мы рассмотрим алгоритмы, не требующие дополнительной памяти, требующие большого объема дополнительной памяти, сохраняющие двоичное дерево в массиве, рекурсивные алгоритмы, а также алгоритмы, которые объединяют элементы нескольких списков.

Коды для основных алгоритмов сортировки, описанных в этой главе, можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDSorts.pas**.

Перед тем, как приступить к подробному рассмотрению алгоритмов, давайте введем несколько фундаментальных правил. Все типы сортировок, которые будут описаны ниже, используют сравнение. Чтобы алгоритм "знал", как располагать элементы в списке, их необходимо сравнивать между собой. Мы не будем приводить примеры сортировки для целых чисел, строк или переменных типа **TMyRecord**. Давайте рассматривать проблему сортировки более широко. Примем, что необходимо выполнить сортировку элементов, которые заданы указателями. Это тот же принцип, который мы использовали при изучении структур данных: данные задаются их указателями. Отделяя данные от манипулирования ими, мы уделяем больше внимания характеристикам алгоритмов или структур данных, а не занимаемся ненужной разработкой или оптимизацией методов для целых чисел, строк или других типов данных.

В этой главе элементы будут сравниваться с помощью функции сравнения, описываемой стандартным прототипом **TtdCompareFunc**. Поэтому функции сортировки в качестве одного из входных параметров должны содержать функцию сравнения.

Поскольку фактически будет выполняться сравнение указателей, имеет смысл использовать структуры данных, которые хранят элементы в форме указателей. Для начала рассмотрим алгоритм сортировки на примере стандартного массива **TList**. Написанные нами функции сортировки могут быть обобщенными: они будут перегруппировывать и сравнивать указатели с использованием функции сравнения, заданной пользователем. Затем обобщенные функции сортировки можно будет преобразовать с целью применения с другими типами массивов. После описания стандартных алгоритмов сортировки мы рассмотрим сортировку в связанных списках. Таким образом, большинство написанных функций сортировки в качестве входного параметра смогут принимать экземпляр **TList**.

И, наконец, для создания действительно обобщенных функций, мы введем возможность сортировки не только всего массива **TList**, но и некоторого его диапазона. Для описания диапазона будут использоваться два параметра: индекс первого элемента диапазона и индекс последнего элемента диапазона. Это подразумевает, что функция сортировки должна выполнять проверку попадания индекса первого и последнего элемента сортируемого диапазона в диапазон допустимых индексов массива **TList** (т.е. оба индекса больше или равны нулю и меньше, чем **Count**, и первый индекс меньше второго).

Листинг 5.1. Проверка попадания индекса в допустимый диапазон индексов для массива TList

[illegible]

Выполнение проверки до сортировки массива дает нам дополнительное преимущество. Стандартный метод доступа к элементам массива **TList** – это свойство **Items**. Поскольку это свойство по умолчанию, его можно опускать, т.е. вместо **MyList.Items[i]** записывать **MyList[i]**. Несмотря на кажущуюся простоту, здесь скрыта большая проблема, по крайней мере, если говорить об эффективности сортировки. Дело в том, что, например, оператор **MyList[i]** приводит к тому, что компилятор вместо вызова элемента вставляет метод **MyList.Get(i)** – метод записи свойства **Items**. И первое, что делает метод **Get**, – проверяет, что индекс *i* находится в диапазоне от 0 до **Count-1**. Тем не менее, если мы реализовали алгоритм сортировки правильно, мы можем *гарантировать*, что переданный методу индекс уже находится внутри допустимого диапазона индексов массива. То же относится и к записи значения в **MyList[i]**: вызывается метод **Put**, который также проверяет попадание переданного ему индекса внутрь допустимого диапазона. Таким образом, используя свойство **Items**, мы выполняем ненужный объем работы: вызываем метод, который проверяет уже проверенный индекс. Не очень-то хорошо.

Можно ли каким-то образом устранить двойную проверку индексов? К счастью, это так. Класс **TList** имеет еще одно свойство – **List**. Это свойство, доступное только для чтения, возвращает указатель типа **PPointerList** на внутренний массив указателей, используемый массивом **TList** для хранения элементов. Никакие вспомогательные методы не вызываются и никаких проверок не производится. Конечно, применяя это свойство, мы принимаем на себя ответственность, что при попытках чтения и записи мы не должны выходить за границы массива.

Принимая во внимание все вышесказанное, можно объявить функцию сортировки следующего вида:

Type

```
TtdSortRoutine = procedure(aList : TList;  
                           aFirst, aLast : integer;  
                           aCompare : TtdCompareFunc)
```


Поскольку все типы сортировок, основанных на сравнении, будут иметь приведенный прототип, мы получаем возможность легко выполнять эксперименты с каждым алгоритмом, например, сравнивать времена их выполнения.

Существуют три основных типа последовательностей данных, которые можно использовать для тестирования функций сортировки. Можно сортировать данные, находящиеся в произвольном порядке (перетасованные данные, если хотите), уже отсортированные данные и данные, отсортированные в обратном порядке. Вторая последовательность данных позволит оценить поведение алгоритма на отсортированном списке – некоторые алгоритмы в подобной ситуации выполняются неэффективно.

Список, отсортированный в обратной последовательности, также является критическим для некоторых алгоритмов – многие элементы должны пройти *длинный* путь до попадания в требуемую позицию.

Кроме того, есть еще одна последовательность данных, которую можно использовать при тестировании алгоритмов сортировки, – набор данных, содержащий большое количество повторений ограниченного количества элементов. Другими словами, в таком наборе находятся элементы, для многих из которых функция сравнения будет возвращать значение 0. Это немного странно, но есть, по крайней мере, один алгоритм сортировки, который традиционно плохо работает с наборами данных с повторениями элементов.

Ситуация, в которой мы сейчас находимся, напоминает замкнутый круг (для тестирования алгоритмов сортировки и определения их эффективности необходимы наборы отсортированных данных, но как их получить?), однако два набора отсортированных данных можно сформировать очень просто. Первый набор, используемый при тестировании алгоритмов, случайная последовательность, заслуживает отдельного рассмотрения. Как это ни парадоксально, но обсуждение сортировки мы начнем с описания методов перестановки данных с целью получения случайной последовательности. Выражаясь языком физики, сейчас мы научимся увеличивать энтропию, перед тем как показать, как ее уменьшать.

Тасование массива **TList**

Каким образом можно перетасовать элементы массива **TList**? Большинство из вас в качестве первого алгоритма приведут самый простой: посетить каждый элемент массива, от первого до последнего, и переставить его с другим, случайно выбранным элементом. Реализация такого алгоритма в Delphi будет выглядеть следующим образом:

Листинг 5.2. Простое тасование элементов

```
procedure TDSimpleListShuffle(aList : TList;  
                             aStart, aEnd : integer);  
  
var  
    Range : integer;  
    Inx : integer;
```



```

RandomInx : integer;
TempPtr : pointer;
begin
  TDValidateListRange(aList, aStart, aEnd, 'TDSimpleListShuffle');
  Range := succ(aEnd - aStart);
  for Inx := aStart to aEnd do begin
    RandomInx := aStart + Random(Range);
    TempPtr := aList.List^[Inx];
    aList.List^[Inx] := aList.List^[RandomInx];
    aList.List^[RandomInx] := TempPtr;
  end;
end;

```

А теперь давайте попробуем определить, сколько последовательностей можно получить с помощью приведенного алгоритма. После первого выполнения цикла мы получим одну из n возможных комбинаций (первый элемент может быть переставлен с любым другим, включая самого себя). После второго выполнения цикла мы снова получим одну из n возможных комбинаций, которые совместно с n комбинациями после первого выполнения дадут n^2 возможных комбинаций. Очевидно, что после выполнения всего цикла мы получим одну из n^n возможных комбинаций.

С описанным алгоритмом связана только одна проблема. Если рассматривать тасование с другой точки зрения, с позиции главных принципов, можно показать, что для первой позиции можно выбрать один из n элементов. После этого для второй позиции останется выбор только из $n - 1$ элементов. Далее для третьей позиции элементов будет уже $n - 2$ и т.д. В результате таких рассуждений можно прийти к выводу, что общее количество возможных комбинаций будет вычисляться как $n!$ ($n!$ означает n факториал и сводится к произведению $n * (n - 1) * (n - 2) * \dots * 1$.)

Вернемся к проблеме: если не брать во внимание случай, когда $n = 1$, n^n больше, а часто намного больше, чем $n!$. Таким образом, с помощью описанного алгоритма формируются повторяющиеся последовательности, причем некоторые из них будут повторяться чаще, нежели другие, поскольку n^n не делится на $n!$ без остатка.

В качестве более эффективного алгоритма тасования можно предложить метод, с помощью которого мы определили точное количество возможных комбинаций: брать первый элемент со всех n элементов, второй – из оставшихся $(n - 1)$ элементов и т.д. На основе такого алгоритма можно создать следующую реализацию, где для удобства вычисления индекса цикл начинается с конца, а не с начала массива.

Листинг 5.3. Корректный метод тасования массива TList

```

procedure TDListShuffle(aList : TList; aStart, aEnd : integer);
var
  Range : integer;
  Inx : integer;
  RandomInx : integer;
  TempPtr : pointer;

```

begin

TDValidateListRange(aList, aStart, aEnd, 'TDListShuffle');

{для каждого элемента, считая справа...}

for Inx := (aEnd - aStart) **downto** aStart + 1 **do begin**

{сгенерировать случайное число из диапазона от aStart до текущего индекса}

RandomInx := aStart + Random(Inx - aStart + 1);

{если случайный индекс не равен текущему, переставить элементы}

if (RandomInx <> Inx) **then begin**

TempPtr := aList.List^[Inx];

aList.List^[Inx] := aList.List^[RandomInx];

aList.List^[RandomInx] := TempPtr;

end;

end;

end;

Основы сортировки

Алгоритмы сортировки можно разделить на два типа: *устойчивые* и *неустойчивые*. К устойчивой сортировке относятся те алгоритмы, которые при наличии в наборе данных нескольких равных элементов в отсортированном наборе оставляют их в том же порядке, в котором эти элементы были в исходном наборе. Например, предположим, что в наборе имеется три элемента и значение каждого элемента равно 42 (т.е. элементы равны). Пусть в исходном наборе элемент А находится в позиции 12, элемент В – в позиции 234, а С – в позиции 3456. После выполнения устойчивой сортировки они будут находиться в последовательности А, В, С, т.е. их взаимный порядок не изменится. С другой стороны, неустойчивая сортировка не гарантирует, что элементы с равными значениями будут находиться в определенной последовательности. Для нашего примера элементы А, В и С могут оказаться в последовательности А, В, С, или С, В, А, или любой другой.

В большинстве случаев устойчивость сортировки не имеет никакого значения. Устойчивая сортировка бывает нужна только для отдельных алгоритмов, но, как правило, нам нечего беспокоиться об устойчивости.

Каждый из алгоритмов сортировки с целью упрощения понимания будет описан на примере сортировки колоды карт. Выберите все черви из колоды и перетасуйте их (манипулирование только 13 картами позволит упростить вашу работу).

Самые медленные алгоритмы сортировки

Мы будем рассматривать все алгоритмы сортировки, разделяя их на три группы. К первой группе отнесем медленные алгоритмы, принадлежащие к классу $O(n^2)$, хотя парочка из них в отдельных ситуациях на определенных распределениях данных дает очень высокие показатели производительности.

Пузырьковая сортировка

Первый алгоритм, с которым сталкиваются все программисты при изучении азов программирования, – это *пузырьковая сортировка (bubble sort)*. Как это ни прискорбно, но из всех известных алгоритмов пузырьковая сортировка является самой медленной. Хотя, возможно, ее легче запрограммировать, чем другие алгоритмы сортировки (хотя и не намного).

Пузырьковая сортировка работает следующим образом. Разложите ваши карты (помните, что их всего 13?). Посмотрите на двенадцатую и тринадцатую карту. Если двенадцатая карта старше тринадцатой, поменяйте их местами. Теперь перейдите к одиннадцатой и двенадцатой картам. Если одиннадцатая карта старше двенадцатой, поменяйте их местами. То же сделайте и для пар (10, 11), (9, 10) и т.д., пока не дойдете до первой и второй карты. После первого прохода по всей колоде туз окажется на первой позиции. Фактически когда вы "зацепились" за туз он "выплыл" на первую позицию. Теперь вернитесь к двенадцатой и тринадцатой картам. Выполните описанный выше процесс, на этот раз остановившись на второй и третьей картах. Обратите внимание, что вам удалось переместить двойку на вторую позицию. Продолжайте процесс сортировки, уменьшая с каждым новым циклом количество просматриваемых карт и поступая так до тех пор, пока вся колода не будет отсортирована.

Полагаем, вы согласитесь с тем, что сортировка была довольно-таки утомительной. При реализации алгоритма на языке Pascal "утомительность" выражается медленной скоростью работы. Тем не менее, существует один простой метод оптимизации пузырьковой сортировки: если при выполнении очередного прохода не было выполнено ни одной перестановки, значит, карты уже отсортированы в требуемом порядке.

Листинг 5.4. Пузырьковая сортировка

```

procedure TDBubbleSort(aList : TList; aFirst : integer;
                        aLast : integer; aCompare : TtdCompareFunc);
var
    i, j : integer;
    Temp : pointer;
    Done : boolean;
begin
    TDValidateListRange(aList, aFirst, aLast, 'TDBubbleSort');

```

5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
 5 Король 4 2 Валет Туз 9 8 3 10 Дама **6** 7
 5 Король 4 2 Валет Туз 9 8 3 10 **6** Дама 7
 5 Король 4 2 Валет Туз 9 8 3 **6** 10 Дама 7
 5 Король 4 2 Валет Туз 9 8 **3** 6 10 Дама 7
 5 Король 4 2 Валет Туз 9 **3** 8 6 10 Дама 7
 5 Король 4 2 Валет Туз **3** 9 8 6 10 Дама 7
 5 Король 4 2 Валет **Туз** 3 9 8 6 10 Дама 7
 5 Король 4 2 **Туз** Валет 3 9 8 6 10 Дама 7
 5 Король 4 **Туз** 2 Валет 3 9 8 6 10 Дама 7
 5 Король **Туз** 4 2 Валет 3 9 8 6 10 Дама 7
 5 **Туз** Король 4 2 Валет 3 9 8 6 10 Дама 7
Туз 5 Король 4 2 Валет 3 9 8 6 10 Дама 7

Рисунок 5.1. Один проход с помощью алгоритма пузырьковой сортировки

```

for i := aFirst to pred(aLast) do begin
  Done := true;
  for j := aLast downto succ(i) do
    if (aCompare(aList.List^[j], aList.List^[j-1]) < 0) then begin
      {переставить j-ый и (j-1)-ый элементы}
      Temp := aList.List^[j];
      aList.List^[j] := aList.List^[j-1];
      aList.List^[j-1] := Temp;
      Done := false;
    end;
  if Done then
    Exit;
end;
end;

```

Пузырьковая сортировка принадлежит к алгоритмам класса $O(n^2)$. Как видите, в реализации присутствуют два цикла: внешний и внутренний, при этом количество выполнений каждого цикла зависит от количества элементов в массиве. При первом выполнении внутреннего алгоритма будет произведено $n - 1$ сравнений, при втором — $n - 2$, при третьем — $n - 3$ и т.д. Всего будет $n - 1$ таких циклов, таким образом, общее количество сравнений составит:

$$(n - 1) + (n - 2) + \dots + 1$$

Приведенную сумму можно упростить до $n(n - 1)/2$ или $(n^2 - n)/2$. Другими словами, получаем $O(n^2)$. Количество перестановок вычислить несколько сложнее, но в худшем случае (когда элементы в исходном наборе были отсортированы в обратном порядке) количество перестановок будет равно количеству сравнений, т.е. снова получаем $O(n^2)$.

Небольшая оптимизация метода пузырьковой сортировки, о которой мы говорили чуть выше, означает, что если элементы в наборе уже отсортированы в нужном порядке, пузырьковая сортировка будет выполняться очень быстро: будет выполнен всего один проход по списку, не будет сделано ни одной перестановки и выполнение алгоритма завершится. $(n - 1)$ сравнений и ни одной перестановки говорят о том, что в лучшем случае быстродействие пузырьковой сортировки равно $O(n)$.

Одна большая проблема, связанная с пузырьковой сортировкой, да и честно говоря, со многими другими алгоритмами, состоит в том, что переставляются только соседние элементы. Если элемент с наименьшим значением оказывается в самом конце списка, он будет меняться местами с соседними элементами до тех пор, пока он не достигнет первой позиции.

Пузырьковая сортировка относится к нестабильным алгоритмам, поскольку из двух элементов с равными значениями первым в отсортированном списке будет тот, который находился в исходном списке дальше от начала. Если изменить тип сравнения на "меньше чем" или "равен", а не просто "меньше", тогда пузырьковая

сортировка станет устойчивой, но количество перестановок увеличится, и введенная нами оптимизация не даст запланированного выигрыша в скорости.

Шейкер-сортировка

Пузырьковая сортировка имеет одну малоизвестную вариацию, которая на практике дает незначительное увеличение скорости, — это так называемая *шейкер-сортировка* (*shaker sort*).

Вернемся к картам. Выполните первый проход согласно алгоритму сортировки. Туз попадет на первую позицию. Теперь, вместо прохода колоды карт справа налево, пройдите слева направо: сравните вторую и третью карты и старшую карту поместите на третью позицию. Сравните третью и четвертую карты, и при необходимости поменяйте их местами. Продолжайте сравнения вплоть до достижения пары (12, 13). По пути к правому краю колоды вы "захватили" короля и переместили его на последнюю позицию.

А теперь снова пройдите колоду справа налево до второй карты. Во вторую позицию попадет двойка. Продолжайте чередовать направления проходов до тех пор, пока не будет отсортирована вся колода.

Листинг 5.5. Шейкер-сортировка

```
procedure TDSHakerSort(aList : TList; aFirst : integer;
                      aLast : integer; aCompare : TtdCompareFunc);
var
  i : integer;
  Temp : pointer;
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDSHakerSort');
  while (aFirst < aLast) do begin
    for i := aLast downto succ(aFirst) do
      if (aCompare(aList.List^[i], aList.List^[i-1]) < 0) then begin
        Temp := aList.List^[i];
        aList.List^[i] := aList.List^[i-1];
        aList.List^[i-1] := Temp;
      end;
    inc(aFirst);
```

5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
 5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
 5 Король 4 2 Валет Туз 9 8 3 10 6 Дама 7
 5 Король 4 2 Валет Туз 9 8 3 6 10 Дама 7
 5 Король 4 2 Валет Туз 9 8 3 6 10 Дама 7
 5 Король 4 2 Валет Туз 9 3 8 6 10 Дама 7
 5 Король 4 2 Валет Туз 3 9 8 6 10 Дама 7
 5 Король 4 2 Валет Туз 3 9 8 6 10 Дама 7
 5 Король 4 2 Туз Валет 3 9 8 6 10 Дама 7
 5 Король 4 Туз 2 Валет 3 9 8 6 10 Дама 7
 5 Король Туз 4 2 Валет 3 9 8 6 10 Дама 7
 5 Туз Король 4 2 Валет 3 9 8 6 10 Дама 7
 Туз 5 Король 4 2 Валет 3 9 8 6 10 Дама 7
 Туз 5 Король 4 2 Валет 3 9 8 6 10 Дама 7
 Туз 5 4 Король 2 Валет 3 9 8 6 10 Дама 7
 Туз 5 4 2 Король Валет 3 9 8 6 10 Дама 7
 Туз 5 4 2 Валет Король 3 9 8 6 10 Дама 7
 Туз 5 4 2 Валет 3 Король 9 8 6 10 Дама 7
 Туз 5 4 2 Валет 3 9 Король 8 6 10 Дама 7
 Туз 5 4 2 Валет 3 9 8 Король 6 10 Дама 7
 Туз 5 4 2 Валет 3 9 8 6 Король 10 Дама 7
 Туз 5 4 2 Валет 3 9 8 6 10 Король Дама 7
 Туз 5 4 2 Валет 3 9 8 6 10 Дама Король 7
 Туз 5 4 2 Валет 3 9 8 6 10 Дама 7 Король

Рисунок 5.2. Два прохода с помощью шейкер-сортировки

```

for i := succ(aFirst) to aLast do
  if (aCompare(aList.List^[i], aList.List^[i-1]) < 0) then begin
    Temp := aList.List^[i];
    aList.List^[i] := aList.List^[i-1];
    aList.List^[i-1] := Temp;
  end;
dec(aLast);
end;
end;

```

Несмотря на то что шейкер-сортировка принадлежит к алгоритмам класса $O(n^2)$, время ее выполнения немного меньше, чем для пузырьковой сортировки. Причина, по которой алгоритм назван именно шейкер-сортировкой, состоит в том, что элементы в списке колеблются относительно своих позиций до тех пор, пока список не будет отсортирован.

Как и пузырьковая сортировка, шейкер-сортировка относится к неустойчивым алгоритмам.

Сортировка методом выбора

Следующим алгоритмом, который мы рассмотрим, будет *сортировка методом выбора* (*selection sort*). Это пока что первый метод, который действительно можно использовать в повседневной практике (о пузырьковой сортировке и шейкер-сортировке можно уже забыть).

Начиная с правого края колоды, просмотрите все карты и найдите самую младшую (конечно, это будет туз). Поменяйте местами туз с первой картой. Теперь, игнорируя первую карту, снова просмотрите всю колоду справа налево в поисках самой младшей карты. Поменяйте местами младшую карту со второй картой. Далее, игнорируя первые две карты, просмотрите всю колоду справа налево в поисках самой младшей карты и поменяйте найденную карту с третьей картой. Продолжайте процесс до тех пор, пока вся колода не будет отсортирована. Очевидно, что тринадцатый цикл не понадобится, поскольку он будет манипулировать только с одной картой, которая к тому времени уже будет находиться в требуемой позиции.

5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
Туз Король 4 2 Валет 5 9 8 3 10 Дама 6 7
 Туз 2 4 **Король** Валет 5 9 8 3 10 Дама 6 7
 Туз 2 3 Король Валет 5 9 8 4 10 Дама 6 7
 Туз 2 3 4 Валет 5 9 8 **Король** 10 Дама 6 7
 Туз 2 3 4 5 **Валет** 9 8 Король 10 Дама 6 7
 Туз 2 3 4 5 6 9 8 Король 10 Дама **Валет** 7
 Туз 2 3 4 5 6 7 8 Король 10 Дама Валет 9
 Туз 2 3 4 5 6 7 8 9 10 Дама Валет **Король**
 Туз 2 3 4 5 6 7 8 9 10 **Валет Дама** Король

Рисунок 5.3. Сортировка методом выбора

Листинг 5.6. Сортировка методом выбора

```

procedure TDSelectionSort(aList : TList; aFirst : integer;
                           aLast : integer; aCompare : TtdCompareFunc);
var
  i, j : integer;
  IndexOfMin : integer;
  Temp : pointer;

```



```
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDSelectionSort');
  for i := aFirst to pred(aLast) do begin
    IndexOfMin := i;
    for j := succ(i) to aLast do
      if (aCompare(aList.List^[j], aList.List^[IndexOfMin]) < 0) then
        IndexOfMin := j;
    if (aIndexOfMin <> i) then begin
      Temp := aList.List^[i];
      aList.List^[i] := aList.List^[IndexOfMin];
      aList.List^[IndexOfMin] := Temp;
    end;
  end;
end;
```

Как видите, в приведенном коде снова присутствуют два вложенных цикла, следовательно, сортировка методом выбора относится к алгоритмам класса $O(n^2)$. В первом цикле индекс проходит значения от **aFirst** до **aLast-1** и при каждом его выполнении во внутреннем цикле определяется элемент с минимальным значением в оставшейся части списка. В отличие от нашего примера с картами, внутренний цикл заранее не знает, каковым будет минимальный элемент в списке, поэтому ему нужно просмотреть все элементы. После обнаружения минимального элемента он переставляется в требуемую позицию.

Сортировка методом выбора интересна одной своей особенностью. Количество выполняемых сравнений для первого прохода равно n , для второго – $n-1$ и т.д. Общее количество сравнений будет равно $n(n+1)/2 - 1$, т.е. сортировка принадлежит к классу алгоритмов $O(n^2)$. Тем не менее, количество перестановок намного меньше: при каждом выполнении внешнего цикла производится всего одна перестановка. Таким образом, общее количество перестановок $(n-1)$, т.е. $O(n)$. Что это означает на практике? Если стоимость перестановки элементов намного больше, чем время сравнения (под *стоимостью* в данном случае понимается время или требуемые ресурсы), сортировка методом выбора оказывается достаточно эффективной.

Сортировка методом выбора относится к группе устойчивых алгоритмов. Поиск наименьшего значения будет возвращать первое в списке наименьшее значение из нескольких имеющихся. Таким образом, равные значения будут находиться в отсортированном списке в том же порядке, в котором они были в исходном списке.

Сортировка методом вставок

И последний алгоритм из первого рассматриваемого нами набора – *сортировка методом вставок*, или *сортировка простыми вставками* (*insertion sort*). Этот алгоритм покажется знакомым всем, кто играет в такие карточные игры, как вист или бридж, поскольку большинство игроков сортирует свои карты именно так.

Начинаем с левого края колоды. Сравниваем две первые карты и располагаем их в правильном порядке. Смотрим на третью карту. Вставляем ее в требуемое место по отношению к первым двум картам. Смотрим на четвертую карту и вставляем ее в требуемое место по отношению к первым трем картам. Те же операции выполняем с пятой, шестой, седьмой и всеми последующими картами. При перемещении слева направо левая часть колоды будет отсортированной.

5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
4 5 Король 2 Валет Туз 9 8 3 10 Дама 6 7
2 4 5 Король Валет Туз 9 8 3 10 Дама 6 7
 2 4 5 **Валет Король** Туз 9 8 3 10 Дама 6 7
Туз 2 4 5 Валет Король 9 8 3 10 Дама 6 7
 Туз 2 4 5 **9 Валет Король** 8 3 10 Дама 6 7
 Туз 2 4 5 **8 9 Валет Король** 3 10 Дама 6 7
 Туз 2 **3 4 5 8 9 Валет Король** 10 Дама 6 7
 Туз 2 3 4 5 8 9 **10 Валет Король** Дама 6 7
 Туз 2 3 4 5 8 9 10 Валет **Дама Король** 6 7
 Туз 2 3 4 5 **6 8 9 10 Валет Дама Король** 7
 Туз 2 3 4 5 6 7 8 9 10 Валет **Дама Король**

Рисунок 5.4. Стандартная сортировка методом вставок

Листинг 5.7. Стандартная сортировка методом вставок

```
procedure TDInsertionSortStd(aList : TList; aFirst : integer;
                             aLast : integer; aCompare : TtdCompareFunc);
var
  i, j : integer;
  Temp : pointer;
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDInsertionSortStd');
  for i := succ(aFirst) to aLast do begin
    Temp := aList.List^[i];
    j := i;
    while (j > aFirst) and (aCompare(Temp, aList.List^[j-1]) < 0) do
      begin
        aList.List^[j] := aList.List^[j-1];
        dec(j);
      end;
      aList.List^[j] := Temp;
    end;
  end;
```

В приведенной реализации сортировки методом вставок имеется одна очень интересная особенность: значение текущего элемента сохраняется в локальной переменной, а затем при поиске нужного места его вставки (внутренний цикл) мы перемещаем каждый элемент, значение которого больше текущего, на одну позицию вправо, тем самым, перемещая "дыру" в списке влево. В конце концов, мы находим нужное место и помещаем сохраненное значение в освободившееся место.

Давайте посмотрим на внутренний цикл. Его выполнение завершается при соблюдении одного из двух условий: достигнуто начало списка, т.е. текущее значение меньше значений всех уже отсортированных элементов, или обнаружено значение, меньшее текущего. Тем не менее, обратите внимание, что первое условие

проверяется при каждом выполнении внутреннего цикла, несмотря на то что оно соблюдается достаточно редко, когда текущее значение меньше, чем значения всех уже отсортированных элементов, однако оно предотвращает выход за пределы списка. Традиционным методом исключения этой дополнительной проверки является введение в начало списка сигнального элемента, который меньше любого другого элемента в списке. К сожалению, в общем случае минимальный элемент в списке заранее неизвестен и, кроме того, в списке нет места для вставки дополнительного элемента. (Теоретически потребуется скопировать весь список в другой, размер которого больше на один элемент, установить значение первого элемента в этом новом списке равным минимальному значению из сортируемого списка, а затем после сортировки скопировать элементы в исходный список. И все это ради того, чтобы исключить одну проверку. Нет уж, спасибо.)

5 King 4 2 Jack Ace 9 8 3 10 Queen 6 7	
Ace King 4 2 Jack 5 9 8 3 10 Queen 6 7	→ Первый проход сортировки методом выбора
Ace King 4 2 Jack 5 9 8 3 10 Queen 6 7	→ Стандартный метод вставок
Ace 4 King 2 Jack 5 9 8 3 10 Queen 6 7	→
Ace 2 4 King Jack 5 9 8 3 10 Queen 6 7	→
Ace 2 4 Jack King 5 9 8 3 10 Queen 6 7	→
Ace 2 4 5 Jack King 9 8 3 10 Queen 6 7	→
Ace 2 4 5 9 Jack King 8 3 10 Queen 6 7	→
Ace 2 4 5 8 9 Jack King 3 10 Queen 6 7	→
Ace 2 3 4 5 8 9 Jack King 10 Queen 6 7	→
Ace 2 3 4 5 8 9 10 Jack King Queen 6 7	→
Ace 2 3 4 5 8 9 10 Jack Queen King 6 7	→
Ace 2 3 4 5 6 8 9 10 Jack Queen King 7	→
Ace 2 3 4 5 6 7 8 9 10 Jack Queen King	→

Рисунок 5.5. Сортировка методом вставок

Существует более эффективный метод оптимизации: просмотреть весь список, найти элемент с наименьшим значением и переставить его на первое место (фактически это выполнение первого цикла сортировки методом 'выбора'). Теперь, когда первый элемент находится в требуемой позиции, можно выполнять стандартную процедуру метода вставок и игнорировать возможность выхода за начало списка.

Листинг 5.8. Оптимизированная сортировка методом вставок

```

procedure TDInsertionSort(aList : TList; aFirst : integer;
                           aLast : integer; aCompare : TtdCompareFunc);
var
  i, j : integer;
  IndexOfMin : integer;
  Temp : pointer;
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDInsertionSort');
  { найти наименьший элемент и поместить его в первую позицию }
  IndexOfMin := aFirst;

```

```

for i := succ(aFirst) to aLast do
  if (aCompare(aList.List^[i], aList.List^[IndexOfMin]) < 0) then
    IndexOfMin := i;
if (aFirst <> IndexOfMin) then begin
  Temp := aList.List^[aFirst];
  aList.List^[aFirst] := aList.List^[IndexOfMin];
  aList.List^[IndexOfMin] := Temp;
end;
{отсортировать с использованием метода простых вставок}
for i := aFirst+2 to aLast do begin
  Temp := aList.List^[i];
  j := i;
  while (aCompare(Temp, aList.List^[j-1]) < 0) do begin
    aList.List^[j] := aList.List^[j-1];
    dec(j);
  end;
  aList.List^[j] := Temp;
end;
end;

```

Хотите верьте, хотите нет, но предварительная установка наименьшего значения в первую позицию и исключение дополнительной проверки выхода за границы списка при тестировании дала увеличение быстродействия примерно на 7%.

Как и три предыдущие рассмотренные нами алгоритма, сортировка методом вставок принадлежит к классу алгоритмов $O(n^2)$. Как и в случае с пузырьковой сортировкой, если исходный список уже отсортирован, сортировка методом вставок практически не выполняет никаких действий помимо сравнения пар двух соседних элементов. Худшим случаем для сортировки методом вставок является ситуация, когда исходный список отсортирован в обратном порядке (как и для пузырьковой сортировки) – для попадания в требуемое место каждому элементу нужно пройти максимальное расстояние.

Тем не менее, если список частично отсортирован, и каждый элемент находится недалеко от требуемого места, сортировка методом вставок будет выполняться очень быстро. Фактически она превращается в алгоритм класса $O(n)$. (Другими словами, внешний цикл выполняется $n - 1$ раз, а внутренний – всего несколько раз, что соответствует небольшому расстоянию элементов от их позиции в отсортированном списке.) Таким образом, во внутреннем цикле выполняется некоторое постоянное количество проходов (т.е. сравнений и перемещений), скажем, d . Для внешнего цикла, как мы уже говорили, количество проходов равно $n - 1$. Следовательно, общее количество сравнений и перемещений будет выражаться значением $d(n - 1)$ (алгоритм класса $O(n)$). Несмотря на то что на практике частично отсортированные списки встречаются достаточно редко, тем не менее, возможна ситуация, когда с частично отсортированными списками можно сталкиваться гораздо чаще. Мы рассмотрим эту ситуацию немного ниже.

Сортировка методом вставок (любая ее вариация) принадлежит к группе устойчивых алгоритмов. Она сохраняет относительное положение элементов с равными значениями, поскольку поиск требуемой позиции для элемента завершается, когда найден элемент, значение которого меньше или *равно* значению текущего элемента. Следовательно, относительное положение элементов с равными значениями сохраняется.

Как и при пузырьковой сортировке, при сортировке методом вставок элементы попадают в нужные позиции только за счет смены позиций с соседними элементами. Если элемент находится далеко от требуемой позиции, его перемещение занимает много времени. Если бы только мы могли перемещать элементы не через соседние элементы, а сразу в некоторый диапазон, где текущий элемент должен находиться! Давайте познакомимся со вторым набором алгоритмов.

Быстрые алгоритмы сортировки

Алгоритмы второго набора работают быстрее всех тех методов, которые мы только что рассмотрели. Тем не менее, в отличие от набора самых быстрых сортировок, к которому мы вскоре перейдем, очень сложно выполнить их математический анализ. Несмотря на то что на практике алгоритмы этой группы выполняются достаточно быстро, используют их сравнительно редко.

Сортировка методом Шелла

Этот метод разработал Дональд Л. Шелл (Donald L. Shell) в 1959 году. Он основан на сортировке методом вставок и при первом рассмотрении может показаться несколько странным.

Сортировка методом Шелла (Shell sort) пытается повысить скорость работы за счет более быстрого перемещения элементов, находящихся далеко от нужных им позиций. Она предполагает перемещение таких элементов большими "прыжками" через несколько элементов одновременно, уменьшая размер "прыжков" и, в конце концов, окончательная установка элементов в нужные позиции выполняется с помощью классической сортировки методом вставок.

Выполнение сортировки методом Шелла на примере карточной колоды требует немало усилий, но не будем терять времени. Разложите колоду в длинную линию. Извлеките из колоды первую и каждую четвертую карту после первой (т.е., пятую, девятую и тринадцатую). Выполните сортировку выбранных карт с помощью метода вставок и снова поместите все карты в колоду. Извлеките из колоды вторую и каждую четвертую карту после второй (т.е., шестую и десятую). Выполните сортировку выбранных карт с помощью метода вставок и снова поместите все карты в колоду. Выполните те же операции над третьей и каждой четвертой картой после третьей, а затем над четвертой и каждой четвертой картой после четвертой.

После первого прохода карты будут находиться в отсортированном порядке по 4. Какую бы карту вы не выбрали, карты, которые находятся на количество пози-

ций, кратном 4 вперед и назад, будут отсортированы в требуемом порядке. Обратите внимание, что карты в целом не отсортированы, но, тем не менее, независимо от исходного положения карт, после первого прохода они будут находиться недалеко от своих мест в отсортированной последовательности.

Теперь выполним стандартную сортировку методом вставок, в результате чего получим отсортированную колоду карт. Как уже говорилось, при небольших расстояниях между элементами в

исходном списке и их позициями в отсортированном списке (что мы и получили после первого прохода) быстродействие сортировки методом вставок линейно зависит от числа элементов.

Говоря более строгим языком, сортировка методом Шелла работает путем вставки отсортированных подмножеств основного списка. Каждое подмножество формируется за счет выборки каждого h -ого элемента, начиная с любой позиции в списке. В результате будет получено h подмножеств, которые отсортированы методом вставок. Полученная последовательность элементов в списке называется отсортированной по h . Затем значение h уменьшается и снова выполняется сортировка. Уменьшение значения h происходит до тех пор, пока h не будет равно 1, после чего последний проход будет представлять собой стандартную сортировку методом вставок (которая, если быть точным, представляет собой сортировку по 1).

Суть сортировки методом Шелла заключается в том, что сортировка по h быстро переносит элементы в область, где они должны находиться в отсортированном списке, а уменьшение значения h позволяет постепенно уменьшать размер "прыжков" и, в конце концов, поместить элемент в требуемую позицию. Медленному перемещению элементов предшествуют большие "скачки", сводящиеся к простой сортировке методом вставок, которая практически не передвигает элементы.

Какие значения h лучше всего использовать? Шелл в своей первой статье на эту тему предложил значения 1, 2, 4, 8, 16, 32 и т.д. (естественно, в обратном порядке), но с этими значениями связана одна проблема: до последнего прохода элементы с четными индексами никогда не сравниваются с элементами с нечетными индексами. И, следовательно, при выполнении последнего прохода все еще возможны перемещения элементов на большие расстояния (представьте себе, например, искусственный случай, когда элементы с меньшими значениями находятся в позициях с четными индексами, а элементы с большими значениями — в позициях с нечетными индексами).

5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
 3 Король 4 2 5 Туз 9 8 7 10 Дама 6 Валет
 3 Туз 4 2 5 10 9 8 7 Король Дама 6 Валет
 3 Туз 4 2 5 10 9 8 7 Король Дама 6 Валет
 3 Туз 4 2 5 10 9 6 7 Король Дама 8 Валет
 Туз 3 4 2 5 10 9 6 7 Король Дама 8 Валет
 Туз 2 3 4 5 10 9 6 7 Король Дама 8 Валет
 Туз 2 3 4 5 9 10 6 7 Король Дама 8 Валет
 Туз 2 3 4 5 6 9 10 7 Король Дама 8 Валет
 Туз 2 3 4 5 6 7 9 10 Король Дама 8 Валет
 Туз 2 3 4 5 6 7 9 10 Король Дама 8 Валет
 Туз 2 3 4 5 6 7 8 9 10 Дама Король 8 Валет
 Туз 2 3 4 5 6 7 8 9 10 Дама Король Валет
 Туз 2 3 4 5 6 7 8 9 10 Валет Дама Король

Рисунок 5.6. Сортировка методом Шелла

В 1969 году Дональд Кнут (Donald Knuth) предложил последовательность 1, 4, 13, 40, 121 и т.д. (каждое последующее значение на единицу больше, чем утроенное предыдущее значение). Для списков средних размеров эта последовательность позволяет получить достаточно высокие характеристики быстродействия (на основе эмпирических исследований Кнут оценил быстродействие для среднего случая как $O(n^{5/4})$, а для худшего случая было доказано, что скорость работы равна $O(n^{3/2})$) при несложном методе вычисления значений самой последовательности. Ряд других последовательностей позволяют получить более высокие значения скорости работы (хотя и не намного), но требуют предварительного вычисления значений последовательности, поскольку используемые формулы достаточно сложны. В качестве примера можно привести самую быструю известную на сегодняшний день последовательность, разработанную Робертом Седжвиком (Robert Sedgewick): 1, 5, 19, 41, 109 и т.д. (формируется путем слияния двух последовательностей — $9 * 4i - 9 * 2i + 1$ для $i > 0$ и $4i - 3 * 2i + 1$ для $i > 1$). Известно, что для этой последовательности время работы в худшем случае определяется как $O(n^{4/3})$ при $O(n^{7/6})$ для среднего случая. В этой книге мы не будем приводить математические выкладки для определения приведенных зависимостей. Пока не известно, существуют ли еще более быстрые последовательности. (Подробнее выкладки и анализ всех фундаментальных алгоритмов, в числе которых и алгоритмы, рассмотренные в данной книге, а также эффективная их реализация на языках C, C++ и Java, можно найти в многотомниках Роберта Седжвика “Фундаментальные алгоритмы на C++”, “Фундаментальные алгоритмы на C” и “Фундаментальные алгоритмы на Java”, которые выпущены издательством “Диасофт”.)

Листинг 5.9. Сортировка методом Шелла при использовании последовательности Кнута

```
procedure TDSHELLSort(aList : TList; aFirst : integer;
                      aLast : integer; aCompare : TtdCompareFunc);
var
  i, j : integer;
  h : integer;
  Temp : pointer;
  Ninth : integer;
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDSHELLSort');
  {прежде всего вычисляем начальное значение h; оно должно быть близко к одной
   девятой количества элементов в списке}
  h := 1;
  Ninth := (aLast - aFirst) div 9;
  while (h <= Ninth) do
    h := (h * 3) + 1;
  {начать выполнение цикла, который при каждом проходе уменьшает значение h на
   треть}
```

```
while (h > 0) do begin
  {выполнить сортировку методом вставки для каждого подмножества}
  for i := (aFirst + h) to aLast do begin
    Temp := aList.List^[i];
    j := i;
    while (j >= (aFirst+h)) and
      (aCompare(Temp, aList.List^[j-h]) < 0) do begin
      aList.List^[j] := aList.List^[j-h];
      dec(j, h);
    end;
    aList.List^[j] := Temp;
  end;
  {уменьшить значение h на треть}
  h := h div 3;
end;
end;
```

Математические зависимости для анализа быстродействия сортировки методом Шелла достаточно сложны. В общем случае для оценки времени выполнения сортировки при различных значениях h приходится ограничиваться статистическими данными. Тем не менее, анализ быстродействия алгоритма Шелла практически не имеет смысла, поскольку существуют более быстрые алгоритмы.

Что касается устойчивости, то при перестановке элементов, далеко отстоящих друг от друга, возможно нарушение порядка следования элементов с равными значениями. Следовательно, сортировка методом Шелла относится к группе неустойчивых алгоритмов.

Сортировка методом прочесывания

Этот раздел будет посвящен действительно странному алгоритму сортировки – *сортировке методом прочесывания (comb sort)*. Он не относится к стандартным алгоритмам. На сегодняшний день он малоизвестен и поиск информации по нему может не дать никаких результатов. Тем не менее, он отличается достаточно высоким уровнем быстродействия и удобной реализацией. Метод был разработан Стефаном Лейси (Stephan Lacey) и Ричардом Боксом (Richard Box) и опубликован в журнале "Byte" в апреле 1991 года. Фактически он использует пузырьковую сортировку таким же образом, как сортировка методом Шелла использует сортировку методом вставок.

Перетасуйте карты и снова разложите их на столе. Выделите первую и девятую карту. Если они находятся в неправильном порядке, поменяйте их местами. Выделите вторую и десятую карты и, при необходимости, поменяйте их местами. То же самое сделайте для третьей и одиннадцатой карты, четвертой и двенадцатой, а затем пятой и тринадцатой. Далее сравнивайте и переставляйте пары карт (1, 7), (2, 8), (3, 9), (4, 10), (5, 11), (6, 12) и (7, 13) (т.е. карты, отстоящие друг от друга на шесть позиций). А теперь выполните проход по колоде для карт,

отстоящих друг от друга на четыре позиции, затем на три и две позиции. После этого выполните стандартную пузырьковую сортировку (которую можно рассматривать как продолжение предыдущего алгоритма для соседних карт).

Таким образом, вначале карты большими "прыжками" передвигаются в требуемую область. Как и сортировка методом Шелла, прочесывание неудобно выполнять на картах, но в функции для сортировки методом прочесывания требуется всего два цикла – один для уменьшения размера "прыжков", а второй – для выполнения разновидности пузырьковой сортировки.

Как были получены значения расстояний 8, 6, 4, 3, 2, 1? Разработчики этого метода сортировки провели большое количество экспериментов и эмпирическим путем пришли к выводу, что значение каждого последующего расстояния "прыжка" должно быть получено в результате деления предыдущего на 1.3. Этот "коэффициент уменьшения" был лучшим из рассмотренных и позволял сбалансировать зависимость времени выполнения от длины последовательности значений расстояний и времени выполнения пузырьковой сортировки.

Более того, создатели алгоритма пришли к необъяснимому выводу, что значения расстояний между сравниваемыми элементами 9 и 10 являются неоптимальными, т.е. если в последовательности расстояний присутствует значение 9 или 10, его лучше поменять на 11. В этом случае сортировка будет выполняться гораздо быстрее. Проведенные эксперименты подтверждают этот вывод. Теоретических исследований сортировки методом прочесывания на сегодняшний день не производилось, и поэтому нет определенного объяснения, почему приведенная последовательность расстояний является оптимальной.

```

5 Король 4 2 Валет Туз 9 8 3 10 Дама 6 7
3 Король 4 2 Валет Туз 9 8 5 10 Дама 6 7 → расстояние между картами 8
3 10 4 2 Валет Туз 9 8 5 Король Дама 6 7
3 10 4 2 7 Туз 9 8 5 Король Дама 6 Валет
3 8 4 2 7 Туз 9 10 5 Король Дама 6 Валет → расстояние между картами 6
3 Туз 4 2 7 8 9 10 5 Король Дама 6 Валет
3 Туз 4 2 5 8 9 6 7 Король Дама 10 Валет → расстояние между картами 4
2 Туз 4 3 5 8 9 6 7 Король Дама 10 Валет → расстояние между картами 3
2 Туз 4 3 5 7 9 6 8 Король Дама 10 Валет
2 Туз 4 3 5 7 9 6 8 Валет Дама 10 Король
2 Туз 4 3 5 6 9 7 8 Валет Дама 10 Король → расстояние между картами 2
2 Туз 4 3 5 6 8 7 9 Валет Дама 10 Король
2 Туз 4 3 5 6 8 7 9 10 Дама Валет Король
Туз 2 4 3 5 6 8 7 9 10 Дама Валет Король → расстояние между картами 1
Туз 2 3 4 5 6 8 7 9 10 Дама Валет Король
Туз 2 3 4 5 6 7 8 9 10 Дама Валет Король
Туз 2 3 4 5 6 7 8 9 10 Валет Дама Король

```

Рисунок 5.7. Сортировка методом прочесывания (показаны только перестановки)

Листинг 5.10. Сортировка методом прочесывания

```

procedure TDCombSort(aList : TList;
                     aFirst : integer;
                     aLast : integer;
                     aCompare : TtdCompareFunc);

var
    i, j : integer;
    Temp : pointer;
    Done : boolean;
    Gap : integer;
begin
    TDValidateListRange(aList, aFirst, aLast, 'TDCombSort');
    {начать с расстояния, равного количеству элементов}
    Gap := succ(aLast - aFirst);
    repeat
        {предположить, что сортировка будет выполнена на этом проходе}
        Done := true;
        {calculate the new gap}
        Gap := (longint(Gap) * 10) div 13; {Gap := Trunc(Gap / 1.3);}
        if (Gap < 1) then
            Gap := 1
        else if (Gap = 9) or (Gap = 10) then
            Gap := 11;
        {упорядочить два элемента, отстоящих друг от друга на Gap элементов}
        for i := aFirst to (aLast - Gap) do begin
            j := i + Gap;
            if (aCompare(aList.List^[j], aList.List^[i]) < 0) then begin
                {поменять местами элементы с индексами j и (j-Gap)}
                Temp := aList.List^[j];
                aList.List^[j] := aList.List^[i];
                aList.List^[i] := Temp;
                {была выполнена перестановка, следовательно, сортировка не завершена}
                Done := false;
            end;
        end;
    until Done and (Gap = 1);
end;

```

В экспериментах, проведенных автором книги, сортировка методом прочесывания была немного быстрее сортировки методом Шелла (на последовательности Кнута). Кроме того, ее легче запрограммировать (если не говорить о необходимости исключения расстояний 9 и 10). Очевидно, что сортировка методом прочесывания, как и методом Шелла, принадлежит к группе неустойчивых алгоритмов.

Самые быстрые алгоритмы сортировки

И вот, наконец, мы добрались до самых быстрых алгоритмов сортировки. Они очень широко используются на практике и очень важно понимать их особенности, что позволит оптимальным образом реализовывать их в различных приложениях.

Сортировка слиянием

Сортировка слиянием (merge sort) считается весьма интересным алгоритмом. Она привлекательна своей простотой и наличием некоторых важных особенностей (например, она принадлежит к алгоритмам класса $O(n \log(n))$ и не имеет худших случаев), но если приступить к его реализации, можно натолкнуться на большую проблему. Тем не менее, сортировка слиянием очень широко используется при необходимости сортировки содержимого файлов, размер которых слишком велик, чтобы поместиться в памяти.

Мы будем рассматривать сортировку слиянием по шагам, начиная со слияния. Затем мы опишем, как использовать алгоритм для выполнения сортировки. В качестве примера мы не будем пользоваться картами – алгоритм легко понять и без карт.

Представьте себе, что имеется два уже отсортированных списка и необходимо сформировать один список, объединяющий все элементы исходных списков. План А состоит в том, чтобы скопировать оба списка в результирующий и выполнить его сортировку. Но в этом случае, к сожалению, мы не пользуемся тем, что исходные списки уже отсортированы. План Б предусматривает слияние. Смотрим на первые элементы в обоих списках. Элемент с меньшим значением переносим в результирующий список. Снова смотрим на первые элементы в обоих списках и снова переносим в результирующий список элемент с меньшим значением, удаляя его из исходного списка. Описанный процесс продолжается до тех пор, пока не исчерпаются элементы одного из списков. После этого в результирующий список можно перенести все оставшиеся в исходном списке элементы. Такой алгоритм формально известен под названием *алгоритма двухпутевого слияния (two-way merge algorithm)*.

Конечно, на практике элементы не удаляются из исходных списков. Вместо удаления используются указатели на текущие начальные элементы списков, которые при копировании передвигаются на следующий элемент.

Листинг 5.11. Слияние двух отсортированных массивов TList

```
procedure TListMerge(aList1, aList2, aTargetList : TList;  
                     aCompare : TtdCompareFunc);  
  
var  
    Inx1, Inx2, Inx3 : integer;  
  
begin  
    {подготовить результирующий список}  
    aTargetList.Clear;  
    aTargetList.Capacity := aList1.Count + aList2.Count;
```

```

{инициализировать счетчики}
Inx1 := 0;
Inx2 := 0;
Inx3 := 0;
{выполнять цикл до исчерпания элементов одного из списка...}
while (Inx1 < aList1.Count) and (Inx2 < aList2.Count) do begin
    {определить наименьшее значение из двух списков и скопировать его в резуль-
    тирующий список; увеличить значения индексов на единицу}
    if aCompare(aList1.List^[Inx1], aList2.List^[Inx2]) <= 0 then begin
        aTargetList.List^[Inx3] := aList1.List^[Inx1];
        inc(Inx1);
    end
    else begin
        aTargetList.List^[Inx3] := aList2.List^[Inx2];
        inc(Inx2);
    end;
    inc(Inx3);
end;
{выполнение цикла прекращается при исчерпании элементов одного из списков;
если в первом списке еще остались элементы, скопировать их в результирующий
список}
if (Inx1 < aList1.Count) then
    Move(aList1.List^[Inx1], aTargetList.List^[Inx3],
        (aList1.Count - Inx1) * sizeof(pointer))
    {в противном случае скопировать все элементы, оставшиеся во втором списке,
    в результирующий список}
else
    Move(aList2.List^[Inx2], aTargetList.List^[Inx3],
        (aList2.Count - Inx2) * sizeof(pointer));
end;

```

Обратите внимание, что в коде копирование оставшихся элементов в одном или другом списке выполняется с помощью процедуры **Move**. Для копирования можно было бы организовать небольшой цикл, однако процедура **Move** работает намного быстрее.

Время выполнения алгоритма двухпутевого слияния зависит от количества элементов в обоих исходных списках. Если в первом из них находится n элементов, а во втором – m , нетрудно прийти к выводу, что в худшем случае будет произведено $(n + m)$ сравнений. Следовательно, алгоритм двухпутевого слияния принадлежит к классу $O(n)$.

Каким же образом алгоритм двухпутевого слияния помогает выполнить сортировку? Для его работы необходимо иметь два отсортированных списка меньшей длины, из которых создается один больший список. На основе такого описания можно прийти к рекурсивному определению сортировки слиянием: разделите ис-

ходный список на две половины, примените к каждой половине алгоритм сортировки слиянием, а затем с помощью алгоритма слияния объедините подсписки в один отсортированный список. Рекурсия заканчивается, когда под-под-подсписок, переданный алгоритму сортировки, содержит всего один элемент, поскольку он, очевидно, является отсортированным.

Сортировка слиянием обладает только одним недостатком – алгоритм слияния требует наличия *третьего* списка, в котором будут храниться результаты слияния.

В отличие от всех ранее рассмотренных методов сортировки, которые сортируют элементы непосредственно в самом исходном списке, сортировка слиянием для работы требует большого дополнительного объема памяти. В качестве первого приближения в самой простой реализации может показаться, что для выполнения сортировки понадобится новый вспомогательный список, размер которого равен сумме размеров двух исходных списков. Элементы из обоих списков будут помещаться во вспомогательный список, а затем после слияния – в основной список. Несмотря на то что можно разработать алгоритм, который выполняет операцию слияния, не требуя вспомогательного списка, на практике его выполнение занимает намного больше времени. Поэтому при необходимости применения сортировки слиянием нужно смириться с дополнительными требованиями в отношении памяти.

Сколько же памяти потребуется? Только что мы решили, что в худшем случае будет использоваться список, размер которого равен размеру исходного списка, но за счет небольшой хитрости можно снизить требования по дополнительной памяти до половины размера исходного списка.

Представьте себе, что мы находимся на самом верхнем уровне рекурсивного алгоритма. Только что мы выполнили сортировку двух половин исходного списка (будем считать, что первый отсортированный подсписок находится в первой половине списка, а второй – во второй половине), а теперь переходим к их слиянию. Вместо того чтобы выполнить слияние во вспомогательный список, равный по размеру исходному, скопируем первую половину списка в другой список, размер которого равен только половине исходного. Теперь у нас есть вспомогательный список, заполненный элементами из первой половины исходного списка, и исходный список, первая половина которого считается пустой, а вторая заполнена вторым подсписком элементов. При слиянии мы не перезапишем ни один из элементов второго подсписка, поскольку точно известно, что все содержимое вспомогательного списка может поместиться в свободную половину исходного списка.

Листинг 5.12. Стандартная сортировка слиянием

```
procedure MSS(aList : TList;  
              aFirst : integer;  
              aLast  : integer;  
              aCompare : TtdCompareFunc;  
              aTempList : PPointerList);
```

```

var
  Mid : integer;
  i, j : integer;
  ToInx : integer;
  FirstCount : integer;
begin
  {вычислить среднюю точку}
  Mid := (aFirst + aLast) div 2;
  {выполнить рекурсивную сортировку слиянием первой и второй половин списка}
  if (aFirst < Mid) then
    MSS(aList, aFirst, Mid, aCompare, aTempList);
  if (succ(Mid) < aLast) then
    MSS(aList, succ(Mid), aLast, aCompare, aTempList);
  {скопировать первую половину списка во вспомогательный список}
  FirstCount := succ(Mid - aFirst);
  Move(aList.List^[aFirst], aTempList^[0], FirstCount * sizeof(pointer));
  {установить значения индексов: i - индекс для вспомогательного списка (т.е.
    первой половины списка), j - индекс для второй половины списка,
    ToInx - индекс в результирующем списке, куда будут копироваться отсортиро-
    ванные элементы}
  i := 0;
  j := succ(Mid);
  ToInx := aFirst;
  {выполнить слияние двух списков}
  {повторять до тех пор, пока один из списков не опустеет}
  while (i < FirstCount) and (j <= aLast) do begin
    {определить элемент с наименьшим значением из следующих элементов в обоих
      списках и скопировать его; увеличить значение соответствующего индекса}
    if (aCompare(aTempList^[i], aList.List^[j]) <= 0) then begin
      aList.List^[ToInx] := aTempList^[i];
      inc(i);
    end
    else begin
      aList.List^[ToInx] := aList.List^[j];
      inc(j);
    end;
    {в объединенном списке есть еще один элемент}
    inc(ToInx);
  end;
  {если в первом списке остались элементы, скопировать их}
  if (i < FirstCount) then
    Move(aTempList^[i], aList.List^[ToInx],
      (FirstCount - i) * sizeof(pointer));
  {если во втором списке остались элементы, то они уже находятся в нужных пози-
    циях, значит, сортировка завершено; если второй список пуст, сортировка
    также завершена}
end;

```

```
procedure TDMergeSortStd(aList : TList;
                        aFirst : integer;
                        aLast : integer;
                        aCompare : TtdCompareFunc);
var
    TempList : PPointerList;
    ItemCount: integer;
begin
    TDValidateListRange(aList, aFirst, aLast, 'TDMergeSortStd');
    {если есть хотя бы два элемента для сортировки}
    if (aFirst < aLast) then begin
        {создать временный список указателей}
        ItemCount := succ(aLast - aFirst);
        GetMem(TempList, (succ(ItemCount) div 2) * sizeof(pointer));
        try
            MSS(aList, aFirst, aLast, aCompare, TempList);
        finally
            FreeMem(TempList, (succ(ItemCount) div 2) * sizeof(pointer));
        end;
    end;
end;
```

Если вы внимательно изучите код, приведенный в листинге 5.12, то обнаружите, что он содержит процедуру-драйвер, **TDMergeSortStd**, которая вызывается для выполнения сортировки списка, и отдельную вспомогательную процедуру, **MSS**, выполняющую рекурсивную сортировку. Прежде всего, процедура **TDMergeSortStd** проверяет попадание индекса в допустимые пределы и сам список, а затем – присутствуют ли в списке хотя бы два элемента, которые можно сортировать. После этого создается вспомогательный список указателей с размером, достаточным для хранения половины количества элементов исходного массива. Далее вызывается рекурсивная процедура **MSS**.

Процедура **MSS** рекурсивно вызывает сама себя для сортировки первой и второй половин переданной ей части массива. Затем она копирует первую половину во вспомогательный массив. Начиная с этого момента, код представляет собой стандартную реализацию сортировки слиянием, копируя две половины списка в исходный список. Если после выполнения цикла сравнения и копирования во вспомогательном массиве остались элементы, процедура **MSS** их просто копирует. Если же элементы остались во второй половине списка, их можно не копировать, как и в стандартной реализации метода слияния: они уже находятся на своих местах.

Вывод функции быстрого действия сортировки слиянием достаточно сложен. Для простоты ее определения лучше принять, что в исходном списке находится 2^x элементов. Предположим, что элементов 32. На первом уровне рекурсии процедура **MSS** будет вызываться один раз и на этапе слияния будет не более 32 сравнений. На втором уровне рекурсии процедура **MSS** будет вызываться два раза, причем ко-

личество сравнений при каждом вызове не будет превышать 16. Далее рассматриваем третий, четвертый и, наконец, пятый уровень рекурсии (когда будет выполняться сортировка всего двух элементов), на котором будет иметь место 16 вызовов процедуры по два сравнения в каждом. Таким образом, общее количество сравнений будет равно $5 * 32$. Но причиной, по которой было получено пять уровней рекурсии, является то, что мы постоянно на каждом уровне постепенно делили список на две равные половины, а $2^5 = 32$, что, естественно означает, что $\log_2 32 = 5$. Следовательно, не утруждая себя переходом от рассмотренного нами частного случая к общему, можно сказать, что сортировка слиянием принадлежит к классу $O(n \log(n))$ алгоритмов.

Что касается устойчивости, то поскольку элементы перемещаются только при выполнении процедуры слияния, устойчивость всей сортировки слиянием будет зависеть от устойчивости самого слияния двух половин списка. Обратите внимание, что если в обеих половинах имеются элементы с одинаковым значением, то оператор сравнения гарантирует, что первым в результирующий список попадет элемент из первой половины списка. Это означает, что операция слияния сохраняет относительный порядок элементов, и, следовательно, сортировка слиянием будет устойчивой.

Если отслеживать вызовы процедуры **MSS** в отладчике, то можно обратить внимание, что для небольших интервалов она вызывается очень часто. Например, если в списке содержится 32 элемента, то для списка из 32 элементов процедура **MSS** будет вызвана один раз, для списка из 16 элементов – дважды, четыре раза для 8 элементов, восемь раз для 4 элементов и шестнадцать раз для 2 элементов (список минимальной длины), т.е. всего 31 раз. Это очень много, особенно если учитывать, что большая часть вызовов (29) приходится для списков длиной восемь и менее элементов. Если бы исходный список содержал 1024 элемента, процедура **MSS** была бы вызвана 1023 раза, из которых 896 вызовов приходилось бы на долю списков длиной восемь и менее элементов. Просто ужасно! Фактически, для сортировки коротких списков было бы эффективнее использовать нерекурсивный алгоритм. Это позволило бы повысить скорость всей сортировки. Кроме того, применение более простой процедуры дало бы возможность для коротких диапазонов исключить копирование элементов между основным и вспомогательным списком. И одним из лучших методов для ускорения сортировки слиянием является сортировка методом вставок.

Листинг 5.13. Оптимизированная сортировка слиянием

```
const
    MSCutOff = 16;
procedure MSInsertionSort(aList : TList;
    aFirst : integer;
    aLast : integer;
    aCompare : TtdCompareFunc);
```

```

var
  i, j : integer;
  IndexOfMin : integer;
  Temp : pointer;
begin
  {найти наименьший элемент в списке}
  IndexOfMin := aFirst;
  for i := succ(aFirst) to aLast do
    if (aCompare(aList.List^[i], aList.List^[IndexOfMin]) < 0) then
      IndexOfMin := i;
  if (aFirst <> IndexOfMin) then begin
    Temp := aList.List^[aFirst];
    aList.List^[aFirst] := aList.List^[IndexOfMin];
    aList.List^[IndexOfMin] := Temp;
  end;
  {выполнить сортировку методом вставок}
  for i := aFirst+2 to aLast do begin
    Temp := aList.List^[i];
    j := i;
    while (aCompare(Temp, aList.List^[j-1]) < 0) do begin
      aList.List^[j] := aList.List^[j-1];
      dec(j);
    end;
    aList.List^[j] := Temp;
  end;
end;

procedure MS(aList : TList;
              aFirst : integer;
              aLast : integer;
              aCompare : TtdCompareFunc;
              aTempList : PPointerList);

var
  Mid : integer;
  i, j : integer;
  ToInx : integer;
  FirstCount : integer;
begin
  {вычислить среднюю точку}
  Mid := (aFirst + aLast) div 2;
  {выполнить сортировку первой половины списка с помощью сортировки слиянием
  или, если количество элементов достаточно мало, при помощи сортировки
  методом вставок}
  if (aFirst < Mid) then
    if (Mid-aFirst) <= MSCutOff then
      MSInsertionSort(aList, aFirst, Mid, aCompare)
    else
      MS(aList, aFirst, Mid, aCompare, aTempList);

```

```

{аналогично выполнить сортировку второй половины}
if (succ(Mid) < aLast) then
  if (aLast-succ(Mid)) <= MSCutOff then
    MSInsertionSort(aList, succ(Mid), aLast, aCompare)
  else
    MS(aList, succ(Mid), aLast, aCompare, aTempList);
{скопировать первую половину списка во вспомогательный список}
FirstCount := succ(Mid - aFirst);
Move(aList.List^[aFirst], aTempList^[0], FirstCount*sizeof(pointer));
{установить значения индексов: i - индекс для вспомогательного списка (т.е.
  первой половины списка), j - индекс для второй половины списка, ToInx -
  индекс в результирующем списке, куда будут копироваться отсортированные
  элементы}
i := 0;
j := succ(Mid);
ToInx := aFirst;
{выполнить слияние двух списков}
{повторять до тех пор, пока один из списков не опустеет}
while (i < FirstCount) and (j <= aLast) do begin
  {определить элемент с наименьшим значением из следующих элементов в обоих
    списках и скопировать его; увеличить значение соответствующего индекса}
  if (aCompare(aTempList^[i], aList.List^[j]) <= 0) then begin
    aList.List^[ToInx] := aTempList^[i];
    inc(i);
  end
  else begin
    aList.List^[ToInx] := aList.List^[j];
    inc(j);
  end;
  {в объединенном списке есть еще один элемент}
  inc(ToInx);
end;
{если в первом списке остались элементы, скопировать их}
if (i < FirstCount) then
  Move(aTempList^[i], aList.List^[ToInx],
    (FirstCount - i) * sizeof(pointer));
{если во втором списке остались элементы, то они уже находятся в нужных
  позициях, значит, сортировка завершена; если второй список пуст, сортировка
  также завершена}
end;

procedure TDMergeSort(aList : TList;
  aFirst : integer;
  aLast : integer;
  aCompare : TtdCompareFunc);

var
  TempList : PPointerList;
  ItemCount: integer;

```



```
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDMergeSort');
  {если есть хотя бы два элемента для сортировки}
  if (aFirst < aLast) then begin
    {создать временный список указателей}
    ItemCount := succ(aLast - aFirst);
    GetMem(TempList, (succ(ItemCount) div 2) * sizeof(pointer));
    try
      MS(aList, aFirst, aLast, aCompare, TempList);
    finally
      FreeMem(TempList, (succ(ItemCount) div 2) * sizeof(pointer));
    end;
  end;
end;
```

Несмотря на то что объем кода достаточно велик, в нем находятся всего три процедуры. Прежде всего, драйвер – **TDMergeSort** – процедура, которую мы вызываем. Как и в предыдущем случае, она используется для выделения из кучи памяти под вспомогательный список указателей и вызывает рекурсивную процедуру, названную в приведенном коде **MS**. В общих чертах процедура **MS** работает примерно так, как и ее предшественница – **MSS** (рекурсивная процедура для стандартной сортировки слиянием). Разница возникает только тогда, когда дело касается сортировки подсписков. Для небольших диапазонов элементов, длина которых меньше, чем значение **MSCutoff**, процедура **MS** вызывает третью процедуру, **MSInsertionSort**, которая сортирует элементы без рекурсивного вызова. Для длинных диапазонов элементов, естественно, происходит рекурсивный вызов процедуры **MS**. **MSInsertionSort** ничем не отличается от рассмотренной нами ранее процедуры **TDInsertionSort**, за исключением одного – она не проверяет корректность входных параметров (в проверке нет необходимости, поскольку все параметры были проверены в **TDMergeSort**).

Поскольку в приведенном коде для сортировки коротких диапазонов в списке используется сортировка методом вставок, которая сама по себе является устойчивой, можно сказать, что оптимизированная сортировка слиянием также принадлежит к группе устойчивых алгоритмов.

Несмотря на то что сортировка слиянием требует дополнительной памяти (объем которой пропорционален количеству элементов в исходном списке), она обладает некоторыми интересными свойствами. Первое из них – сортировка слиянием принадлежит к классу $O(n \log(n))$. Второе – она устойчива. Еще два алгоритма со скоростью работы $O(n \log(n))$ и дополнительными требованиями к памяти, которые будут рассмотрены в этой главе, являются неустойчивыми. Третье – для сортировки слиянием не имеет значения ни порядок элементов в исходном списке (будь то список, отсортированный в прямом порядке или обратном), ни повторения значений в списке. Другими словами, она не имеет худшего случая.

В конце этой главы мы рассмотрим случай, в котором сортировка слиянием просто необходима, – сортировка связного списка.

И, наконец, сортировка слиянием используется для сортировки содержимого файлов, размер которых слишком велик, чтобы поместиться в памяти. В этой ситуации выполняется сортировка частей файлов, запись этих частей в отдельные файлы, а затем их слияние в один файл.

Быстрая сортировка

И последний алгоритм, который будет рассмотрен в этой главе – *быстрая сортировка* (*quicksort*). (В книге мы опишем еще одну сортировку "в памяти" – пирамидальную сортировку, но она требует дополнительных знаний структуры данных – бинарного дерева. По этой причине рассмотрение пирамидальной сортировки отложено до главы 9.)

Алгоритм быстрой сортировки был разработан К.А.Р. Хоаром (C.A.R. Hoare) в 1960 году. Этот алгоритм, наверное, еще более известен, чем пузырьковая сортировка. В настоящее время он является самым широко используемым в программировании методом сортировки, что вызвано его крайне положительными характеристиками: это алгоритм класса $O(n \log(n))$ для общего случая, он требует лишь незначительного объема дополнительной памяти, работает с различными типами входных списков и достаточно удобен для реализации. Но, к сожалению, быстрая сортировка имеет и несколько *нежелательных* характеристик: при его реализации допускается очень много ошибок (простые ошибки в реализации могут остаться незамеченными и при выполнении могут потребовать дополнительного времени), быстроедействие в худшем случае составляет $O(n^2)$ и к тому же она неустойчива.

Кроме того, быстрая сортировка наиболее изучена. Со времени выхода в свет первой статьи Хоара многие исследователи изучали быструю сортировку и сформировали значительную базу данных по теоретическому определению времени выполнения, подкрепленную эмпирическими данными. Было предложено немало улучшений базового алгоритма, позволяющих увеличить скорость работы. Некоторые из предложенных улучшений будут рассмотрены в этой главе. При таком богатстве литературных источников по алгоритму быстрой сортировки, если следовать всем рекомендациям, у вас не должно возникнуть проблем с реализацией. (В последней оптимизированной реализации алгоритма использовалось более шести различных справочных пособий по алгоритмам. Причем в одной из них была приведена "оптимизированная" быстрая сортировка, которая была написана так плохо, что при одних и тех же входных данных работала даже медленнее, чем стандартный метод **TList.Sort**.)

Быстрая сортировка встречается везде. Во всех версиях Delphi, за исключением версии 1, метод **TList.Sort** реализован на основе алгоритма быстрой сортировки. Метод **TStringList.Sort** во всех версиях Delphi реализован с помощью быстрой сортировки. В C++ функция **qsort** из стандартной библиотеки времени выполнения также реализована на базе быстрой сортировки.

Основной алгоритм быстрой сортировки, как и сортировку слиянием, можно отнести к классу "разделяй и властвуй". Он *разбивает* исходный список на два, а затем для выполнения сортировки рекурсивно вызывает сам себя для каждой части списка. Таким образом, особое внимание в быстрой сортировке нужно уделить процессу деления. В разбитом списке происходит следующее: выбирается элемент, называемый *базовым*, относительно которого переставляются элементы в списке. Элементы, значения которых меньше, чем значение базового элемента, переносятся левее базового, а элементы, значения которых больше, чем значение базового элемента, переносятся правее базового. После этого можно сказать, что базовый элемент находится на своем месте в отсортированном списке. Затем выполняется рекурсивный вызов функции быстрой сортировки для левой и правой частей списка (относительно базового элемента). Рекурсивные вызовы прекращаются, когда список, переданный функции сортировки, будет содержать всего один элемент, а, следовательно, весь список оказывается отсортированным.

Таким образом, для выполнения быстрой сортировки необходимо знать два алгоритма более низкого уровня: как выбирать базовый элемент и как наиболее эффективно переставить элементы списка таким образом, чтобы получить два набора элементов: со значениями, меньшими, чем значение базового элемента, и со значениями, большими, чем значение базового элемента.

Начнем с описания алгоритма выбора базового элемента. В идеале следовало бы выбирать средний элемент списка. Затем при разбиении количество элементов в наборе значений, меньших значения базового элемента, будет равно количеству элементов в наборе значений, больших значения базового элемента. Другими словами, при разбиении исходный список был бы разделен на две равные половины. Вычисление среднего элемента списка (или его медианы) представляет собой достаточно сложный процесс, к тому же стандартный алгоритм его определения использует метод разбиения быстрой сортировки, который мы сейчас обсуждаем. Поэтому нам придется отказаться от определения среднего элемента списка.

Худшим случаем будет иметь место, если в качестве базового элемента мы выберем элемент с максимальным или минимальным значением. В этом случае после выполнения процесса разбиения один из результирующих списков будет пуст, а во втором будут содержаться все элементы, поскольку все они будут находиться по одну сторону от базового элемента. Конечно, заранее (по крайней мере, без просмотра списка) невозможно узнать, выбран ли элемент с минимальными или максимальным значением, но если при каждом рекурсивном вызове в качестве базового элемента будет выбираться один из граничных элементов, то для n элементов будет выполнено n уровней рекурсии. При большом количестве сортируемых элементов это может вызывать проблемы. (При реализации алгоритма быстрой сортировки особое внимание следует уделить исключению возможности заикливания рекурсивных вызовов.)

Таким образом, после рассмотрения этих двух граничных случаев можно сказать, что желательно выбирать базовый элемент, который был бы как можно ближе к среднему элементу и как можно дальше от минимального и максимального.

Во многих книгах в качестве базового элемента выбирается первый или последний элемент списка. Если в исходном списке элементы располагались в произвольном порядке, стратегия выбора первого или последнего элемента ничем не отличается от любой другой. Но если исходный список был отсортирован в прямом или обратном порядке, выбор в качестве базового элемента первого или последнего элемента списка приводит нас к наихудшему случаю для алгоритма быстрой сортировки. Следовательно, первый или последний элемент нежелательно выбирать в качестве базового. Никогда так не делайте.

Намного лучше в качестве базового элемента брать средний элемент исходного списка. Остается только надеяться, что он будет находиться вблизи среднего элемента списка. В списке, элементы которого не упорядочены, выбор базового элемента не имеет значения, но если список уже отсортирован в прямом или обратном порядке, средний элемент будет лучшим выбором.

После выбора базового элемента можно перейти к описанию алгоритма разбиения списка. Добро пожаловать в известные своей быстротой внутренние циклы быстрой сортировки! Мы будем оперировать с двумя индексами: первый будет использоваться для прохождения по элементам списка слева направо, а второй – справа налево. Начинаем справа, и идем к левому краю списка, сравнивая значение каждого элемента со значением базового элемента. Выполнение цикла завершается, если найден элемент, значение которого меньше или равно значению базового элемента. Это был внутренний цикл 1: сравнение двух элементов и уменьшение значения индекса. Затем та же операция выполняется слева. Проход выполняется в направлении к правому концу списка. Значение каждого элемента сравнивается со значением базового элемента. Цикл завершается, если найден элемент, значение которого больше или равно значению базового элемента. Это внутренний цикл 2: сравнение двух элементов и увеличение значения индекса.

На этом этапе могут возникнуть две ситуации. Первая – левый индекс меньше правого. Это говорит о том, что два элемента, на которые указывают индексы, расположены в списке в неверном порядке (т.е. значение элемента слева больше значения базового элемента, а значение элемента справа меньше значения базового элемента). Меняем элементы местами и продолжаем выполнение внутренних циклов. Вторая ситуация – индексы равны (т.е. значение левого индекса равно значению правого индекса) или индексы пересеклись (т.е. значение левого индекса больше значения правого индекса). В таком случае выполнение циклов можно завершить: список был успешно разделен.

Листинг 5.14. Стандартная быстрая сортировка

```
procedure QSS(aList : TList;  
              aFirst : integer;  
              aLast : integer;  
              aCompare : TtdCompareFunc);  
  
var  
  L, R : integer;  
  Pivot : pointer;  
  Temp : pointer;  
begin  
  {пока в списке есть хотя бы два элемента}  
  while (aFirst < aLast) do begin  
    {в качестве базового элемента выбирается средний элемент списка}  
    Pivot := aList.List^[ (aFirst+aLast) div 2];  
    {задать начальные значения индексов и приступить к разбиению списка}  
    L := pred(aFirst);  
    R := succ(aLast);  
    while true do begin  
      repeat dec(R); until (aCompare(aList.List^[R], Pivot) <= 0);  
      repeat inc(L); until (aCompare(aList.List^[L], Pivot) >= 0);  
      if (L >= R) then Break;  
      Temp := aList.List^[L];  
      aList.List^[L] := aList.List^[R];  
      aList.List^[R] := Temp;  
    end;  
    {выполнить быструю сортировку первого подфайла}  
    if (aFirst < R) then  
      QSS(aList, aFirst, R, aCompare);  
    {выполнить быструю сортировку второго подфайла – устранение рекурсии}  
    aFirst := succ(R);  
  end;  
end;  
  
procedure TDQuickSortStd(aList : TList;  
                        aFirst : integer;  
                        aLast : integer;  
                        aCompare : TtdCompareFunc);  
  
begin  
  TDValidateListRange(aList, aFirst, aLast, 'TDQuickSortStd');  
  QSS(aList, aFirst, aLast, aCompare);  
end;
```

Поскольку алгоритм рекурсивный, быстрая сортировка в приведенной реализации разбита на две процедуры, как это имело место в случае сортировки слиянием. Первая процедура, **TDQuickSortStd**, – это процедура-драйвер. Она проверяет корректность задания входных параметров и вызывает вторую процедуру – **QSS**.

Именно она является рекурсивной, и именно она – суть всей реализации. Первое, что необходимо отметить, – процедура **QSS** работает только в том случае, когда в сортируемом списке есть хотя бы два элемента. В качестве базового элемента выбирается средний элемент списка. Затем устанавливаются начальные значения для индексов **L** и **R** – перед первым элементом и после последнего элемента списка соответственно. После этого в процедуре организован бесконечный цикл – при необходимости мы сами выйдем из него с помощью оператора **break**. Обратите внимание, что внутренние циклы организованы с помощью операторов **Repeat...until**. В первом цикле уменьшается значение индекса **R** до тех пор, пока он не будет указывать на элемент, значение которого меньше или равно значению базового элемента. Во втором цикле значение индекса **L** увеличивается до тех пор, пока он не будет указывать на элемент, значение которого больше или равно значению базового элемента. Затем сравниваются значения индексов **L** и **R**. Если значение **L** больше или равно значению **R**, индексы встретились или пересеклись, и мы выходим из бесконечного цикла. В противном случае два элемента, на которые указывают индексы, меняются местами, и выполнение цикла продолжается.

После выхода из бесконечного цикла во многих реализациях алгоритма быстрой сортировки присутствует примерно следующий код:

```
QSS(aList, aFirst, R, aCompare);  
QSS(aList, R+1, aList, aCompare);
```

Другими словами, рекурсивно выполняется сортировка первой части, а затем – второй части раздела. Один из самых простых хитростей искусства программирования заключается в исключении рекурсивного вызова в конце рекурсивной процедуры. Как правило, бывает достаточно изменения переменных в процедуре и перехода к ее началу. В **QSS** для исключения рекурсии используется цикл **while** при изменении значения переменной **aFirst**. Очень простой метод устранения рекурсии, не правда ли?

После изучения такого рода процедур, особенно процедур с циклами, любой программист начнет искать способы увеличения ее эффективности. В случае с быстрой сортировкой лучше не пытайтесь изменять приведенную процедуру. Даже незначительное изменение может привести к существенному снижению скорости работы или к тому, что бесконечный цикл оправдает свое название. Давайте рассмотрим несколько часто встречаемых ошибок. Первое желание – установить начальные значения индексов **L** и **R** так, чтобы они указывали на фактические первый и последний элементы списка, а не на предшествующий первому и следующий после последнего элементы, а затем заменить цикл **Repeat...until** циклом **while**, естественно, изменив условие цикла. По крайней мере, в этом случае можно исключить одну операцию декремента и одну – инкремента. Первый цикл превратится в цикл "выполнять, пока больше чем", а второй – в цикл "выполнять, пока меньше чем". Если на вход такой "улучшенной" процедуры быстрой сортировки подавать список с произвольных расположением элементов, он будет

работать без ошибок. Но для списка, все элементы которого равны, бесконечный цикл будет действительно выполняться бесконечно, поскольку значения индексов меняться не будут. Изменение условий циклов включает равенство, позволяет избежать заикливания процедуры, но приводит к еще одной проблеме: индексы будут выходить за границы списка.

Последнее утверждение требует более подробного обсуждения. За счет выбора среднего элемента списка в качестве базового мы не только избегаем худшего случая, но и гарантируем, что два внутренних цикла будут заканчиваться при допустимых значениях индексов. Базовый элемент представляет собой сигнальное значение для обоих внутренних циклов. Даже в худшем случае каждый цикл будет завершаться после достижения базового элемента. Если бы в качестве базового был бы выбран, например, первый или последний элемент, пришлось бы изменить один из циклов для проверки того, что индекс не выходит за допустимые пределы (для другого цикла границей служил бы базовый элемент).

Есть надежда, что приведенное выше описание некоторых часто встречаемых ошибок при реализации алгоритма быстрой сортировки позволит лучше понять сам алгоритм, даже несмотря на то, что его реализация содержит всего несколько строк кода. Экспериментируйте, просмотрите реализацию быстрой сортировки в методе **TList.Sort** в исполнении компании Borland, но будьте осторожны и тестируйте результаты своих экспериментов на различных типах списков.

Теперь, когда вы предупреждены о возможных последствиях внесения изменений в реализацию алгоритма быстрой сортировки, давайте аккуратно модернизируем приведенную реализацию.

Прежде всего, давайте изучим влияние выбора базового элемента на быстродействие алгоритма. Если вы помните, в нашей первой процедуре быстрой сортировки в качестве базового элемента выбирался средний элемент. До этого мы коротко рассмотрели и отклонили выбор первого и последнего элемента списка. В идеальном случае следовало бы каждый раз выбирать средний элемент отсортированного списка или, в крайнем случае, избегать выбора в качестве базового элемента с минимальным и максимальным значением (поскольку в этом случае быстрая сортировка вырождается в длинную серию пустых подсписков и подсписков с одним или меньшим количеством элементов). Часто в качестве базового элемента выбирается случайный элемент. Затем этот элемент меняется местом со средним элементом, и алгоритм выполняется, как и в случае выбора среднего элемента.

Что дает нам случайный выбор базового элемента? При условии, что у нас есть достаточно хороший генератор псевдослучайных чисел, такой выбор гарантирует, что вероятность попадания на "худший" элемент становится пренебрежительно малой. Но, тем не менее, она не превращается в 0, просто выбор наихудшего для быстрой сортировки элемента в качестве базового становится весьма маловероятным.

Листинг 5.15. Быстрая сортировка со случайным выбором базового элемента

```

procedure QSR(aList : TList;
               aFirst : integer;
               aLast : integer;
               aCompare : TtdCompareFunc);

var
  L, R : integer;
  Pivot : pointer;
  Temp : pointer;
begin
  while (aFirst < aLast) do begin
    {выбрать случайный элемент, переставить его со средним элементом
     и взять в качестве базового элемента}
    R := aFirst + Random(aLast - aFirst + 1);
    L := (aFirst + aLast) div 2;
    Pivot := aList.List^[R];
    aList.List^[R] := aList.List^[L];
    aList.List^[L] := Pivot;
    {задать начальные значения индексов и приступить к разбиению списка}
    L := pred(aFirst);
    R := succ(aLast);
    while true do begin
      repeat dec(R); until (aCompare(aList.List^[R], Pivot) <= 0);
      repeat inc(L); until (aCompare(aList.List^[L], Pivot) >= 0);
      if (L >= R) then Break;
      Temp := aList.List^[L];
      aList.List^[L] := aList.List^[R];
      aList.List^[R] := Temp;
    end;
    {выполнить быструю сортировку первого подфайла}
    if (aFirst < R) then
      QSR(aList, aFirst, R, aCompare);
    {выполнить быструю сортировку второго подфайла – устранение рекурсии}
    aFirst := succ(R);
  end;
end;

procedure TDQuickSortRandom(aList : TList;
                             aFirst : integer;
                             aLast : integer;
                             aCompare : TtdCompareFunc);

begin
  TDValidateListRange(aList, aFirst, aLast, 'TDQuickSortRandom');
  QSR(aList, aFirst, aLast, aCompare);
end;

```

Как видите, различия между стандартным алгоритмом быстрой сортировки и приведенным в листинге 5.15 совсем незначительны. Основное отличие представляет собой вставленный блок кода, который специально выделен в листинге. В нем первый индекс выбирается случайным образом из диапазона от **aFirst** до **aLast** включительно, а затем элемент с выбранным индексом меняется местами со средним элементом. Для удобства в приведенной реализации используется Delphi-функция **Random**. Она предоставляет хорошие последовательности псевдослучайных чисел. Перестановка выбранного и среднего элементов дает преимущества, о которых мы уже говорили.

Несмотря на то что внесенное изменение снижает вероятность выбора "худшего" элемента при каждом проходе цикла, тем не менее, оно не увеличивает скорость выполнения процедуры. Фактически скорость даже падает (как это и можно было предположить). Генерация случайного числа в качестве индекса для базового элемента работает отлично, в том смысле, что вероятность выбора "плохого" элемента в качестве базового снижается, но это положительное свойство не приводит к повышению быстродействия процедуры. Сложность линейного конгруэнтного метода генерации случайных чисел, используемого функцией **Random**, только увеличивает время выполнения процедуры. Можно было бы исследовать быстродействие при использовании различных генераторов (некоторые из них будут рассмотрены в главе 6), но оказывается, что существует намного более удачный алгоритм выбора базового элемента.

Самым эффективным методом выбора базового элемента на сегодняшний день является метод медианы трех. Мы уже говорили, что в идеальном случае желательно было бы выбирать средний элемент (или медиану) всех элементов списка. Тем не менее, определение медианы – достаточно сложная задача. Более простым кажется приближенное определение медианы. Для этого из подсписка выбирается три элемента и в качестве базового элемента выбирается медиана этих трех элементов. Медиана трех элементов служит приближением фактической медианы всех элементов списка. Конечно, такой алгоритм предполагает, что в списке должно быть, по крайней мере, три элемента. Но даже если элементов меньше, выполнить их сортировку не представляет большого труда.

Выбор трех элементов ничем не ограничивается, но имеет смысл выбирать первый, последний и средний элементы. Почему? Такая схема может облегчить весь процесс сортировки. Мы находим не только медиану трех элементов, но и расставляем их в требуемом *порядке*. Элемент с наименьшим значением попадает в первую позицию, средний элемент – в середину списка, а элемент с наибольшим значением – в последнюю позицию. Таким образом, при выборе базового элемента размер частей списка сокращается на два элемента, поскольку уже известно, что они находятся в правильных частях списка относительно базового элемента. Кроме того, такой алгоритм автоматически помещает базовый элемент в нужное место: в середину подсписка.

Листинг 5.16. Быстрая сортировка со случайным выбором базового элемента

```

procedure QSM(aList : TList;
               aFirst : integer;
               aLast : integer;
               aCompare : TtdCompareFunc);

var
  L, R : integer;
  Pivot : pointer;
  Temp : pointer;
begin
  while (aFirst < aLast) do begin
    {если в списке есть, по крайней мере, три элемента, выбрать
     базовый элемент как медиану первого, последнего и среднего
     элементов списка и записать его в позицию в середине списка}
    if (aLast - aFirst) >= 2 then begin
      R := (aFirst + aLast) div 2;
      if (aCompare(aList.List^[aFirst],
                  aList.List^[R]) > 0) then begin
        Temp := aList.List^[aFirst];
        aList.List^[aFirst] := aList.List^[R];
        aList.List^[R] := Temp;
      end;
      if (aCompare(aList.List^[aFirst],
                  aList.List^[aLast]) > 0) then begin
        Temp := aList.List^[aFirst];
        aList.List^[aFirst] := aList.List^[aLast];
        aList.List^[aLast] := Temp;
      end;
      if (aCompare(aList.List^[R],
                  aList.List^[aLast]) > 0) then begin
        Temp := aList.List^[R];
        aList.List^[R] := aList.List^[aLast];
        aList.List^[aLast] := Temp;
      end;
      Pivot := aList.List^[R];
    end
    {в противном случае в списке всего 2 элемента, выбрать в качестве
     базового первый элемент}
    else
      Pivot := aList.List^[aFirst];
    {задать начальные значения индексов и приступить к разбиению списка}
    L := pred(aFirst);
    R := succ(aLast);
    while true do begin
      repeat dec(R); until (aCompare(aList.List^[R], Pivot) <= 0);
      repeat inc(L); until (aCompare(aList.List^[L], Pivot) >= 0);
      if (L >= R) then Break;
    end;
  end;

```

```

    Temp := aList.List^[L];
    aList.List^[L] := aList.List^[R];
    aList.List^[R] := Temp;
end;
{выполнить быструю сортировку первого подфайла}
if (aFirst < R) then
    QSM(aList, aFirst, R, aCompare);
    {выполнить быструю сортировку второго подфайла – устранение рекурсии}
    aFirst := succ(R);
end;
end;
procedure TDQuickSortMedian(aList : TList;
                             aFirst : integer;
                             aLast : integer;
                             aCompare : TtdCompareFunc);
begin
    TDValidateListRange(aList, aFirst, aLast, 'TDQuickSortMedian');
    QSM(aList, aFirst, aLast, aCompare);
end;

```

В этот раз размер дополнительного блока кода (также специальным образом выделенного) намного больше, чем в предыдущем случае. Большая его часть представляет собой код выбора и сортировки элементов для алгоритма медианы трех. Конечно, новый добавленный код выполняется только тогда, когда в списке имеется не менее трех элементов.

Сортировка трех выбранных элементов выполняется на основе одного малоизвестного и малоиспользуемого метода. Предположим, что выбраны элементы a , b и c . Сравниваем a и b . Если b меньше чем a , поменять их местами. Таким образом, получим $a \leq b$. Сравниваем a и c . Если c меньше чем a , поменять их местами. Получим $a \leq c$. После выполнения этих сравнений нам будет известно, что элемент a содержит минимальное значение из трех выбранных элементов, поскольку оно меньше или равно значениям элементов b и c . Сравниваем b и c . Если c меньше чем b , поменять их местами. Теперь элементы расположены в порядке $a \leq b \leq c$, т.е. они отсортированы. Если количество элементов в списке не больше двух, в качестве базового элемента выбирается первый.

Это улучшение, несмотря на кажущуюся медлительность, на практике работает быстрее, чем стандартная быстрая сортировка. Надо признать, увеличение скорости незначительно, но, тем не менее, ощутимо.

Теперь оставим в покое алгоритм выбора базового элемента и перейдем к рассмотрению улучшений, касающихся других аспектов быстрой сортировки. Быстрая сортировка – это рекурсивный алгоритм, и уже было показано, каким образом легко избавиться от одного из рекурсивных вызовов. Исключение второго рекурсивного вызова потребует больших усилий, но это может быть оправдано, поскольку позволит устранить некоторые вызовы, установку стека и т.п.

Рассмотрим рекурсивный вызов. Задаются четыре параметра, два из которых постоянны, а два других от вызова к вызову могут изменяться. Постоянные параметры – **aList** и **aCompare**, переменные параметры – **aFirst** и **aSecond**. Рекурсивный вызов можно устранить за счет использования явного стека, который предназначен для заталкивания и выталкивания переменных параметров. При этом цикл будет выполняться до тех пор, пока стек не будет пустым.

Листинг 5.17. Быстрая сортировка со случайным выбором базового элемента

```

procedure QSNR(aList : TList;
                aFirst : integer;
                aLast : integer;
                aCompare : TtdCompareFunc);

var
    L, R : integer;
    Pivot : pointer;
    Temp : pointer;
    Stack : array [0..63] of integer; {позволяет разместить до
                                       2 миллиардов элементов}

    SP : integer;
begin
    {инициализировать стек}
    Stack[0] := aFirst;
    Stack[1] := aLast;
    SP := 2;
    while (SP <> 0) do begin
        {вытолкнуть верхний подфайл}
        dec(SP, 2);
        aFirst := Stack[SP];
        aLast := Stack[SP+1];
        {пока в списке есть хотя бы два элемента}
        while (aFirst < aLast) do begin
            {в качестве базового выбирается средний элемент}
            Pivot := aList.List^[ (aFirst+aLast) div 2 ];
            {задать начальные значения индексов и приступить к разбиению списка}
            L := pred(aFirst);
            R := succ(aLast);
            while true do begin
                repeat dec(R); until (aCompare(aList.List^[R], Pivot) <= 0);
                repeat inc(L); until (aCompare(aList.List^[L], Pivot) >= 0);
                if (L >= R) then Break;
                Temp := aList.List^[L];
                aList.List^[L] := aList.List^[R];
                aList.List^[R] := Temp;
            end;
        end;
    end;

```



```

    {затолкнуть больший подфайл в стек и повторить цикл для меньшего
    подфайла}
    if (R - aFirst) < (aLast - R) then begin
        Stack[SP] := succ(R);
        Stack[SP+1] := aLast;
        inc(SP, 2);
        aLast := R;
    end
    else begin
        Stack[SP] := aFirst;
        Stack[SP+1] := R;
        inc(SP, 2);
        aFirst := succ(R);
    end;
end;
end;
end;
end;

procedure TDQuickSortNoRecurse(aList : TList;
                                aFirst : integer;
                                aLast : integer;
                                aCompare : TtdCompareFunc);

begin
    TDValidateListRange(aList, aFirst, aLast, 'TDQuickSortNoRecurse');
    QSNR(aList, aFirst, aLast, aCompare);
end;

```

И в этом листинге код состоит из двух процедур: процедуры-драйвера и процедуры собственно сортировки. Таким образом, общая схема работы алгоритма осталась неизменной, но в этом случае внутренняя процедура, **QSNR**, вызывается только один раз.

В процедуре **QSNR** объявляется стек **Stack** для хранения 64 элементов типа **longint** и указатель на стек **SP**, который будет указывать на начало стека. Комментарий жизнерадостно утверждает, что размера стека будет достаточно для хранения 2 миллиардов элементов. Через несколько минут мы докажем справедливость комментария. В начале процедуры в стек записываются переданные процедуре начальный и конечный индексы. Предполагается, что на индекс первого элемента указывает указатель стека, а индекс последнего элемента хранится в позиции **SP+1**. После записи индексов указатель стека перемещается на 2 позиции. (В реализации алгоритма может использоваться два стека: один для индексов **aFirst**, а второй – для индексов **aLast**. При этом для обоих стеков будет нужен только один указатель.)

Далее начинается выполнение цикла **while**, которое завершается, когда стек опустеет, что эквивалентно равенству **SP=0**.

В цикле из стека выталкиваются переменные **aFirst** и **aLast** и значение указателя стека уменьшается на 2. После этого мы входим в цикл, который присутствует и в стандартной быстрой сортировке. Он повторяется до тех пор, пока значение индекса **aFirst** не превысит значение индекса **aLast**. Заключительные операторы в цикле, где ранее находился рекурсивный вызов процедуры сортировки, представляют собой интересный блок кода. К этому моменту времени базовый элемент находится на своем месте, и подписание успешно разбит на две части. Определяем, какая из частей длиннее и записываем ее в стек (т.е. заталкиваем в стек значения индексов его первого и последнего элемента) и переходим к меньшей части.

Давайте на минутку задумаемся, что происходит. Если нам несказанно повезло, и для каждого подписки в качестве базового элемента были выбраны их действительные медианы, то размеры подписок будут составлять ровно половину размера подписки более высокого уровня. Если в исходном списке было, например, 32 элемента, он будет разбит на 2 подписки по 16 элементов, каждый из которых, в свою очередь, будет разбит еще на два подписки по 8 элементов и т.д. Таким образом, максимальная глубина вложения подписок в стеке будет равна пяти, поскольку $2^5=32$. Подумайте над этим. Мы затолкнем в стек подписание из 16 элементов, разобьем второй такой же список на два списка по 8 элементов, затолкнем в стек один из списков длиной 8 элементов, а второй разобьем на два подписки по 4 элемента и т.д. Пока мы дойдем до подписки с одним элементом, в стеке уже будут находиться подписание из 16 элементов, подписание из 8 элементов, подписание из 4 элементов, подписание из 2 элементов и подписание из 1 элемента. Пять уровней. Таким образом, для сортировки списка, содержащего 2 миллиарда элементов, будет достаточно 32 уровней (как это указано в комментарии к процедуре **QSNR**), если, конечно, каждый раз мы будем удачно выбирать базовый элемент.

Однако приведенное выше доказательство справедливо только в том случае, если нам очень повезет, не правда ли? На самом деле, нет. Если каждый раз в стек помещать больший подписание, а продолжать работать с меньшим, то глубину вложения подписок будет определять именно меньший подписание. Поскольку размер меньшего подписки будет всегда меньше или равен половине разбиваемого списка, результирующая глубина стека не будет превышать глубину стека для описанного выше случая удачного выбора базового элемента. Таким образом, размера объявленного в процедуре стека окажется вполне достаточно.

Обратите внимание, что такое же улучшение можно было ввести и в рекурсивный алгоритм сортировки. При этом внутренняя процедура быстрой сортировки вызывалась бы для меньшего списка. Внесенное нами небольшое изменение гарантирует, что стек не будет переполнен, если алгоритм быстрой сортировки будет работать на "наихудшем" списке элементов.

Таким образом, нам удалось избавиться от рекурсии, но, как ни странно, экономия времени оказалась незначительной. Более того, в некоторых случаях последний алгоритм работает даже медленнее стандартного (можно предположить, что снижение скорости вызвано определением меньшего списка из двух). Известны и другие улучшения, но они также не дают значительного выигрыша в скорости.

Может быть, у некоторых читателей после изучения кода, приведенного в листинге 5.16, возникла идея написания кода, который бы выполнялся в случае, когда в подсписке находится менее трех элементов. Это и будет нашей следующей областью внесения изменений в алгоритм быстрой сортировки.

Следуя тому же ходу мыслей, что и для сортировки слиянием, можно сказать, что быстрая сортировка будет пытаться сортировать все меньшие и меньшие подсписки, которые эффективнее было бы обрабатывать с помощью других методов.

Представьте себе, что разбиваются только подсписки размером не менее определенного количества элементов. К чему бы привел такой алгоритм быстрой сортировки? Мы получим грубо отсортированный список, т.е. все его элементы будут находиться вблизи требуемых позиций. Подсписки, которые были получены перед прекращением процесса разбиения, будут отсортированы в том смысле, что если подсписок *X* находится перед подписанием *Y*, то все элементы подсписка *X* будут расположены в отсортированном списке перед элементами подсписка *Y*. Это как раз самое удобное распределение для сортировки методом вставок. Таким образом, работу, начатую быстрой сортировкой, можно завершить с помощью сортировки методом вставок.

Это будет последнее улучшение быстрой сортировки, которое мы рассмотрим. Мы реализовали сверхоптимизированную сортировку без рекурсии, с использованием выбора базовой точки по медиане трех и сортировки методом вставок с целью завершения сортировки.

Листинг 5.18. Оптимизированная быстрая сортировка

```
const
  QSCutOff = 15;
procedure QSort(aList : TList;
                aFirst : integer;
                aLast  : integer;
                aCompare : TtdCompareFunc);
var
  i, j : integer;
  IndexOfMin : integer;
  Temp : pointer;
begin
  { найти элемент с наименьшим значением из первых QSCutOff элементов и перемес-
    тить его на первую позицию }
  IndexOfMin := aFirst;
  j := QSCutOff;
```

```

if (j > aLast) then
  j := aLast;
for i := succ(aFirst) to j do
  if (aCompare(aList.List^[i], aList.List^[IndexOfMin]) < 0) then
    IndexOfMin := i;
if (aFirst <> IndexOfMin) then begin
  Temp := aList.List^[aFirst];
  aList.List^[aFirst] := aList.List^[IndexOfMin];
  aList.List^[IndexOfMin] := Temp;
end;
{выполнить сортировку методом вставок}
for i := aFirst+2 to aLast do begin
  Temp := aList.List^[i];
  j := i;
  while (aCompare(Temp, aList.List^[j-1]) < 0) do begin
    aList.List^[j] := aList.List^[j-1];
    dec(j);
  end;
  aList.List^[j] := Temp;
end;
end;

procedure QS(aList : TList;
              aFirst : integer;
              aLast : integer;
              aCompare : TtdCompareFunc);

var
  L, R : integer;
  Pivot : pointer;
  Temp : pointer;
  Stack : array [0..63] of integer; {позволяет разместить до
                                     2 миллиардов элементов}

  SP : integer;
begin
  {инициализировать стек}
  Stack[0] := aFirst;
  Stack[1] := aLast;
  SP := 2;
  {пока в стеке есть подфайлы}
  while (SP <> 0) do begin
    {вытолкнуть верхний подфайл}
    dec(SP, 2);
    aFirst := Stack[SP];
    aLast := Stack[SP+1];
    {повторять пока в подфайле есть достаточное количество элементов}
    while ((aLast - aFirst) > QSCutOff) do begin
      {выполнить сортировку первого, среднего и последнего элементов и в каче-
       стве базовой точки выбрать средний – метод медианы трех}
      R := (aFirst + aLast) div 2;

```

```
if aCompare(aList.List^[aFirst], aList.List^[R]) > 0 then begin
    Temp := aList.List^[aFirst];
    aList.List^[aFirst] := aList.List^[R];
    aList.List^[R] := Temp;
end;
if aCompare(aList.List^[aFirst], aList.List^[aLast]) > 0 then begin
    Temp := aList.List^[aFirst];
    aList.List^[aFirst] := aList.List^[aLast];
    aList.List^[aLast] := Temp;
end;
if aCompare(aList.List^[R], aList.List^[aLast]) > 0 then begin
    Temp := aList.List^[R];
    aList.List^[R] := aList.List^[aLast];
    aList.List^[aLast] := Temp;
end;
Pivot := aList.List^[R];
{задать начальные значения индексов и приступить к разбиению списка}
L := aFirst;
R := aLast;
while true do begin
    repeat dec(R); until (aCompare(aList.List^[R], Pivot) <= 0);
    repeat inc(L); until (aCompare(aList.List^[L], Pivot) >= 0);
    if (L >= R) then Break;
    Temp := aList.List^[L];
    aList.List^[L] := aList.List^[R];
    aList.List^[R] := Temp;
end;
{затолкнуть больший подфайл в стек и повторить цикл для меньшего подфайла}
if (R - aFirst) < (aLast - R) then begin
    Stack[SP] := succ(R);
    Stack[SP+1] := aLast;
    inc(SP, 2);
    aLast := R;
end
else begin
    Stack[SP] := aFirst;
    Stack[SP+1] := R;
    inc(SP, 2);
    aFirst := succ(R);
end;
end;
end;
end;
```

```
procedure TDQuickSort( aList : TList;  
                       aFirst : integer;  
                       aLast  : integer;  
                       aCompare : TtdCompareFunc);  
begin  
    TDValidateListRange(aList, aFirst, aLast, 'TDQuickSort');  
    QS(aList, aFirst, aLast, aCompare);  
    QSInsertionSort(aList, aFirst, aLast, aCompare);  
end;
```

Эта оптимизированная быстрая сортировка состоит из трех процедур. Первая из них – вызываемая процедура **TDQuickSort**. Она проверяет корректность переданных параметров, для частично сортировки списка вызывает процедуру **QS**, а затем для окончательной сортировки вызывает процедуру **QSInsertionSort**. Процедура **QS** выполняет нерекурсивный процесс разбиения списка до получения подсписков определенного минимального размера. **QSInsertionSort** представляет собой процедуру оптимизированной сортировки методом вставок для частично отсортированного списка. В частности, обратите внимание, что элемент с наименьшим значением находится в первых **QSCutOff** элементах списка. Это вызвано выполнением процесса разбиения и тем фактом, что при достижении размеров подсписков **QSCutOff** элементов разбиение прекращается.

Стоила ли игра свеч? Тесты однозначно показывают, что стоила. При сортировке 100000 элементов типа **longint** оптимизированный алгоритм сортировки потребовал на 18% меньше времени, чем стандартный.

Сортировка слиянием для связанных списков

Последним алгоритмом, который мы рассмотрим в этой главе, снова будет сортировка слиянием, но в этот раз применительно к связным спискам. Как вы, наверное, помните, несмотря на высокие показатели быстродействия (алгоритм класса $O(n \log(n))$), использование сортировки слиянием требует наличия вспомогательного массива, размер которого составляет половину размера сортируемого массива. Такая необходимость вызвана тем, что на этапе слияния сортировке нужно куда-то помещать элементы.

Для связанных списков сортировка слиянием *не требует* наличия вспомогательного массива, поскольку элементы можно свободно перемещать, разрывая и восстанавливая связи, с быстродействием $O(1)$, т.е. за постоянное время.

Код для сортировки связанных списков можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDLnkLst.pas**.

Давайте рассмотрим, каким образом работает код для односвязных списков, а затем расширим концепцию для двухсвязных списков.

Предположим, что имеется связный список с фиктивным начальным узлом. (С учетом этого предположения алгоритм сортировки намного упрощается.) Таким

образом, каждый сортируемый нами узел будет иметь родительский узел. Рассмотрим процесс слияния. Пусть имеются два списка, описываемых родительскими узлами первых узлов. Будем считать, что оба списка отсортированы. Можно легко разработать алгоритм слияния с целью объединения двух списков в один. При этом процесс слияния будет заключаться в выполнении удалений и вставок.

Сравниваем два элемента, на которые указывают два родительских узла. Если меньший элемент находится в первом узле, он находится на своем месте, поэтому переходим к следующему узлу. При этом первый узел будет новым родительским узлом. Если же меньший элемент находится во втором списке, его необходимо удалить из списка и вставить после родительского узла первого списка, а затем перейти к следующему узлу. При этом вновь вставленный узел будет новым родительским узлом. Далее описанный процесс продолжается вплоть до исчерпания узлов одного из списков. Если пройден весь первый список, в него добавляются оставшиеся элементы из второго.

Все кажется простым. Тем не менее, может показаться, что в процессе сортировки нам приходится разделять исходный список на большое количество списков, содержащих всего один реальный и один фиктивный узел, а затем объединять их в один список. К счастью, это не так, поскольку в качестве фиктивных начальных узлов можно временно использовать другие узлы из списка и даже не разбивать исходный список на подсписки. Давайте рассмотрим, как это сделать.

Во-первых, потребуется написать метод-драйвер сортировки слиянием. Он будет просто вызывать рекурсивный метод, который и будет заниматься собственно сортировкой. Методу-драйверу будут передаваться два параметра: узел, с которого начинается сортируемый список, и количество элементов в списке. Мы не будем использовать `nil` в качестве сигнализатора окончания списка – для этого будет применяться счетчик узлов. Реализация простого метода-драйвера приведена в листинге 5.19.

Листинг 5.19. Метод-драйвер для сортировки слиянием односвязных списков

```
procedure TtdSingleLinkedList.Sort(aCompare : TtdCompareFunc);  
begin  
  {если в списке более одного элемента, выполнить сортировку слиянием}  
  if (Count > 1) then  
    sllMergesort(aCompare, FHead, Count);  
  MoveBeforeFirst;  
  FIsSorted := true;  
end;
```

Как видите, для выполнения сортировки метод-драйвер вызывает функцию `sllMergeSort`. Эта функция сначала вызывает сама себя для первой, а затем – для второй половины списка, после чего обе половины объединяются в один список. Для обеспечения слияния функция `sllMergeSort` возвращает последний отсортированный узел.

Листинг 5.20. Рекурсивная сортировка слиянием для односвязных списков

```
function TtdSingleLinkList.sllMergesort(aCompare : TtdCompareFunc;  
                                       aPriorNode : PslNode;  
                                       aCount : longint) : PslNode;  
  
var  
    Count2 : longint;  
    PriorNode2 : PslNode;  
  
begin  
    {сначала обрабатывается простой случай: если в списке всего один элемент, он  
    отсортирован, поэтому выполнение функции завершается}  
    if (aCount = 1) then begin  
        Result := aPriorNode^.slnNext;  
        Exit;  
    end;  
    {разбить список на две части}  
    Count2 := aCount div 2;  
    aCount := aCount - Count2;  
    {выполнить сортировку слиянием первой половины:  
    вернуть начальный узел для второй половины}  
    PriorNode2 := sllMergeSort(aCompare, aPriorNode, aCount);  
    {выполнить сортировку слиянием второй половины}  
    sllMergeSort(aCompare, PriorNode2, Count2);  
    {объединить две половины}  
    Result := sllMerge(aCompare, aPriorNode, aCount, PriorNode2, Count2);  
end;
```

Метод сортировки слиянием вызывается с указанием начального узла сортируемого списка и количества узлов в списке. Имея такие входные данные, за счет прохождения списка и подсчета узлов можно определить, где начинается вторая половина списка. В качестве возвращаемого параметра после сортировки первой половины списка используется последний узел первой половины, который служит фиктивным начальным узлом для второй половины. В любом случае нам придется проходить список. Тогда почему бы нам заодно не определить положение средней точки?

И последняя часть реализации сортировки – сама функция слияния. Ее код приведен в листинге 5.21. Она не представляет никаких трудностей для понимания. Начальным узлом объединенного списка будет служить родительский узел первого подсписка. Функция возвращает последний элемент объединенного списка (он будет использоваться в качестве родительского узла для несортированной части подсписка).

Листинг 5.21. Фаза слияния при сортировке слиянием односвязного списка

```
function TtdSingleLinkList.sllMerge( aCompare : TtdCompareFunc;  
    aPriorNode1 : PslNode; aCount1 : longint;  
    aPriorNode2 : PslNode; aCount2 : longint) : PslNode;  
var  
    i : integer;  
    Node1 : PslNode;  
    Node2 : PslNode;  
    LastNode : PslNode;  
    Temp : PslNode;  
begin  
    LastNode := aPriorNode1;  
    {извлечь первые два узла}  
    Node1 := aPriorNode1^.slnNext;  
    Node2 := aPriorNode2^.slnNext;  
    {повторять цикл до исчерпания элементов одного из списков}  
    while (aCount1 > 0) and (aCount2 > 0) do begin  
        if (aCompare(Node1^.slnData, Node2^.slnData) <= 0) then begin  
            LastNode := Node1;  
            Node1 := Node1^.slnNext;  
            dec(aCount1);  
        end  
        else begin  
            Temp := Node2^.slnNext;  
            Node2^.slnNext := Node1;  
            LastNode^.slnNext := Node2;  
            LastNode := Node2;  
            Node2 := Temp;  
            dec(aCount2);  
        end;  
    end;  
    {если закончились элементы в первом списке, связать последний узел с  
    оставшейся частью второго списка и пройти список до последнего узла}  
    if (aCount1 = 0) then begin  
        LastNode^.slnNext := Node2;  
        for i := 0 to pred(aCount2) do  
            LastNode := LastNode^.slnNext;  
    end  
    {если закончились элементы во втором списке, то Node2 будет первым узлом в  
    оставшемся списке; пройти список до последнего узла и связать его с узлом  
    Node2}  
    else begin  
        for i := 0 to pred(aCount1) do  
            LastNode := LastNode^.slnNext;  
        LastNode^.slnNext := Node2;  
    end;  
    {вернуть последний узел}  
    Result := LastNode;  
end;
```

Обратите внимание, что в односвязном списке сортировка слиянием не требует выполнения обратного прохода. Мы не были в ситуации, когда требовалось знание родительского узла определенного узла, а он не был известен. Это означает, что сортировка слиянием в двухсвязном списке может выполняться точно так же, как и в односвязном, но после сортировки нужно будет пройти весь список и восстановить обратные ссылки.

Листинг 5.22. Сортировка слиянием для двухсвязного списка

```
function TtdDoubleLinkedList.dllMerge(aCompare : TtdCompareFunc;
    aPriorNode1: PdlNode; aCount1 : longint;
    aPriorNode2: PdlNode; aCount2 : longint) : PdlNode;
var
    i : integer;
    Node1 : PdlNode;
    Node2 : PdlNode;
    LastNode : PdlNode;
    Temp : PdlNode;
begin
    LastNode := aPriorNode1;
    {извлечь первые два узла}
    Node1 := aPriorNode1^.dlnNext;
    Node2 := aPriorNode2^.dlnNext;
    {повторять до тех пор, пока один из списков не опустеет}
    while (aCount1 <> 0) and (aCount2 <> 0) do begin
        if (aCompare(Node1^.dlnData, Node2^.dlnData) <= 0) then begin
            LastNode := Node1;
            Node1 := Node1^.dlnNext;
            dec(aCount1);
        end
        else begin
            Temp := Node2^.dlnNext;
            Node2^.dlnNext := Node1;
            LastNode^.dlnNext := Node2;
            LastNode := Node2;
            Node2 := Temp;
            dec(aCount2);
        end;
    end;
    {если закончились элементы в первом списке, связать последний узел с
    оставшейся частью второго списка и пройти список до последнего узла}
    if (aCount1 = 0) then begin
        LastNode^.dlnNext := Node2;
        for i := 0 to pred(aCount2) do
            LastNode := LastNode^.dlnNext;
    end
end;
```

```

{если закончились элементы во втором списке, то Node2 будет первым узлом в
оставшемся списке; пройти список до последнего узла и связать его с узлом
Node2}
else begin
  for i := 0 to pred(aCount1) do
    LastNode := LastNode^.dlnNext;
    LastNode^.dlnNext := Node2;
  end;
  {вернуть последний узел}
  Result := LastNode;
end;

function TtdDoubleLinkedList.dllMergesort(aCompare : TtdCompareFunc;
      aPriorNode : PdlNode; aCount : longint) : PdlNode;
var
  Count2 : longint;
  PriorNode2 : PdlNode;
begin
  {сначала обрабатывается простой случай: если в списке всего один элемент, он
отсортирован, поэтому выполнение функции завершается}
  if (aCount = 1) then begin
    Result := aPriorNode^.dlnNext;
    Exit;
  end;
  {разбить список на две части}
  Count2 := aCount div 2;
  aCount := aCount - Count2;
  {выполнить сортировку слиянием первой половины: вернуть начальный узел для
второй половины}
  PriorNode2 := dllMergeSort(aCompare, aPriorNode, aCount);
  {выполнить сортировку слиянием второй половины}
  dllMergeSort(aCompare, PriorNode2, Count2);
  {объединить две половины}
  Result := dllMerge(aCompare, aPriorNode, aCount, PriorNode2, Count2);
end;

procedure TtdDoubleLinkedList.Sort(aCompare : TtdCompareFunc);
var
  Dad, Walker : PdlNode;
begin
  {если в списке больше одного элемента, выполнить сортировку для односвязного
списка, а затем восстановить обратные ссылки}
  if (Count > 1) then begin
    dllMergesort(aCompare, FHead, Count);
    Dad := FHead;
    Walker := FHead^.dlnNext;
  end;
end;

```

```
while (Walker<>nil) do begin
  Walker^.dlnPrior := Dad;
  Dad := Walker;
  Walker := Dad^.dlnNext;
end;
end;
MoveBeforeFirst;
FIsSorted := true;
end;
```

Резюме

В этой главе мы рассмотрели различные алгоритмы сортировки и изучили особенности и характеристики каждого из них. Были описаны базовые алгоритмы: пузырьковая сортировка, шейкер-сортировка и сортировка методом вставок, и было показано, что они принадлежат к классу $O(n^2)$. Затем были описаны два алгоритма со средним быстродействием: сортировка методом Шелла и сортировка прочесыванием. Их анализ был сложнее, чем для алгоритмов первой группы, но они были быстрее базовых алгоритмов. И, наконец, были рассмотрены два самых быстрых метода сортировки: сортировка слиянием и быстрая сортировка, которые принадлежат к классу $O(n \log(n))$. Было показано, что в отличие от всех других методов, сортировка слиянием требует организации вспомогательного массива.

Для быстрой сортировки мы рассмотрели целый ряд возможных улучшений, подробно описывая каждое из них и оценивая его влияние на время выполнения алгоритма. Улучшения не оказывали влияния на функцию быстродействия алгоритма в контексте O -нотации, но, тем не менее, приводили к снижению константы пропорциональности, тем самым увеличивая скорость работы алгоритма.

И, наконец, было показано, каким образом сортировка слиянием применяется в отношении связанных списков. В этом случае она не требует наличия вспомогательного массива и позволяет достичь максимальной эффективности.



Глава 6

Рандомизированные алгоритмы

Возможно, у кого-то из вас, кто просто листал эту книгу и случайно наткнулся на данную главу, возник вопрос, что же такое *рандомизированные алгоритмы*? Это алгоритмы, работающие случайным образом? Ничего подобного. Здесь термин *рандомизированный алгоритм* (*randomized algorithm*) употребляется в отношении алгоритма, который генерирует или использует случайные числа.

Если вы на минутку отвлекетесь и подумаете над выражением "генерация случайных чисел", то, скорее всего, придете к выводу, что оно не имеет смысла. Компьютеры – это детерминированные машины: если существует определенная программа или функция, предназначенная для выполнения определенной работы, то для одного и того же набора входных данных она будет давать один и тот же набор выходных данных. (Если это не так, компьютер можно преспокойно отправлять в ремонт.) Без использования специального оборудования для генерации случайных чисел программные генераторы также представляют собой всего-навсего функции. Каким же образом вычисляемые ими числа могут быть случайными? Если запустить генератор в некотором определенном состоянии, то, изучив исходный код генератора, можно предсказать всю последовательность генерируемых им случайных чисел. Какие же это случайные числа? Скоро мы более подробно обсудим эту дилемму.

В состав ядра операционной системы Linux входит модуль, который анализирует, каким образом пользователь вводит данные с клавиатуры и оценивает интервал между нажатиями клавиш, а затем использует полученные данные для вычисления рандомизирующего коэффициента. Подобным образом генераторы случайных чисел, имеющиеся в ядре, дают более "случайные" последовательности значений.

С применением случайных чисел в алгоритмах мы уже встречались в главе 5: алгоритм быстрой сортировки со случайным выбором базового элемента. Причина, по которой в алгоритме сортировки использовались случайные числа, состояла в том, что этот алгоритм, несмотря на его высокие общие характеристики, обладает очень низкими характеристиками в худшем случае. За счет применения случайных чисел можно значительно снизить вероятность попадания на сценарий худшего случая. В этой главе мы рассмотрим новую структуру данных – списки с пропусками, которые представляют собой метод организации отсортированных связных списков с помощью случайных чисел, что существенно увеличивает скорость выполнения операции вставки нового элемента.

Есть и другие алгоритмы, использующие случайные числа. Необходимость применения случайных чисел бывает вызвана, как правило, двумя причинами: 1) пространство решений алгоритма очень велико и поиск определенного решения будет выполняться слишком медленно, 2) существует необходимость моделирования физической системы для ее оптимизации и т.п.

Все исходные коды для генераторов случайных чисел можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDRandom.pas**.

Генерация случайных чисел

Прежде всего, давайте опишем, что мы понимаем под *случайным числом* (*random number*). Без четкого определения термина мы будем неуверенно себя чувствовать при разработке и реализации генератора случайных чисел.

Будет ли число 2 случайным числом? Просто так, не привязываясь к контексту, в котором используется это число, нельзя сказать ни да, ни нет. Если один раз бросить игральный кубик, мы можем получить число 2, но оно ни о чем нам не говорит: может, это была просто удача, а, может, на всех гранях кубика были двойки, или центр тяжести кубика был смещен таким образом, что всегда выпадала только двойка. Чтобы определить, является ли число 2 случайным, нужно изучить *последовательность* выходных данных генератора, в которой встречается число 2. Только так можно оценить, было ли определенное число случайным.

Хорошо. А что можно определить из последовательности чисел 1, 2, 3, 4? Числа не выглядят случайными, не так ли? Если бы у нас в распоряжении был генератор случайных чисел на основе квантового источника данных (т.е. источника, который генерирует действительно случайные события), вероятность получения приведенной последовательности, как и любой другой последовательности из четырех чисел, была бы 1:10000, т.е. исходя из теории вероятностей, последовательность повторялась бы один раз из 10000 попыток. Но в данном случае наша интуиция не помогает. Чтобы определить, является ли полученная последовательность, а, следовательно, и сам генератор, случайной, необходимо провести определенные тесты и призвать на помощь теорию вероятностей или математическую статистику.

На основе вышесказанного можно вывести определение генератора случайных чисел. Генератор случайных чисел – это программа, дающая на своем выходе последовательность чисел, которая может успешно пройти статистические или вероятностные тесты на случайность. Строго говоря, программы и функции, генерирующие случайные числа, называют *генераторами псевдослучайных чисел* (*pseudorandom number generators*), чтобы отличать их от генераторов действительно случайных чисел, которые основаны на определенного рода случайных событиях, происходящих на квантовом уровне. (Современные теории утверждают, что квантовые события происходят случайным образом. Время распада радиоактивного атома нельзя предсказать; можно говорить только об определенном периоде времени, который можно оценить путем наблюдения за распадом большого количества атомов.)

Какие тесты необходимо выполнить над последовательностью чисел, чтобы определить, случайны они или нет? Все тесты такого рода будут статистическими по своей природе; за счет наблюдения за большим количеством событий можно сделать вывод о наличии или отсутствии в данных статистических комбинаций. Один из самых простых тестов заключается в группировке чисел из последовательности. Пусть, например, имеется последовательность однозначных чисел, которую требуется проверить на случайность. Разбиваем последовательность на категории, вычисляя количество нулей, единиц, двоек и т.д. Для случайной последовательности ожидаемое количество появлений каждого числа будет примерно равно одной десятой общего количества чисел в последовательности. Так, в последовательности из 1000 случайных однозначных чисел будет содержаться примерно 100 нулей, 100 единиц, 100 двоек и т.д. до девяток. Конечно, количество вхождений каждого числа не будет в точности равно 100, но будет достаточно близко от ожидаемого значения.

"Ожидать", "примерно", "около". Такие слова не вселяют уверенности в том, что используемые нами тесты дают объективные, а не субъективные результаты. В конце концов, если при подсчете нулей было получено, например, значение 110, для одного человека это может быть вполне приемлемым, но для другого совершенно неприемлемым.

Критерий хи-квадрат

Представьте себе, что есть две монеты, над которыми поработал мошенник. Каким образом можно доказать, что монеты имеют смещенный центр тяжести? Конечно, наш предполагаемый мошенник мог быть достаточно глупым и просто сбалансировать монеты таким образом, чтобы они всегда падали решкой вверх. Но такой мошенник был бы давным-давно пойман, а более изобретательный мошенник вполне мог бы остаться на свободе. Давайте бросим две монеты, скажем, 100 раз, и внесем полученные данные в таблицу. Полученная таблица может выглядеть следующим образом (см. табл. 6.1):

Таблица 6.1. Результаты бросания 100 раз двух монет со смещенным центром тяжести

	Две решки (1)	Одна решка и один орел (2)	Два орла (3)
Наши броски (100 раз)	28	51	21
Вероятность события	1/4	1/2	1/4
Ожидаемое количество для 100 бросков	25	50	25

В таблице 6.1 для каждого возможного события приведена вероятность его возникновения и, кроме того, указано ожидаемое количество появлений каждого из событий для 100 бросков. (Ожидаемое количество появлений событий представляет собой просто результат умножения вероятности на общее количество событий.)

Одного взгляда достаточно, чтобы сказать, что две решки выпадают чаще, чем этого следует ожидать, однако достаточно ли велико отклонение, чтобы можно было сказать, что монеты имеют смещенный центр тяжести? Давайте посмотрим на разброс (т.е. отличие) полученных и ожидаемых результатов. Чтобы выделить разности и избавиться от отрицательных значений, возведем их в квадрат. Сумма полученных квадратов разностей и будет служить оценкой случайности результатов проведенных тестов. В нашем случае вычисление суммы квадратов разностей дает 26 ($= 3^2 + 1^2 + (-4)^2$). Но подождите-ка минутку, нам нужно каким-то образом учесть вероятность возникновения каждого события. Так для события "орел и решка" квадрат разности должен быть больше, чем для события "две решки", хотя бы только потому, что первое событие должно происходить чаще. Другими словами, разница 3 для события "две решки" будет намного более значительна, чем разница 1 для события "орел и решка". Поэтому давайте разделим каждый квадрат разности на ожидаемое количество появлений соответствующего события. Новая сумма будет вычисляться следующим образом:

$$X = \sum_1^3 \frac{(C_i - 100p_i)^2}{100p_i}$$

где C_i – наблюдаемое количество, а p_i – вероятность возникновения события i . Для наших данных значение X будет равно 1.02. Полученная нами сумма известна под названием *критерия хи-квадрат (chi-squared value)*. Полученное значение можно найти в таблице стандартного распределения хи-квадрат (см. табл. 6.2).

Таблица 6.2. Процентные точки распределения хи-квадрат

	1%	5%	95%	99%
$v = 1$	0.000157	0.00393	3.84	6.63
$v = 2$	0.0201	0.103	5.99	9.21
$v = 3$	0.115	0.352	7.81	11.3
$v = 4$	0.297	0.711	9.49	13.3
$v = 5$	0.554	1.15	11.1	15.1
$v = 6$	0.872	1.64	12.6	16.8

Вид таблицы слегка пугает, но понять ее совсем не сложно. Значения, приведенные в таблице, представляют собой значения распределения хи-квадрат для ν степеней свободы (греческая буква ν – это стандартный символ для обозначения степеней свободы). В свободной интерпретации можно сказать, что значение степеней свободы на единицу меньше количества возможных типов событий. В нашем случае возможны три типа событий: "две решки", "орел и решка" и "два орла". Следовательно, для нашего эксперимента количество степеней свободы будет равно 2. Строка для $\nu = 2$ содержит четыре значения – по одному значению в каждом из четырех столбцов. Значение в столбце 1% (0.0201) можно интерпретировать следующим образом: "Значение критерия X должно быть меньше 0.0201 только 1% времени". Другими словами, при повторении эксперимента 100 раз только примерно в одном из них будет получено значение X , меньшее 0.0201. Если будет обнаружено, что во многих экспериментах будет получено значение меньше 0.0201, можно будет с достаточно высокой степенью уверенности сказать, что бросание монет не является случайным событием, т.е. монеты имеют смещенный центр тяжести. То же самое можно сказать и для столбца 5%. О столбце 95% можно сказать, что значение параметра X должно быть меньше 5.99 примерно 95% времени или, что эквивалентно, значение параметра X должно быть больше 5.99 примерно 5% времени. Аналогичные рассуждения справедливы и для столбца 99%.

Полученное нами значение параметра X попадает в диапазон от 5% до 95%, т.е. на его основе мы не можем прийти к четкому заключению о смещенном центре тяжести монет. Приходится предполагать, что монеты являются настоящими (без всяких "хитростей"). Если же, с другой стороны, значение X было равно 10, можно было бы сказать, что такая ситуация может складываться не более чем в 1% экспериментов (10 больше чем 9.21 – значения для столбца 99%). Это послужило бы веским доказательством того, что монеты имеют смещенный центр тяжести. Конечно, потребуется провести большее количество экспериментов, и посмотреть, каким образом получаемые данные соотносятся со стандартным распределением хи-квадрат. По такому расширенному набору данных можно будет более точно оценить случайность получаемых данных. Не хотелось бы делать выводы, основываясь на результатах, которые согласно теории вероятностей, хотя и редко, но все же могут быть получены.

Как правило, при оценке случайного характера получаемых результатов берется одна и та же граница с каждого конца распределения хи-квадрат, скажем, 5% и 95%, и утверждается, что эксперимент является *достоверным* на уровне 5%, если данные эксперимента не попадают в эти границы, и *недостоверным* на уровне 5% – в противном случае.

До сих пор мы не упоминали еще один аспект: какое количество отдельных событий нужно генерировать? В нашем примере с монетами их было 100. Достаточно ли такого количества? Или можно обойтись и меньшим объемом экспериментов? Или же количество событий должно быть больше? К сожалению, четкого

ответа на поставленные вопросы не существует. Кнут (Knuth) утверждает, что хорошим практическим методом для определения достаточности объема экспериментов является следующее: количество ожидаемых событий каждого типа должно быть не менее пяти (в нашем случае ожидаемыми значениями являются 25, 50 и 25, следовательно, объем нашего эксперимента вполне достаточен для оценки случайности результатов), но чем больше событий каждого типа, тем лучше [11].

Давайте оставим наши монеты в покое и вернемся к гипотетической последовательности случайных чисел. Воспользуемся всеми только что полученными знаниями. Определим количество вхождений каждого числа, вычислим значение параметра X и посмотрим, как оно соответствует распределению хи-квадрат с девятью степенями свободы (для последовательности однозначных чисел возможно выпадение одного из 10 чисел; таким образом, количество степеней свободы будет на единицу меньше, т.е. 9). Минимальный объем экспериментов должен составлять, по крайней мере, 50 чисел (чтобы количество разных чисел было не менее 5), хотя чем длиннее последовательность, тем лучше.

Можно пойти даже дальше. Если рассматривать последовательность как серию пар чисел от 00 до 99, считая каждую пару отдельным событием, ее можно будет разбить на 100 типов событий. Следовательно, количество степеней свободы будет равно 99. Вероятность выпадения каждой пары составляет 1:100. Таким образом, для обеспечения возможности оценки случайности последовательности она должна содержать не менее 500 пар (1000 чисел).

Более того, можно использовать не пары чисел, а тройки, но в этом случае понадобится проводить еще больший объем экспериментов. Существуют и другие виды тестов, но перед их рассмотрением давайте выясним, как можно генерировать случайные числа. После изучения нескольких генераторов последовательностей случайных чисел можно будет прогнать тесты на результатах их работы.

Еще раз хотелось бы повторить, что детерминированные алгоритмы не могут генерировать последовательности случайных чисел, аналогичные получаемым при бросках игрального кубика или при подсчете количества бета-частиц во время распада радиоактивного материала. Детерминированные алгоритмы на основе одинаковых исходных данных будут генерировать одни и те же последовательности чисел. Если, например, генератор X , основанный на четко определенном алгоритме, для *начального числа* 12 345 678 генерирует случайное число 65 584 256, то даже через пять месяцев тот же генератор X при том же начальном числе даст значение 65 584 256. Следовательно, в *вычислении* последовательности случайных чисел нет случайности, но с помощью статистических тестов можно показать, что *последовательность* чисел, генерируемая подобным образом, содержит случайные числа.

Более того, в некоторых случаях повторяемость последовательности случайных чисел бывает даже желательна. Она позволяет использовать генератор для многократного воспроизведения одной и той же последовательности. Такая возможность бывает необходимой в процессе отладки с целью воспроизведения ошибки.

Метод средних квадратов

История генераторов случайных чисел уходит корнями к одному из самых известных имен в теории вычислительных машин – Джону фон Нейману (John von Neumann). В 1946 году он предложил следующую схему генерации последовательностей случайных чисел: возьмите N -значное число, возведите его в квадрат и из результата, выраженного в виде $2N$ -значного числа (при необходимости дополненного слева до $2N$ -значного), возьмите средние N цифр. Это и будет следующее число в последовательности. Так, например, если N равно 4, в качестве начального числа можно взять 1234. Следующими числами в последовательности будут 5227, 3215, 3362, 3030, 1809 и т.д. Описанный метод известен под названием *метода средних квадратов* (*middle-square method*).

Листинг 6.1. Метод средних квадратов в действии

```
var
  MidSqSeed : integer;
function GetMidSquareNumber : integer;
var
  Seed : longint;
begin
  Seed := longint(MidSqSeed) * MidSqSeed;
  MidSqSeed := (Seed div 100) mod 10000;
  Result := MidSqSeed;
end;
```

К сожалению, с приведенным алгоритмом связано несколько *больших* проблем, которые исключают его применение в практических целях. Вернемся к нашему примеру с четырехзначными случайными числами. Предположим, что в последовательности нам встретилось число меньше 10. При вычислении квадрата будет получено число меньше 100. Это, в свою очередь, означает, что следующим числом в последовательности будет 0 (поскольку мы возьмем четыре средние цифры из числа 000000xx). Это число также меньше 10, следовательно, все последующие числа в последовательности будут равны 0. Вряд ли кто-то может сказать, что такая последовательность будет случайной! (Если в качестве начального взять число 1234, то до попадания в 0 последовательность будет содержать 55 чисел.) Кроме того, если начать, например, с числа 4100, последовательность будет состоять из 8100, 6100, 2100, 4100 и так до бесконечности. Существуют и другие патологические последовательности, на которые очень легко натолкнуться и очень трудно избежать.

Метод средних квадратов позволяет легко генерировать случайные числа на основе 16-битного целого числа. Возведение 16-битного числа в квадрат дает 32-битное число. Затем для вычисления средних 16-бит нужно всего лишь сдвинуть полученный результат на 8 бит вправо и выполнить операцию **AND** с числом **\$FFFF**. Тем не менее, даже в этом случае алгоритм средних квадратов будет да-

вать бесполезные результаты. После 50–60 случайных чисел алгоритм приводит к генерации нулей или попадает в цикл. То же самое происходит и для 32-битных чисел. В общем случае, несмотря простоту, применение метода средних квадратов вследствие его недостатков предельно ограничено.

Линейный конгруэнтный метод

Следующий большой шаг в разработке генераторов случайных чисел был сделан Д. Лемером (D.H. Lehmer) в 1949 году. Предложенный им генератор носит название *линейного конгруэнтного метода* (*linear congruential method*). Выберите три числа m , a и c и начальное число X_0 . Для генерации последовательности случайных чисел используется следующая формула:

$$X_{n+1} = (aX_n + c) \bmod m$$

Операция взятия по модулю m ($\bmod m$) представляет собой вычисление остатка от деления числа на m , например, $24 \bmod 10 = 4$.

При удачном выборе начальных чисел генерируемая последовательность будет содержать случайные числа. Например, стандартный генератор случайных чисел в Delphi использует значения $a = 134775813$ (\$8088405), $c = 1$ и $m = 2^{32}$, а значение X_0 выбирается самим пользователем. (Значение начального числа содержится в глобальной переменной **RandSeed**. Его можно задавать напрямую или использовать процедуру **Randomize** для вычисления его на основе показаний системных часов.)

Следует отметить, что если в двух разных точках последовательности получено одно и то же значение x , то последовательность в этих двух точках должна полностью повторяться, поскольку алгоритм детерминированный. Так как в формуле используется операция определения остатка от деления, все значения в последовательности будут меньше m , т.е. будут находиться в диапазоне от 0 до $m - 1$. Следовательно, последовательность будет повторяться после не более чем m чисел. При неудачном выборе значения a , c и m повторение последовательности может начаться гораздо раньше. В качестве простого примера можно привести случай, когда $a = 0$: вся последовательность сводится к повторению значения параметра c — $\{c, c, c, \dots\}$.

Каким образом можно выбрать удачные значения для a , c и m ? В литературе содержится немало размышлений, описаний и доказательств. Как правило, значение параметра m выбирается как можно больше, чтобы цикл повторяемости был также как можно большим. Нужно выбирать его, как минимум, равным размеру слова операционной системы (другими словами, для 32-разрядных операционных систем m выбирается равным 31 или 32 бита). Значение параметра a выбирается таким образом, чтобы оно было взаимно простым со значением числа m (два числа являются *взаимно простыми*, если их наибольший общий делитель равен 1). Значение c , как правило, берется равным 0 или 1, несмотря на то, что общее правило гласит, что должно выбираться ненулевое значение, взаимно простое со значением параметра m .

В случае если значение s равно 0, генератор называется *мультипликативным линейным конгруэнтным генератором случайных чисел* (*multiplicative linear congruential generator*). Чтобы гарантировать, что цикл повторения последовательности максимален, необходимо в качестве значения параметра m выбирать простое число. Самым известным генератором подобного рода является так называемый *минимальный стандартный генератор случайных чисел* (*minimal standard random number generator*), предложенный Стивеном Парком (Stephen Park) и Кейтом Миллером (Keith Miller) в 1988 году. Для него $a = 16807$, а $m = 2147483647$ (или $2^{31} - 1$). После разработки этого генератора было проведено большое количество статистических тестов, и генератор прошел большинство из них (несмотря на то что предложенный генератор обладает определенными нежелательными свойствами, которые мы рассмотрим чуть ниже).

Мультипликативные линейные конгруэнтные генераторы случайных чисел имеют одну аномалию: они никогда не дают числа 0. (Это объясняется тем, что, во-первых, m представляет собой простое число, во-вторых, $a \bmod m$ не равно нулю, и, в-третьих, если начальное число не равно нулю, $X_0 \bmod m$ тоже не равно нулю.) Следовательно, если генераторы никогда не дают числа 0, их нельзя называть случайными. На практике невозможность генерации нуля, как правило, игнорируется, – в конце концов, в 32-разрядной операционной системе это всего лишь отсутствие всего одного числа из примерно 2 миллиардов.

При реализации минимального стандартного генератора случайных чисел (как, в общем-то, и любого другого) особое внимание необходимо уделить исключению возможности возникновения переполнения, поскольку значение текущего начального числа, умноженное на a , может легко превысить максимально допустимое значение для 32-битного целого числа. Если не позаботиться об исключении переполнения, возможно возникновение ошибок, которые негативно скажутся на достаточно хорошем генераторе случайных чисел. Для обработки случаев переполнения используется метод Шрейга (Schrage) (его описание в этой книге не приводится, но его можно найти в статье Парка и Миллера [16]).

Для сравнения и тестирования различных генераторов случайных чисел будет создана иерархия классов, базовый класс которой будет содержать виртуальный метод, инкапсулирующий основные функциональные возможности генератора, в частности, генерация случайного числа с плавающей запятой в диапазоне от 0 до 1 (мы будем пользоваться переменными типа **double**). Этот виртуальный метод будет перекрываться в дочерних классах, что позволит генерировать случайное число в соответствии с алгоритмами дочерних классов. В базовом классе метод будет применяться для создания других типов случайных чисел, например, случайных чисел целого типа не больше определенного значения или случайного числа из определенного диапазона.

Наличие иерархии классов генераторов случайных чисел дает еще одно преимущество. Поскольку данные для генератора случайных чисел содержатся ис-

ключительно внутри самого объекта, в одном приложении можно будет использовать несколько независимых генераторов. Стандартная функция **Random** имеет одно и только одно начальное значение, которое будет использоваться для всех вызовов функции в приложении. В ситуации, когда несколько различных процедур прибегают к услугам функции **Random**, очень сложно получить воспроизводимые результаты, поскольку отдельные вызовы будут влиять на получаемые случайные значения.

Листинг 6.2. Базовый класс генератора случайных чисел

```

type
  TtdBasePRNG = class
    private
      FName : TtdNameString;
    protected
      procedure bError(aErrorCode : integer;
                     const aMethodName : TtdNameString);
    public
      function AsDouble : double; virtual; abstract;
      {-вернуть случайное число из диапазона от 0 включительно до 1
       исключительно}
      function AsLimitedDouble(aLower, aUpper : double) : double;
      {-вернуть случайное число из диапазона от aLower включительно до aUpper
       исключительно}
      function AsInteger(aUpper : integer) : integer;
      {-вернуть случайное число из диапазона от 0 включительно до aUpper
       исключительно}
      property Name : TtdNameString read FName write FName;
    end;
function TtdBasePRNG.AsLimitedDouble(aLower, aUpper : double) : double;
begin
  if (aLower < 0.0) or (aUpper < 0.0) or (aLower >= aUpper) then
    bError(tdeRandRangeError, 'AsLimitedDouble');
  Result := (AsDouble * (aUpper - aLower)) + aLower;
end;
function TtdBasePRNG.AsInteger(aUpper : integer) : integer;
begin
  if (aUpper <= 0) then
    bError(tdeRandRangeError, 'AsInteger');
  Result := Trunc(AsDouble * aUpper);
end;
procedure TtdBasePRNG.bError(aErrorCode : integer;
                             const aMethodName : TtdNameString);
begin
  raise EtdRandGenException.Create(
    FmtLoadStr(aErrorCode,
               [UnitName, ClassName, aMethodName, Name]));
end;

```

В листинге 6.2 приведен код базового класса генератора случайных чисел. В нем определен виртуальный метод **AsDouble**, который возвращает случайное число x в диапазоне $0 \leq x < 1$. Кроме того, в классе объявлены два простых метода, один из которых возвращает случайное число с плавающей запятой из заданного диапазона значений, а второй – из диапазона значений от 0 до некоторой заданной верхней границы (аналогично тому, как функция **Random(Limit)** использует целое значение **Limit**). Теперь, когда базовый класс определен, для реализации алгоритма Парка и Миллера можно объявить дочерний класс.

Листинг 6.3. Минимальный стандартный генератор псевдослучайных чисел

```

type
  TtdMinStandardPRNG = class(TtdBasePRNG)
  private
    FSeed : longint;
  protected
    procedure msSetSeed(aValue : longint);
  public
    constructor Create(aSeed : longint);
    function AsDouble : double; override;
    property Seed : longint read FSeed write msSetSeed;
  end;

constructor TtdMinStandardPRNG.Create(aSeed : longint);
begin
  inherited Create;
  Seed := aSeed;
end;

function TtdMinStandardPRNG.AsDouble : double;
const
  a = 16807;
  m = 2147483647;
  q = 127773; {равно m div a}
  r = 2836; {равно m mod a}
  OneOverM : double = 1.0 / 2147483647.0;
var
  k : longint;
begin
  k := FSeed div q;
  FSeed := (a * (FSeed - (k * q))) - (k * r);
  if (FSeed <= 0) then
    inc(FSeed, m);
  Result := FSeed * OneOverM;
end;

function GetTimeAsLong : longint;
{$IFDEF Delphi}
assembler;

```

```

asm
    mov ah, $2C
    call DOS3Call
    mov ax, cx
end;
{ $ENDIF }
{ $IFDEF Delphi2Plus }
begin
    Result := longint(GetTickCount);
end;
{ $ENDIF }
{ $IFDEF Kylix1Plus }
var
    T : TTime_t;
begin
    __time(@T);
    Result := longint(T);
end;
{ $ENDIF }
procedure TtdMinStandardPRNG.msSetSeed(aValue : longint);
const
    m = 2147483647;
begin
    if (aValue > 0) then
        FSeed := aValue
    else
        FSeed := GetTimeAsLong;
        { убедиться, что значение начального числа находится в пределах от 0 до m-1
        включительно}
        if (FSeed >= m-1) then
            FSeed := FSeed - (m - 1) + 1;
end;

```

Как несложно заметить в коде метода **AsDouble**, метод Шрейга выглядит гораздо сложнее, нежели простая формула $X_{n+1} = aX_n \bmod m$ со значениями $a = 16807$ и $m = 2^{31} - 1$. Тем не менее, используя достаточно сложные математические выкладки, можно доказать его равенство приведенной формуле.

Кроме того, как уже упоминалось, в генераторе случайных чисел подобного типа использование нуля в качестве начального числа нежелательно, поскольку тогда бы все генерируемые значения были бы нулевыми. Поэтому метод **msSetSeed** использует значение 0 в качестве флага при необходимости установки начального числа по значению системных часов. К сожалению, для выполнения этой операции в 16- и 32-разрядных системах Windows используется разный код.

Создадим класс случайных чисел, который будет использовать системный генератор случайных чисел – функцию **Random**. В листинге 6.4 показан код метода **AsDouble** для такого класса.

Листинг 6.4. Использование в классе системной функции Random

```
function TtdSystemPRNG.AsDouble : double;  
var  
    OldSeed : longint;  
begin  
    OldSeed := System.RandSeed;  
    System.RandSeed := Seed;  
    Result := System.Random;  
    Seed := System.RandSeed;  
    System.RandSeed := OldSeed;  
end;
```

Теперь, когда в нашем арсенале имеется два генератора случайных чисел, можно перейти к обсуждению методов тестирования их результатов.

Тестирование

В основе всех тестов будут лежать одни и те же принципы. Мы будем генерировать большое количество случайных чисел из диапазона от 0.0 (включительно) до 1.0 (исключительно). Получаемые в результате работы генераторов значения будут разбиваться на несколько категорий, будет подсчитываться количество значений в каждой категории, а затем вероятность попадания значения в каждую категорию. На основе результатов вычислений будет определяться значение функции хи-квадрат, на основе которого будет прогоняться тест по критерию хи-квадрат. При этом количество степеней свободы будет на единицу меньше, чем количество категорий значений. Это было всего лишь краткое введение, но через несколько минут мы приступим к собственно тестированию.

Тест на однородность

Первый тест самый простой – проверка на однородность. О нем мы уже говорили. Фактически случайные числа будут проверяться на равномерность распределения по диапазону от 0.0 до 1.0. Разобьем весь диапазон на 100 поддиапазонов, сформируем набор из 1000000 случайных чисел и вычислим количество значений, попавших в каждый поддиапазон. В поддиапазоне 0 будут находиться значения от 0.00 до 0.01, в поддиапазоне 1 – значения от 0.01 до 0.02 и т.д. Вероятность попадания случайного числа в любой поддиапазон составляет 0.01. Для полученного распределения вычислим значение параметра хи-квадрат и сравним его с данными для стандартного распределения хи-квадрат, находящимися в строке, для 99 степеней свободы.

Листинг 6.5. Тест на однородность

```
procedure UniformityTest(RandGen : TtdBasePRNG;  
    var ChiSquare : double;  
    var DegrFreedo : integer);
```

```

var
  BucketNumber,
  i : integer;
  Expected, ChiSqVal : double;
  Bucket : array [0..pred(UniformityIntervals)] of integer;
begin
  {вычислить количество чисел в каждом поддиапазоне}
  FillChar(Bucket, sizeof(Bucket), 0);
  for i := 0 to pred(UniformityCount) do begin
    BucketNumber := trunc(RandGen.AsDouble * UniformityIntervals);
    inc(Bucket[BucketNumber]);
  end;
  {вычислить значение параметра хи-квадрат}
  Expected := UniformityCount / UniformityIntervals;
  ChiSqVal := 0.0;
  for i := 0 to pred(UniformityIntervals) do
    ChiSqVal := ChiSqVal + (Sqr(Expected - Bucket[i]) / Expected);
  {вернуть значения}
  ChiSquare := ChiSqVal;
  DegrFreedom := pred(UniformityIntervals);
end;

```

Тест на пропуски

Второй тест, который мы проведем, – тест на пропуски – несколько сложнее первого. Тест на пропуски гарантирует, что последовательность случайных чисел не будет попадать сначала в один поддиапазон, а затем в другой, третий и т.д., несмотря на то, что в целом значения будут распределены равномерно по всему диапазону. Определим в диапазоне поддиапазон, скажем, первую половину – от 0.0 до 0.5. Сформируем набор случайных чисел. Для каждого генерируемого числа будем проверять, попадает ли оно в выбранный поддиапазон (*попадание*) или нет (*промах*). В результате проверок будет получена последовательность попаданий и промахов. Найдите последовательности из одного и большего количества промахов (такие последовательности называются пропусками, отсюда и название теста – тест на пропуски). Вы получите последовательности из одного, двух и даже большего количества промахов. Разбейте длины пропусков на категории. Если известно, что вероятность попадания равна p (в нашем случае она будет равна длине выбранного поддиапазона), то вероятность промаха будет $(1 - p)$. На основе этих данных можно определить вероятность возникновения пропуска из одного промаха — $(1 - p)p$, двух промахов — $(1 - p)^2p$, n промахов — $(1 - p)^np$, а, следовательно, вычислить ожидаемое количество пропусков любой длины. После этого применим тест по критерию хи-квадрат. Будем использовать 10 категорий пропусков (поскольку вероятность возникновения пропусков длиной 11 и более промахов очень мала, все пропуски длиной 10 и более будут учитываться в последней категории; при этом, конечно, следует учитывать реальную вероятность по-

падания длины пропуска в эту последнюю категорию), следовательно, мы получим девять степеней свободы. Как правило, тест на пропуски проводится пять раз: для первой и второй половины диапазона, а также для первой, второй и третьей третей диапазона.

Листинг 6.6. Тест на пропуски

```
procedure GapTest(RandGen : TtdBasePRNG; Lower, Upper : double;
    var ChiSquare : double; var DegrFreedom : integer);
var
    NumGaps : integer;
    GapLen : integer;
    i : integer;
    p : double;
    Expected : double;
    ChiSqVal : double;
    R : double;
    Bucket : array [0..pred(GapBucketCount)] of integer;
begin
    {вычислить длины пропусков и определить количество пропусков в каждой
    категории}
    FillChar(Bucket, sizeof(Bucket), 0);
    GapLen := 0;
    NumGaps := 0;
    while (NumGaps < GapsCount) do begin
        R := RandGen.AsDouble;
        if (Lower <= R) and (R < Upper) then begin
            if (GapLen >= GapBucketCount) then
                GapLen := pred(GapBucketCount);
            inc(Bucket[GapLen]);
            inc(NumGaps);
            GapLen := 0;
        end
        else
            if (GapLen < GapBucketCount) then
                inc(GapLen);
        end;
    p := Upper - Lower;
    ChiSqVal := 0.0;
    {обработать все категории, кроме последней}
    for i := 0 to GapBucketCount-2 do begin
        Expected := p * IntPower(1-p, i) * NumGaps;
        ChiSqVal := ChiSqVal + (Sqr(Expected - Bucket[i]) / Expected);
    end;
    {обработать последнюю категорию}
    i := pred(GapBucketCount);
    Expected := IntPower(1-p, i) * NumGaps;
    ChiSqVal := ChiSqVal + (Sqr(Expected - Bucket[i]) / Expected);
    {вернуть значения}
    ChiSquare := ChiSqVal;
    DegrFreedom := pred(GapBucketCount);
end;
```

Тест "покер"

Третий тест известен под названием "покер" (*poker test*). Случайные числа группируются в наборы по пять, а затем преобразуются в "карты", которые представляют собой цифры от 0 до 9. После этого определяется количество разных карт в каждом наборе (оно будет равно от одного до пяти) и полученные результаты разбиваются на категории. Поскольку вероятность пятикратного повторения одной и той же карты достаточно низка, случай выпадения только одной карты, как правило, включается в категорию "две разные цифры". К полученным четырем категориям применяется тест по критерию хи-квадрат (три степени свободы). Вероятность возникновения события для каждой категории вычислить не так уж легко (к тому же математические выкладки основаны на использовании комбинаторных значений, называемых числами Стерлинга), поэтому вычисления в этой книге не приводятся. Если вам интересно, то подробное описание можно найти в [11].

Листинг 6.7. Тест "покер"

```

procedure PokerTest(RandGen : TtdBasePRNG;
                    var ChiSquare : double;
                    var DegrFreedom : integer);

var
    i, j,
    BucketNumber,
    NumFives : integer;
    Accum, Divisor,
    Expected, ChiSqVal : double;
    Bucket : array [0..4] of integer;
    Flag : array [0..9] of boolean;
    p : array [0..4] of double;
begin
    {подготовительные операции}
    FillChar(Bucket, sizeof(Bucket), 0);
    NumFives := PokerCount div 5;
    {вычислить вероятности для каждой категории событий, алгоритм Кнута}
    Accum := 1.0;
    Divisor := IntPower(10.0, 5);
    for i := 0 to 4 do begin
        Accum := Accum * (10.0 - i);
        p[i] := Accum * Stirling(5, succ(i)) / Divisor;
    end;
    {для каждой группы из пяти случайных чисел преобразовать все значения и числа
     от 1 до 10, определить количество разных цифр}
    for i := 1 to NumFives do begin
        FillChar(Flag, sizeof(Flag), 0);
        for j := 1 to 5 do begin
            Flag[trunc(RandGen.AsDouble * 10.0)] := true;
        end;
    end;

```

```

    BucketNumber := -1;
    for j := 0 to 9 do
        if Flag[j] then inc(BucketNumber);
        inc(Bucket[BucketNumber]);
    end;
    {объединить две первые категории – это будет сумма категорий "все цифры
    одинаковы" и "две разные цифры"}
    inc(Bucket[1], Bucket[0]);
    Expected := (p[0]+p[1]) * NumFives;
    ChiSqVal := Sqr(Expected - Bucket[1]) / Expected;
    {обработать другие категории}
    for i := 2 to 4 do begin
        Expected := p[i] * NumFives;
        ChiSqVal := ChiSqVal + (Sqr(Expected - Bucket[i]) / Expected);
    end;
    {вернуть значения}
    ChiSquare := ChiSqVal;
    DegrFreedom := 3;
end;
```

Тест "сбор купонов"

Четвертый тест, который мы будем проводить, называется "сбор купонов" (*coupon collector's test*). Случайные числа считываются по одному и преобразуются в "купоны" – числа от 0 до 4. Фиксируется длина последовательности до получения полного комплекта купонов (т.е. цифр от 0 до 4). Очевидно, что получаемые длины будут от пяти и выше. После набора полного комплекта сбор купонов начинается снова. Длины последовательностей разбиваются на категории, к которым затем применяется тест по критерию хи-квадрат. Как правило, используются категории для длин от 5 до 19 и еще одна дополнительная категория для больших длин. Таким образом, мы получаем 16 категорий, а, следовательно, 15 степеней свободы. Как и в тесте "покер", вычисление вероятностей для каждой из категорий включает сложные математические выкладки, которые в этой книге не приводятся. Соответствующие подробности можно найти в [11].

Листинг 6.8. Тест "сбор купонов"

```

procedure CouponCollectorsTest(RandGen : TtdBasePRNG;
                                var ChiSquare : double;
                                var DegrFreedom : integer);
var
    NumSeqs, LenSeq, NumVals, NewVal,
    i : integer;
    Expected, ChiSqVal : double;
    Bucket : array [5..20] of integer;
    Occurs : array [0..4] of boolean;
    p : array [5..20] of double;
```

```

begin
  {вычислить вероятности для каждой категории событий, алгоритм Кнута}
  p[20] := 1.0;
  for i := 5 to 19 do begin
    p[i] := (120.0 * Stirling(i-1, 4)) / IntPower(5.0, i);
    p[20] := p[20] - p[i];
  end;
  NumSeqs := 0;
  FillChar(Bucket, sizeof(Bucket), 0);
  while (NumSeqs < CouponCount) do begin
    {продолжать сбор купонов (т.е. случайных чисел) до получения полного набора
     из пяти купонов}
    LenSeq := 0;
    NumVals := 0;
    FillChar(Occurs, sizeof(Occurs), 0);
    repeat
      inc(LenSeq);
      NewVal := trunc(RandGen.AsDouble * 5);
      if not Occurs[NewVal] then begin
        Occurs[NewVal] := true;
        inc(NumVals);
      end;
    until (NumVals = 5);
    {обновить значение для соответствующей категории в зависимости от количе-
     ства собранных купонов}
    if (LenSeq > 20) then
      LenSeq := 20;
    inc(Bucket[LenSeq]);
    inc(NumSeqs);
  end;
  {вычислить значение хи-квадрат}
  ChiSqVal := 0.0;
  for i := 5 to 20 do begin
    Expected := p[i] * NumSeqs;
    ChiSqVal := ChiSqVal + (Sqr(Expected - Bucket[i]) / Expected);
  end;
  {вернуть значения}
  ChiSquare := ChiSqVal;
  DegrFreedom := 15;
end;

```

Результаты выполнения тестов

В разделе сопровождающих эту книгу материалов, который расположен на Web-сайте издательства, можно найти тестовую программу, которая применяет все рассмотренные нами тесты к стандартному генератору случайных чисел Delphi и минимальному стандартному генератору случайных чисел. На рис. 6.1 приведены результаты проведения одного из тестов для генератора случайных чисел Delphi.

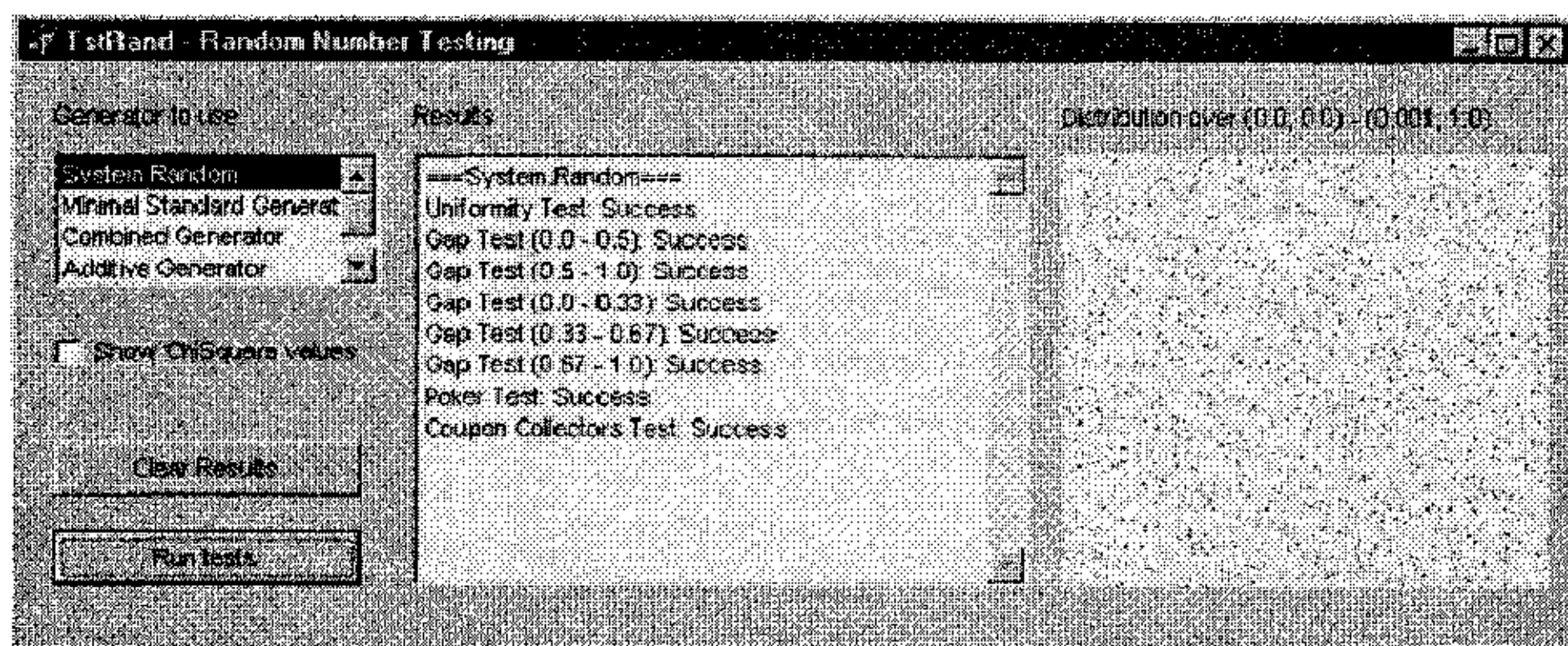


Рисунок 6.1. Тестирование стандартного генератора Delphi

Как видите, данный конкретный тест свидетельствует о том, что стандартный генератор Delphi успешно прошел проверку. (Под успешным прохождением теста понимается, что предложенная к тестированию последовательность случайных чисел не дает результатов, которые являются значимыми на уровне 5%.)

Окно в правой части окна программы представляет собой снимок случайных чисел, полученных на выходе генератора. Координаты точек вычисляются на основе двух случайных чисел: первого для оси x , а второго – для оси y . После этого точки выводятся в окне, если они находятся в прямоугольнике $(0.0, 0.0, 0.001, 1.0)$ или, другими словами, в прямоугольнике, нижний левый угол которого расположен в точке $(0.0, 0.0)$, а верхний правый – в точке $(0.001, 1.0)$. Чтобы распределение точек было удобнее изучать, прямоугольник растянут по оси x . Как видите, на рисунке точки случайным образом разбросаны по всему прямоугольнику. Никакой системы при этом не наблюдается.

У вас может возникнуть удивление, зачем мы так много говорим об этом окне. Посмотрите на рис. 6.2, на котором показана та же программа, но для минимального стандартного генератора случайных чисел. Как видите, и этот генератор успешно прошел все тесты, но посмотрите на распределение случайных точек. Очевидно, что генератор дает последовательность случайных чисел, которые при переносе их на график формируют определенный регулярный рисунок.

Регулярность минимального стандартного генератора не позволяет использовать его для некоторых приложений, особенно тех, которые требуют пар случайных чисел. Даже незначительной регулярности бывает достаточно для того, чтобы приложение давало неверные результаты. Кроме того, отсутствие регулярности в результатах стандартного генератора случайных чисел Delphi в двумерной плоскости не означает, что регулярности не будет в гиперплоскостях более высокой размерности. Существуют тесты, которые проверяют случайные числа на наличие регулярности в k -мерном пространстве, но давайте не будем погружаться в изучение слишком сложных тестов, а рассмотрим методы использования двух уже известных нам генераторов для дальнейшей рандомизации их выходных данных.

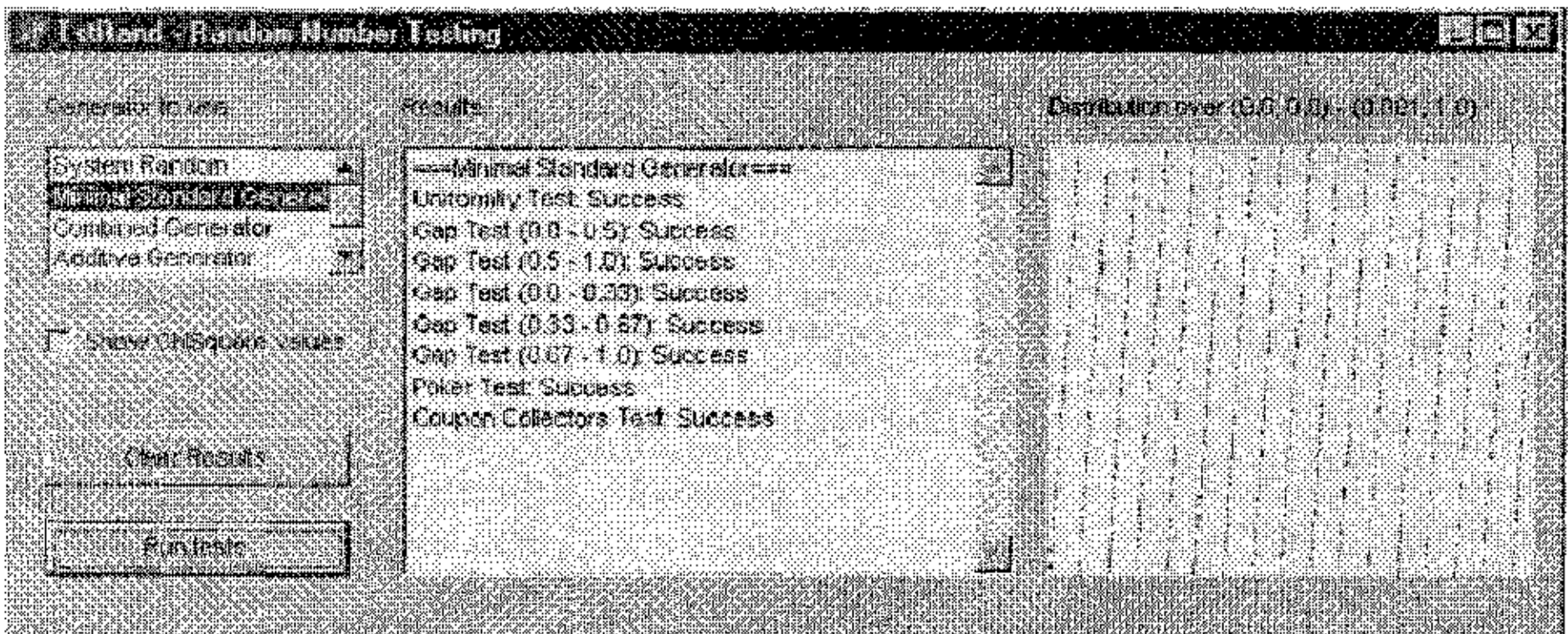


Рисунок 6.2. Тестирование минимального стандартного генератора

Мы рассмотрим три метода: первый известен как *комбинаторный*, второй – *аддитивный* и третий – метод *тасования*.

Комбинирование генераторов

Комбинирование генераторов заключается в параллельном использовании двух (или большего количества) мультипликативных линейных конгруэнтных генераторов с различными длинами циклов. Случайные числа генерируются обоими генераторами, а затем вычисляется их разность. Если получено отрицательное число, необходимо сделать его положительным, сложив его с длиной цикла первого генератора.

Листинг 6.9. Комбинирование генераторов

```
type
  TtdCombinedPRNG = class(TtdBasePRNG)
  private
    FSeed1 : longint;
    FSeed2 : longint;
  protected
    procedure cpSetSeed1(aValue : longint);
    procedure cpSetSeed2(aValue : longint);
  public
    constructor Create(aSeed1, aSeed2 : longint);
    function AsDouble : double; override;
    property Seed1 : longint read FSeed1 write cpSetSeed1;
    property Seed2 : longint read FSeed2 write cpSetSeed2;
  end;

constructor TtdCombinedPRNG.Create(aSeed1, aSeed2 : longint);
begin
  inherited Create;
  Seed1 := aSeed1;
  Seed2 := aSeed2;
end;
```

```

function TtdCombinedPRNG.AsDouble : double;
const
  a1 = 40014;
  m1 = 2147483563;
  q1 = 53668; {равно m1 div a1}
  r1 = 12211; {равно m1 mod a1}
  a2 = 40692;
  m2 = 2147483399;
  q2 = 52774; {равно m2 div a2}
  r2 = 3791; {равно m2 mod a2}
  OneOverM1 : double = 1.0 / 2147483563.0;
var
  k : longint;
  Z : longint;
begin
  {получить случайное число с помощью первого генератора}
  k := FSeed1 div q1;
  FSeed1 := (a1 * (FSeed1 - (k * q1))) - (k * r1);
  if (FSeed1 <= 0) then
    inc(FSeed1, m1);
  {получить случайное число с помощью второго генератора}
  k := FSeed2 div q2;
  FSeed2 := (a2 * (FSeed2 - (k * q2))) - (k * r2);
  if (FSeed2 <= 0) then
    inc(FSeed2, m2);
  {объединить два случайных числа}
  Z := FSeed1 - FSeed2;
  if (Z <= 0) then
    Z := Z + m1 - 1;
  Result := Z * OneOverM1;
end;

procedure TtdCombinedPRNG.cpSetSeed1(aValue : longint);
const
  m1 = 2147483563;
begin
  if (aValue > 0) then
    FSeed1 := aValue
  else
    FSeed1 := GetTimeAsLong;
  {убедиться, что случайное число находится в диапазоне от 1 до m-1}
  {включительно}
  if (FSeed1 >= m1-1) then
    FSeed1 := FSeed1 - (m1 - 1) + 1;
end;

procedure TtdCombinedPRNG.cpSetSeed2(aValue : longint);
const
  m2 = 2147483399;

```

```
begin
  if (aValue > 0) then
    FSeed2 := aValue
  else
    FSeed2 := GetTimeAsLong;
  {убедиться, что случайное число находится в диапазоне от 1 до m-1
   включительно}
  if (FSeed2 >= m2-1) then
    FSeed2 := FSeed2 - (m2 - 1) + 1;
end;
```

Как видите, код метода **AsDouble** в листинге 6.9 содержит два мультипликативных линейных конгруэнтных генератора: первый с параметрами $\{a, m\} = \{40014, 2147483563\}$ и второй с параметрами $\{a, m\} = \{40692, 2147483399\}$. Циклы обоих генераторов отличаются, но, тем не менее, близки к 2^{31} . Для преобразования промежуточного значения типа **longint** в значение типа **double** используется генератор с более длинным циклом.

Приведенный в листинге 6.9 генератор исключает двухмерную регулярность простого мультипликативного линейного конгруэнтного генератора, в чем можно убедиться с помощью программы тестирования. Можно показать, что длина цикла полученного комбинированного генератора составляет примерно $2 * 10^{18}$. (Для сравнения, длина цикла стандартного генератора Delphi примерно равна $4 * 10^9$.) Последовательность, вычисляемая с помощью комбинированного генератора полностью, определяется двумя начальными числами – по одному для каждого внутреннего генератора, в то время как для простого мультипликативного генератора было достаточно одного числа.

Аддитивные генераторы

Второй стандартный метод получения "более случайных" чисел от простого генератора называется *аддитивным*.

В соответствии с этим методом, мы инициализируем массив чисел с плавающей запятой с помощью простого генератора, например, минимального стандартного генератора случайных чисел, а затем используем два индекса в массиве для генерации последовательности случайных чисел на основе следующего алгоритма. Складываем значения, на которые указывают два индекса и записываем результат в элемент, на который указывает первый индекс (если полученная сумма будет больше 1.0, перед сохранением результата мы вычитаем из суммы значение 1.0). Возвращаем полученное значение в качестве следующего случайного числа. Перемещаем оба индекса вперед на одну позицию, при необходимости переходя от конца массива к его началу. Далее процесс повторяется снова.

Листинг 6.10. Аддитивный генератор

```
type
  TtdAdditiveGenerator = class(TtdBasePRNG)
  private
    FInx1 : integer;
    FInx2 : integer;
    FPRNG : TtdMinStandardPRNG;
    FTable : array [0..54] of double;
  protected
    procedure agSetSeed(aValue : longint);
    procedure agInitTable;
  public
    constructor Create(aSeed : longint);
    destructor Destroy; override;
    function AsDouble : double; override;
    property Seed : longint write agSetSeed;
  end;

constructor TtdAdditiveGenerator.Create(aSeed : longint);
begin
  inherited Create;
  FPRNG := TtdMinStandardPRNG.Create(aSeed);
  agInitTable;
  FInx1 := 54;
  FInx2 := 23;
end;

destructor TtdAdditiveGenerator.Destroy;
begin
  FPRNG.Free;
  inherited Destroy;
end;

procedure TtdAdditiveGenerator.agSetSeed(aValue : longint);
begin
  FPRNG.Seed := aValue;
  agInitTable;
end;

procedure TtdAdditiveGenerator.agInitTable;
var
  i : integer;
begin
  for i := 54 downto 0 do
    FTable[i] := FPRNG.AsDouble;
  end;
```

```
function TtdAdditiveGenerator.AsDouble : double;  
begin  
    Result := FTable[FInx1] + FTable[FInx2];  
    if (Result >= 1.0) then  
        Result := Result - 1.0;  
    FTable[FInx1] := Result;  
    inc(FInx1);  
    if (FInx1 >= 55) then  
        FInx1 := 0;  
    inc(FInx2);  
    if (FInx2 >= 55) then  
        FInx2 := 0;  
end;
```

Если внимательно изучить код, показанный в листинге 6.10, можно обратить внимание, что для формирования массива, используемого при работе аддитивного генератора, применяется минимальный стандартный генератор случайных чисел. Несмотря на то что мы не можем определить "начальное число" для аддитивного генератора (фактически по истечении некоторого времени начальное число эквивалентно всему массиву; внутренний генератор псевдослучайных чисел вызывается только 55 раз), мы можем его установить. При установке начального значения вызывается внутренний генератор, который заполняет массив, предназначенный для инициализации аддитивного генератора.

Длина массива, 55, и значения индексов, 54 и 23, – это не просто взятые наугад значения. Было показано, что они дают хорошие последовательности случайных чисел при генерации целых значений. (В книге [11] можно найти таблицы других удачных значений длины массива и индексов.)

Самым хорошим свойством данного генератора является длина цикла. Она просто огромна (при реализации на основе значений типа `longint` длина цикла будет составлять $2^{30}(2^{55} - 1)$, или приблизительно $3 * 10^{25}$). Даже если бы вы генерировали каждую секунду триллион случайных чисел, то для того, чтобы пройти весь цикл, потребовались бы годы.

Тасующие генераторы

И последний тип рассматриваемых нами генераторов, позволяющих получать "более случайные" числа, принадлежит к алгоритмам тасования. Здесь мы опишем генератор, реализованный на основе одного внутреннего генератора, хотя существуют и другие генераторы, аналогичным образом использующие два внутренних генератора.

Как и для аддитивного генератора, на первом этапе создается массив случайных чисел с плавающей запятой. Количество элементов в массиве не имеет особого значения. Кнут (Knuth) предложил использовать длины порядка 100. В нашем примере будет использоваться массив из 97 элементов – простое число, близкое к 100 [11]. (Кстати, применение простого числа не обязательно, оно просто выбрано

в качестве примера.) Заполним массив случайными числами, полученными с помощью минимального стандартного генератора случайных чисел. Введем новую вспомогательную переменную и установим ее значение равным следующему случайному числу в последовательности.

При необходимости генерации следующего случайного числа с помощью тасующего генератора, вспомогательная переменная используется для вычисления случайного числа из диапазона от 0 до 96. Устанавливаем значение вспомогательной переменной равным значению элемента с вычисленным индексом и заменяем элемент новым случайным числом, полученным от внутреннего генератора случайных чисел. В качестве результата тасующего генератора используется значение вспомогательной переменной.

Листинг 6.11. Тасующий генератор

```

type
  TtdShuffleGenerator = class(TtdBasePRNG)
    private
      FAux : double;
      FPRNG : TtdMinStandardPRNG;
      FTable : array [0..96] of double;
    protected
      procedure sgSetSeed(aValue : longint);
      procedure sgInitTable;
    public
      constructor Create(aSeed : longint);
      destructor Destroy; override;
      function AsDouble : double; override;
      property Seed : longint write sgSetSeed;
    end;
  constructor TtdShuffleGenerator.Create(aSeed : longint);
begin
  inherited Create;
  FPRNG := TtdMinStandardPRNG.Create(aSeed);
  sgInitTable;
end;
destructor TtdShuffleGenerator.Destroy;
begin
  FPRNG.Free;
  inherited Destroy;
end;
function TtdShuffleGenerator.AsDouble : double;
var
  Inx : integer;
begin
  Inx := Trunc(FAux * 97.0);
  Result := FTable[Inx];
  FAux := Result;
  FTable[Inx] := FPRNG.AsDouble;
end;
```

```
procedure TtdShuffleGenerator.sgSetSeed(aValue : longint);  
begin  
    FPRNG.Seed := aValue;  
    sgInitTable;  
end;  
procedure TtdShuffleGenerator.sgInitTable;  
var  
    i : integer;  
begin  
    for i := 96 downto 0 do  
        FTable[i] := FPRNG.AsDouble;  
    FAux := FPRNG.AsDouble;  
end;
```

Принимая во внимание, что приведенный генератор возвращает точно те же случайные числа, что и минимальный стандартный генератор, очень интересно обнаружить, что при проверке его в тестовой программе регулярность не проявляется.

Кроме того, следует отметить, что длина цикла тасующего генератора равна длине цикла внутреннего генератора. Суть тасующего генератора заключается в том, что генерируемые им числа выдаются в другом порядке. Длину цикла можно изменить, если для получения индексов использовать еще один генератор случайных чисел. При этом длина цикла соответственно увеличится. (Та же длина цикла получается при использовании двух внутренних генераторов в комбинированном генераторе.)

Выводы по алгоритмам генерации случайных чисел

В предыдущем разделе были рассмотрены несколько достаточно простых генераторов случайных чисел. Наилучшие последовательности чисел позволяют получить два последних генератора, но, к сожалению, они выдвигают жесткие требования к памяти (так, например, последний алгоритм для хранения внутренней таблицы требует почти 800 байт). Самым плохим из рассмотренных был минимальный стандартный генератор, по крайней мере, что касается наличия регулярности в генерируемых им последовательностях случайных чисел, которую, как было показано, можно устранить с помощью алгоритма тасования. Если говорить о личных предпочтениях, то автору книги наиболее импонирует аддитивный генератор: он прост, использует только оператор сложения и генерирует хорошие последовательности статистически независимых случайных чисел. Единственным его недостатком является то, что при необходимости сохранения состояния генератора, нужно сохранять массив и два индекса, что, по сравнению с одним значением начального числа типа **longint** для минимального стандартного генератора, может показаться слишком огромным объемом данных.

Другие распределения случайных чисел

Если случайные числа используются для моделирования некоторого процесса, то вы можете обнаружить, что все рассмотренные выше генераторы случайных чисел не позволяют решить поставленную задачу. Это вызвано *равномерным распределением* генерируемых ими случайных чисел, т.е. вероятность возникновения одного случайного числа равна вероятности возникновения любого другого числа. При проведении моделирования бывают необходимы случайные числа, распределенные не по равномерному закону. Тем не менее, для вычисления последовательностей с другими распределениями можно использовать уже изученные нами генераторы случайных чисел.

Вторым по значимости после равномерного является *нормальное* или *гауссово распределение*. Оно также известно под названием *распределение колоколообразной формы*, поскольку все точки данных расположены симметрично относительно среднего значения, причем, чем дальше точка от среднего значения, тем меньше вероятность ее получения. Нормальное распределение играет очень важную роль в статистике, где оно используется практически повсеместно. Например, рост людей 42-летнего возраста распределен в соответствии с нормальным распределением. Если попросить измерить длину стола нескольких человек с помощью линейки, длина которой намного короче, чем длина стола (другими словами, в случае существования элемента ошибки), полученный ответ будет соответствовать закону нормального распределения. И подобных примеров можно привести очень много.

Для нормально распределенного набора случайных чисел необходимо знать среднее значение и среднеквадратическое отклонение. Если эти параметры известны, генерация последовательности случайных чисел не представит особого труда. Для генерации мы будем использовать преобразование Бокса-Мюллера. Сами математические выкладки в этой книге не приводятся. Преобразование на своем входе требует два равномерно распределенных случайных числа, а на выходе генерирует два нормально распределенных случайных числа. Это не совсем удобно, поскольку нам, как правило, нужно только одно число за один раз. Однако второе число можно записать и выдать в качестве выходного значения при следующем вызове функции. Обратите внимание, что для многопоточных приложений предложенное решение приведет к тому, что функция не будет независимой от потоков, поскольку неиспользуемое значение придется хранить в глобальной переменной. Указанного недостатка можно избежать, если инкапсулировать вычисление случайных чисел в классе.

Обратите внимание, что мы исключаем тот редкий случай, когда оба равномерно распределенных случайных числа равны 0, и сумма их квадратов также равна 0, поскольку от этого значения в дальнейшем мы берем логарифм, который для 0 дает бесконечность. Поэтому подобной ситуации следует избегать.

Листинг 6.12. Случайные числа с нормальным распределением

```

var
  NRGNextNumber : double;
  NRGNextIsSet : boolean;
function NormalRandomNumber(aPRNG : TtdBasePRNG;
                             aMean : double;
                             aStdDev : double) : double;
var
  R1, R2 : double;
  RadiusSqr : double;
  Factor : double;
begin
  if NRGNextIsSet then begin
    Result := NRGNextNumber;
    NRGNextIsSet := false;
  end
  else begin
    {получить два числа, которые определяют точку внутри окружности единичного
    радиуса}
    repeat
      R1 := (2.0 * aPRNG.AsDouble) - 1.0;
      R2 := (2.0 * aPRNG.AsDouble) - 1.0;
      RadiusSqr := sqr(R1) + sqr(R2);
    until (RadiusSqr < 1.0) and (RadiusSqr > 0.0);
    {применить преобразование Бокса-Мюллера}
    Factor := sqrt(-2.0 * ln(RadiusSqr) / RadiusSqr);
    Result := R1 * Factor;
    NRGNextNumber := R2 * Factor;
    NRGNextIsSet := true;
  end;
  Result := (Result * aStdDev) + aMean;
end;

```

Еще одним важным распределением является *экспоненциальное*. Случайные числа, распределенные по этому закону, используются для моделирования ситуаций "времени прибытия", например, времени прибытия покупателей к кассе в супермаркете. Если в среднем покупатели подходят к кассе каждые x секунд, то время прибытия будет распределено по экспоненциальному закону со средним значением x .

Генерировать случайные числа, распределенные по экспоненциальному закону, достаточно просто. Не вдаваясь в математические подробности можно сказать, что если u – случайное число, распределенное по равномерному закону в диапазоне от 0.0 до 1.0, то e , которое равно

$$e = -x \ln(u)$$

будет случайным числом, распределенным по экспоненциальному закону со средним значением x .

Листинг 6.13. Случайные числа, распределенные по экспоненциальному закону

```
function ExponentialRandomNumber( aPRNG : TtdBasePRNG;  
                                   aMean : double) : double;  
  
var  
  R : double;  
begin  
  repeat  
    R := aPRNG.AsDouble;  
  until (R <> 0.0);  
  Result := -aMean * ln(R);  
end;
```

И снова обратите внимание, что исключается редкий случай, когда значение равномерно распределенного случайного числа равно 0, поскольку от него будет браться натуральный логарифм.

Списки с пропусками

После подробного описания нескольких генераторов случайных чисел, давайте рассмотрим структуру данных, которая для обеспечения высоких вероятностных характеристик быстродействия использует случайные числа.

Код класса для списков с пропусками можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDSkpLst.pas**.

Помните, в главе 4 мы говорили о том, что при необходимости поиска определенного значения в связном списке нужно начать с его начала и проходить по узлам с помощью указателей **Next** до тех пор, пока не будет найдено искомое значение. Если список был отсортирован, можно было воспользоваться алгоритмом бинарного поиска, который позволяет минимизировать количество выполняемых сравнений, тем не менее, при этом для прохода по списку также применялись указатели **Next**.

Вильям Пью (William Pugh) в 1990 году в своей статье "Списки с пропусками: вероятностная альтернатива сбалансированным деревьям" ("Skip Lists: Probabilistic Alternative to Balanced Trees") [18] показал, что существует более удобная альтернатива связным спискам, если мы готовы использовать узлы большего размера с большим количеством указателей.

Вильям Пью разработал вариант не совсем обычного связного списка. На своем самом низком уровне это двухсвязный список с прямым указателем на следующий узел и обратным указателем на предыдущий узел. Однако в некоторых узлах списка с пропусками имеется еще один прямой указатель, направленный на узел, расположенный на несколько позиций вперед. Такой указатель позволяет "перепрыгнуть" через целый ряд других, обычных узлов. Кроме того, в некоторых из этих расширенных узлов имеется еще один дополнительный указатель, который

5. Если элемент в узле **NextNode** меньше искомого, то искомый узел должен находиться после узла **NextNode**. Установить переменную **BeforeNode** на узел **NextNode** и перейти к шагу 3.
6. Если элемент в узле **NextNode** больше искомого, то искомый узел, если он присутствует в списке, должен находиться между узлами **BeforeNode** и **NextNode**. Уменьшаем значение переменной **LevelNumber** на единицу (другими словами, уменьшаем количество пропускаемых за один шаг узлов).
7. Если значение переменной **LevelNumber** равно 0 или больше, перейти к шагу 3. В противном случае искомый элемент в списке не найден, и если его необходимо вставить, то его позиция должна находиться между узлами **BeforeNode** и **NextNode**.

В соответствии с этим алгоритмом, при поиске узла g на рис. 6.3 мы начинаем с уровня 3 и начального узла. Переходим по указателю уровня 3 до узла h . Сравниваем h и g . Поскольку h больше g , уменьшаем уровень на единицу и начинаем сначала. По указателю второго уровня от начального узла переходим к узлу d . d меньше, чем g , следовательно, узел d становится новым начальным узлом. Снова переходим по указателю уровня 2 до узла h . Поскольку h больше, чем g , уменьшаем уровень на единицу. Переходим от узла d по указателю уровня 1 до узла f . Он меньше искомого, поэтому делаем его новым начальным узлом. Переходим по указателю уровня 1, и мы снова попадаем в узел h , который больше искомого. Снова понижаем уровень на единицу, переходим вперед по указателю уровня 0 и находим искомый узел g .

Таким образом, при поиске было пройдено шесть ссылок и выполнено шесть сравнений. Звучит не очень впечатляюще, особенно если учитывать, что в простом двухсвязном списке нам пришлось бы перейти по семи указателям и выполнить семь сравнений. Тем не менее, на рис. 6.3 принято допущение, что указатель уровня $n+1$ переходит на расстояние, в два раза превышающее расстояние перехода для указателя уровня n . Но обязательно ли соблюдать это условие? Почему в два раза, а не в три или пять? В списке с пропусками, который будет создан в этой главе, указатели первого уровня будут переходить через четыре узла, указатели второго уровня – через 16 узлов (т.е. $4 * 4$), указатели третьего уровня – через 64 узла (т.е. 4^3) и указатели уровня n – через 4^n узлов.

Подобный выбор расстояний переходов объясняется необходимостью балансировки степени возникновения переходов на большие расстояния на высоких уровнях и скорости поиска на уровне 0 при подходе к искомому узлу. Множитель 4 является хорошим компромиссом.

Насколько большими в таком случае будут узлы? Если предположить, что элемент, хранящийся в списке с пропусками, представляет собой указатель (как это было в главе 3), тогда размер узлов на уровне 0 будет равен, по крайней мере, размеру трех указателей (один указатель на данные, один – прямой указатель и один – обратный). Размер узлов на уровне 1 будет составлять четыре указателя

(поскольку в узле будет находиться два прямых указателя). Для уровня 2 размер узлов будет составлять пять указателей и т.д. Таким образом, на уровне n размер узлов будет равен не менее $n + 3$ указателям. (Если предположить, что размер указателя равен 4 байта, то мы получим узлы 12, 16, 20 и $4n + 12$ байт для узлов уровней 0, 1, 2 и n соответственно.) В действительности, для организации списка с пропусками требуется увеличить полученные размеры узлов, по крайней мере, на 1 байт, поскольку в каждом узле необходимо хранить уровень, к которому принадлежит данный узел.

Как вы уже знаете, узел уровня n содержит указатель на узел, находящийся впереди него на 4^n узлов. Если n равно 16, то указатель уровня n позволяет перейти вперед примерно на 4 миллиарда узлов – абсолютно недостижимое количество. Так, например, в 32-разрядной операционной системе каждый процесс имеет доступ к 4 миллиардам байт, в которых никак не могут разместиться 4 миллиарда узлов разного размера. На практике количество узлов, как правило, не будет превышать одного миллиона, поэтому указателей уровня 11 окажется вполне достаточно (т.е. общее количество уровней составит 12). На высшем уровне переход будет осуществляться на 4 миллиона узлов вперед.

На основе всего вышесказанного можно легко разработать структуру узла списка с пропусками. Это будет структура переменной длины, что несколько усложняет выделение памяти под узлы и ее освобождение. Структура узла приведена в листинге 6.14.

Листинг 6.14. Структура узла списка с пропусками

```
const
    tdcMaxSkipLevels = 12;
type
    PskNode = ^TskNode;
    TskNodeArray = array [0..pred(tdcMaxSkipLevels)] of PskNode;
    TskNode = packed record
        sknData : pointer;
        sknLevel : longint;
        sknPrev : PskNode;
        sknNext : TskNodeArray;
    end;
```

Мы не собираемся объявлять переменные типа **TskNode**. Фактически мы будем иметь дело исключительно с переменными типа **PskNode**, память под которые выделяется из кучи. Размер переменной будет вычисляться как

$$(3 + \text{sknLevel}) * \text{sizeof}(\text{pointer}) + \text{sizeof}(\text{longint})$$

Определившись со структурой узла списка с пропусками, можно перейти к рассмотрению реализации алгоритма поиска, которая приведена в листинге 6.15. Поиск представляет собой внутренний метод класса **TtdSkipList**. Он будет использоваться методами **Add** и **Remove** класса. И как мы сейчас увидим, еще одна его задач заключается в создании списка "предыдущих узлов" для каждого уровня.

Листинг 6.15. Поиск в списке с пропусками

```
function TtdSkipList.slSearchPrim(aItem : pointer;  
                                var aBeforeNodes : TskNodeArray) : boolean;  
var  
    Level : integer;  
    Walker : PskNode;  
    Temp : PskNode;  
    CompareResult : integer;  
begin  
    {заполнить весь массив BeforeNodes начальным узлом}  
    for Level := 0 to pred(tdcMaxSkipLevels) do  
        aBeforeNodes[Level] := FHead;  
    {инициализировать}  
    Walker := FHead;  
    Level := MaxLevel;  
    {начать поиск искомого узла}  
    while (Level >= 0) do begin  
        {найти следующий узел на этом уровне}  
        Temp := Walker^.sknNext[Level];  
        {если следующий узел является конечным, считать его большим, чем искомым  
        узел}  
        if (Temp = FTail) then  
            CompareResult := 1  
        {в противном случае сравнить данные следующего узла с искомыми данными}  
        else  
            CompareResult := FCompare(Temp^.sknData, aItem);  
        {если данные узла равны искомым данным, поиск завершен; выйти из функции}  
        if (CompareResult = 0) then begin  
            aBeforeNodes[Level] := Walker;  
            FCursor := Temp;  
            Result := true;  
            Exit;  
        end;  
        {если данные следующего узла меньше, чем искомые данные, перейти в  
        следующий узел}  
        if (CompareResult < 0) then begin  
            Walker := Temp;  
        end  
        {если данные следующего узла больше, чем искомые данные, понизить уровень}  
        else begin  
            aBeforeNodes[Level] := Walker;  
            dec(Level);  
        end;  
    end;  
    {если мы достигли этой точки, значит, искомый узел не найден}  
    Result := false;  
end;
```

Реализация метода начинается с заполнения всего массива **aBeforeNode** начальным узлом. Затем поиск начинается с высшего уровня списка (**MaxLevel**). Переход по указателям высшего уровня продолжается до тех пор, пока не будет найден узел, данные которого больше искомых. Обратите внимание, что обрабатывается специальный случай для концевой узла. Предполагается, что данные конечного узла больше любых других данных в списке. К сожалению, для класса, предназначенного для любых типов данных, подобная проверка обязательна, поскольку значение конечного узла установить заранее невозможно. Если же, с другой стороны, разрабатывается список с пропусками специально для строк, значение конечного узла можно выбрать таким, чтобы оно было больше любой строки, которая будет храниться в списке.

После этого производится сравнение. Если данные равны, искомый узел найден, и после установки нескольких переменных выполнение метода завершается. Если данные узла меньше, чем искомые данные, осуществляется переход по прямому указателю. В противном случае текущий уровень записывается в массив **aBeforeNode** и значение уровня уменьшается на единицу.

Вставка в список с пропусками

После изучения алгоритма поиска узла в существующем списке с пропусками, давайте рассмотрим алгоритм построения списка с помощью операции вставки нового узла. Вернувшись к рисунку 6.3, можно сказать, что задача сохранения однородной структуры списка после серии выполнения вставок и удалений кажется практически невыполнимой.

Достоинство алгоритма вставки, разработанного Пью, заключается в том, что Пью понимал, что построение абсолютно однородной структуры списка, по сути дела, невозможно или, по крайней мере, является сложной и трудоемкой операцией. Поэтому он предложил список с пропусками, который *в среднем* приближается к однородной структуре. В однородном списке с пропусками с множителем 4 один из четырех узлов больше трех других, поскольку он содержит дополнительный прямой указатель. В свою очередь, один из четырех этих больших узлов содержит еще один дополнительный указатель и т.д. В конце концов, можно прийти к выводу, что в однородном списке с пропусками три четверти всех узлов находятся на уровне 0, три шестнадцатых – на уровне 1, три шестьдесят четвертых – на уровне 2 и т.д. Другими словами, при случайном выборе узла можно установить следующие вероятности выбора узлов по уровням: 0.75 для уровня 0, 0.1875 для уровня 1, 0.046875 для уровня 2 и т.д.

Алгоритм вставки в список с пропусками учитывает эти вероятности таким образом, чтобы в общем на каждом уровне находилось требуемое количество узлов. Это означает, что *в среднем* вероятностный список с пропусками будет работать с той же эффективностью, что и "однородный": поиск некоторых узлов будет осуществляться чуть дольше, а других – чуть быстрее, однако, в среднем, поиск в ре-

альном списке с пропусками будет занимать примерно столько же времени, сколько и в идеальном однородном списке.

После такой теоретической подготовки можно перейти к описанию самого алгоритма вставки. Начинаем с пустого списка. Пустой список с пропусками содержит начальный узел уровня 11 и конечный узел уровня 0. Все прямые указатели начального узла указывают на конечный узел. Обратный указатель конечного узла указывает на начальный узел. Алгоритм вставки работает следующим образом:

1. Выполнить в списке поиск вставляемого элемента с одним дополнительным условием. При каждом понижении уровня сохранять значение переменной **BeforeNode**. В конце концов, мы получим набор значений **BeforeNode**, по одному для каждого уровня (поскольку количество уровней ограничено числом 12, для хранения уровней можно организовать простой массив из 12 элементов).
2. Если искомый элемент найден, вызвать ошибку (мы скоро скажем, по какой причине) и остановиться.
3. Узел не найден. Как уже упоминалось, нам известно, между какими узлами необходимо вставить новый элемент. Кроме того, при поиске мы достигли уровня 0.
4. Установить значение переменной **NewNode** равным нулю.
5. С помощью генератора случайных чисел вычислить случайное число в диапазоне от 0 до 1.
6. Если случайное число меньше 0.25, увеличить значение переменной **NewNode** на единицу.
7. Если значение переменной **NewNode** меньше или равно текущему максимальному уровню списка (т.е. 11), вернуться к шагу 5.
8. Если значение переменной **NewNode** больше текущего максимального уровня списка, присвоить ей значение максимального уровня плюс один.
9. Создать узел уровня **NewNode** и установить его указатель данных на вставляемый элемент.
10. Теперь новый узел нужно учесть во всех указателях вплоть до уровня **NewNode** (именно поэтому мы записывали все значения переменной **BeforeNode** при поиске на шаге 1). Для этого выполняется алгоритм "вставить после" для двухсвязного списка на уровне 0 и для всех односвязных списков для уровней от 1 до **NewNode**.

В приведенном алгоритме существуют несколько "странных" шагов, которые требуют дополнительных объяснений. Так, например, шаги 5, 6, 7 и 8, на которых вычисляется значение переменной **NewNode**, – для чего они нужны? Прежде всего, здесь вычисляется размер нового узла. Как вы, наверное, помните, мы пытаемся создать список с требуемым количеством узлов каждого уровня. Узел

уровня 0 должен создаваться в трех четвертях всех случаев, узел уровня 1 – в трех шестнадцатых всех случаев и т.д. Эти вычисления выполняются в цикле на шагах 5, 6 и 7. Во-вторых, на шаге 8 выполняется проверка того, что мы не вышли за границы максимального уровня списка. Не имеет смысла создавать узел, который находится на намного более высоком уровне, нежели текущий максимальный уровень. Поэтому максимальное значение уровня ограничивается увеличением уровня на единицу.

Шаг 2 также заслуживает отдельного рассмотрения. Фактически, в нем утверждается, что в списке с пропусками не могут храниться повторяющиеся элементы или, если выражаться более строго, элементы, в результате сравнения которых получается равенство. Почему? Представьте себе, что имеется список с пропусками, содержащий 42 узла, все значения которых равны a . В таком случае, что будет означать фраза: "Поиск узла a "? Учитывая саму природу списка с пропусками, на первом шаге поиска при переходе, скажем, на узел 35 будет найдено искомое значение a . Очевидно, что оно не будет ни первым в списке, ни последним – просто одним из 42 имеющихся в списке. Нужно ли в алгоритм вводить прохождение списка в обратном направлении, пока не будет найден первый узел со значением a ? Кто-то может сказать, что узлы с равными значениями должны находиться в списке в том порядке, в котором они вставлялись. Это означает, что при вставке элемента он будет добавляться в конец последовательности узлов с равными значениями, а при поиске нужно будет находить первый из повторяющихся узлов. Для алгоритма вставки при понижении уровней нужно сохранять список "предыдущих узлов". Эту операцию выполнить сложнее. По мнению автора книги, излишняя сложность алгоритмов для обеспечения возможности хранения в списке с пропусками узлов с одинаковыми значениями себя совершенно не оправдывает. Будем считать, что если существует вероятность повторения узлов, то мы знаем, как их различать между собой. В противном случае, они будут трактоваться как действительно один и тот же узел. Если мы можем различать повторяющиеся узлы, то можно предположить, что такая же возможность заложена и в функции сравнения. Следовательно, узлы уже не будут считаться повторениями.

В листинге 6.16 приведена реализация метода **Add** для класса списка с пропусками. В качестве генератора случайных чисел используется минимальный стандартный генератор, который мы изучали в первой части главы. Во всем остальном реализация следует алгоритму, описанному выше.

Листинг 6.16. Вставка в список с пропусками

```
procedure TtdSkipList.Add(aItem : pointer);  
var  
    i, Level : integer;  
    NewNode : PskNode;  
    BeforeNodes : TskNodeArray;
```



```

begin
  {выполнить поиск узла и заполнить значениями массив BeforeNodes}
  if slSearchPrim(aItem, BeforeNodes) then
    slError(tdeSklstDupItem, 'Add');
  {вычислить уровень для нового узла}
  Level := 0;
  while (Level <= MaxLevel) and (FPRNG.AsDouble < 0.25) do
    inc(Level);
  {если мы вышли за границы максимального уровня, сохранить новое значение в
  качестве максимального уровня}
  if (Level > MaxLevel) then
    inc(FMaxLevel);
  {выделить память для нового узла}
  NewNode := slAllocNode(Level);
  NewNode^.sknData := aItem;
  {восстановить указатели для уровня 0 – двухсвязный список}
  NewNode^.sknPrev := BeforeNodes[0];
  NewNode^.sknNext[0] := BeforeNodes[0].sknNext[0];
  BeforeNodes[0].sknNext[0] := NewNode;
  NewNode^.sknNext[0]^^.sknPrev := NewNode;
  {восстановить указатели для других уровней – односвязные списки}
  for i := 1 to Level do begin
    NewNode^.sknNext[i] := BeforeNodes[i].sknNext[i];
    BeforeNodes[i].sknNext[i] := NewNode;
  end;
  {теперь в список с пропусками добавлен новый узел}
  inc(FCount);
end;

```

Обратите внимание, что проверка в самом начале метода необходима для того, чтобы убедиться, что в списке не будет повторяющихся элементов. Кроме того, наличие повторяющихся элементов существенно уложило бы операцию удаления.

Удаление из списка с пропусками

Алгоритм удаления узла из списка с пропусками достаточно прост, несмотря на его длину. Он выглядит следующим образом:

1. Найти удаляемый узел с помощью обычного алгоритма поиска.
2. Предположим, что узел находится на уровне i . Сохранить узел, расположенный перед удаляемым и находящийся на том же уровне, что и i -тый элемент в массиве. Установить значение переменной **LevelNumber** равным i , а предыдущий узел записать в переменную **BeforeNode**.
3. Уменьшить значение переменной **LevelNumber** на единицу.
4. Если переменная **LevelNumber** содержит отрицательное значение, перейти к шагу 7.

5. Начиная с узла **BeforeNode**, переходить по указателям уровня **LevelNumber** вплоть до достижения удаляемого узла. При переходе по указателям уровня **LevelNumber** отслеживать родительские узлы всех проходимых узлов, что позволит идентифицировать узел, предшествующий удаляемому на уровне **LevelNumber**.
6. Записать узел, предшествующий удаляемому, в массив в элемент **LevelNumber**. Установить переменную **BeforeNode** равной этому узлу. Перейти к шагу 3.
7. Если мы достигли этого шага, у нас имеется массив предшествующих узлов для удаляемого узла для уровней от i до 0. Выполнить стандартную операцию "удалить после" для связного списка на каждом уровне.

Шаг 5 гарантированно будет работать (т.е. мы всегда найдем удаляемый узел), поскольку узел уровня n содержит указатели на всех уровнях до уровня n включительно.

В листинге 6.17 приведен код метода **Remove** для класса списка с пропусками. Он основан на описанном выше алгоритме.

Листинг 6.17. Удаление в списке с пропусками

```

procedure TtdSkipList.Remove(aItem : pointer);
var
    i, Level : integer;
    Temp : PskNode;
    BeforeNodes : TskNodeArray;
begin
    {выполнить поиск узла и заполнить значениями массив BeforeNodes}
    if not slSearchPrim(aItem, BeforeNodes) then
        slError(tdeSkpLstItemMissing, 'Remove');
    {действительные предшествующие узлы находятся на уровнях от максимального
    уровня списка до уровня данного узла; необходимо опередить предшествующие
    узлы для других уровней}
    Level := FCursor^.sknLevel;
    if (Level > 0) then begin
        for i := pred(Level) downto 0 do begin
            BeforeNodes[i] := BeforeNodes[i+1];
            while (BeforeNodes[i]^sknNext[i] <> FCursor) do
                BeforeNodes[i] := BeforeNodes[i]^sknNext[i];
            end;
        end;
    {восстановить указатели для уровня 0 – двухсвязный список}
    BeforeNodes[0]^sknNext[0] := FCursor^.sknNext[0];
    FCursor^.sknNext[0]^sknPrev := BeforeNodes[0];
    {восстановить указатели для других уровней – все односвязные списки}
    for i := 1 to Level do
        BeforeNodes[i].sknNext[i] := FCursor^.sknNext[i];

```

```

{восстановить положение курсора и освободить выделяемый узел}
Temp := FCursor;
FCursor := FCursor^.sknNext[0];
slFreeNode(Temp);
{теперь в списке с пропусками на один узел меньше}
dec(FCount);
end;

```

Полная реализация класса связного списка

Теперь, когда мы рассмотрели три сложных операции класса списка с пропусками, можно привести интерфейс самого класса. В отличие от класса связного списка, класс списка с пропусками не имеет функциональных возможностей, характерных для массивов. Дело не в том, что нельзя, например, организовать доступ к элементу списка по его индексу, а в том, что это первая структура данных в этой книге (в эту группу также можно включить хэш-таблицу и бинарное дерево), для которой такая операция просто не имеет смысла. Указание верного индекса для списка с пропусками требует прохода по самому нижнему уровню указателей. В этом случае нет необходимости организовывать столь сложную структуру узлов и указателей для обеспечения переходов различной длины. Поэтому для списков с пропусками обеспечиваются только функциональные возможности, характерные для баз данных: переход к следующему узлу и переход к предыдущему узлу. Очевидно, что для реализации таких методов необходимо ввести внутренний курсор. Методы **MoveNext** и **MovePrior** будут перемещать курсор, а метод **Examine** – возвращать элемент узла, в котором находится курсор. Метод **Delete** будет применяться для удаления элемента в позиции курсора и т.д.

Листинг 6.18. Интерфейс класса списка с пропусками

```

type
  TtdSkipList = class
  private
    FCompare : TtdCompareFunc;
    FCount : integer;
    FCursor : PskNode;
    FDispose : TtdDisposeProc;
    FHead : PskNode;
    FMaxLevel : integer;
    FName : TtdNameString;
    FPRNG : TtdMinStandardPRNG;
    FTail : PskNode;
  protected
    class function slAllocNode(aLevel : integer) : PskNode;
    procedure slError(aErrorCode : integer;
      const aMethodName : TtdNameString);
    procedure slFreeNode(aNode : PskNode);
    class procedure slGetNodeManagers;

```

```

function slSearchPrim(aItem : pointer;
var aBeforeNodes : TskNodeArray) : boolean;
public
  constructor Create( aCompare : TtdCompareFunc;
                     aDispose : TtdDisposeProc);
  destructor Destroy; override;
  procedure Add(aItem : pointer);
  procedure Clear;
  procedure Delete;
  function Examine : pointer;
  function IsAfterLast : boolean;
  function IsBeforeFirst : boolean;
  function IsEmpty : boolean;
  procedure MoveAfterLast;
  procedure MoveBeforeFirst;
  procedure MoveNext;
  procedure MovePrior;
  procedure Remove(aItem : pointer);
  function Search(aItem : pointer) : boolean;
  property Count : integer read FCount;
  property MaxLevel : integer read FMaxLevel;
  property Name : TtdNameString read FName write FName;
end;

```

Назначение большинства методов и свойств станет понятным, если вы вернетесь к описанию методов класса связанных списков, которое приводится в главе 3.

Как и для классов связанных списков, используется диспетчер узлов, который позволяет эффективно выделять и освобождать узлы. Тем не менее, для списков с пропусками имеется небольшое, однако важное отличие: узлы в списке с пропусками имеют разные размеры. Фактически в списке может быть до 12 видов узлов. Следовательно, для работы с узлами потребуется 12 диспетчеров. Процедура класса **slGetNodeManagers** выполняет инициализацию 12 диспетчеров узлов. Она вызывается в конструкторе **Create** класса списка с пропусками. Все объекты списков будут пользоваться одними и теми же диспетчерами. В заключительной части модуля все диспетчеры узлов удаляются.

Листинг 6.19. Конструктор и деструктор класса списка с пропусками

```

constructor TtdSkipList.Create(aCompare : TtdCompareFunc;
                              aDispose : TtdDisposeProc);

var
  i : integer;
begin
  inherited Create;
  {функция сравнения не может быть nil}
  if not Assigned(aCompare) then
    slError(tdeSkpLstNoCompare, 'Create');

```

```

{создать диспетчеры узлов}
slGetNodeManagers;
{выделить начальный узел}
FHead := slAllocNode(pred(tdcMaxSkipLevels));
FHead^.sknData := nil;
{выделить конечный узел}
FTail := slAllocNode(0);
FTail^.sknData := nil;
{задать прямые и обратные указатели для начального и конечного узлов}
for i := 0 to pred(tdcMaxSkipLevels) do
  FHead^.sknNext[i] := FTail;
FHead^.sknPrev := nil;
FTail^.sknNext[0] := nil;
FTail^.sknPrev := FHead;
{установить курсор на начальный узел}
FCursor := FHead;
{сохранить функцию сравнения и процедуру dispose}
FCompare := aCompare;
FDispose := aDispose;
{создать генератор случайных чисел}
FPRNG := TtdMinStandardPRNG.Create(0);
end;

destructor TtdSkipList.Destroy;
begin
  Clear;
  slFreeNode(FHead);
  slFreeNode(FTail);
  FPRNG.Free;
  inherited Destroy;
end;

```

Конструктор использует функцию сравнения, что позволяет корректно выбирать позицию вставляемых узлов (конечно, функция сравнения не может быть **nil**). Кроме того, в качестве входного параметра присутствует процедура **dispose**. Если она содержит **nil**, список с пропусками не является владельцем хранящихся в нем данных, поэтому при удалении списка данные удаляться не будут. В противном случае список является владельцем данных, и при его удалении данные также будут удаляться. Конструктор **Create** создает начальный и конечный узлы и устанавливает их указатели. И, наконец, создается генератор случайных чисел. Он впоследствии будет использоваться в методе **Add**.

Деструктор **Destroy** очищает содержимое списка с помощью метода **Clear**, освобождает начальный и конечный узлы и уничтожает генератор случайных чисел.

Метод **Clear** предназначен для очистки содержимого всех узлов, находящихся между начальным и конечным узлами, путем прохождения списка по указателям нижнего уровня и уничтожения узлов.

Листинг 6.20. Очистка содержимого списка с пропусками

```

procedure TtdSkipList.Clear;
var
  i : integer;
  Walker, Temp : PskNode;
begin
  {пройти по узлам уровня 0, освобождая все узлы}
  Walker := FHead^.sknNext[0];
  while (Walker <> FTail) do begin
    Temp := Walker;
    Walker := Walker^.sknNext[0];
    slFreeNode(Temp);
  end;
  {восстановить начальный и конечный узлы}
  for i := 0 to pred(tdcMaxSkipLevels) do
    FHead^.sknNext[i] := FTail;
  FTail^.sknPrev := FHead;
  FCount := 0;
end;

```

Методы выделения и уничтожения узлов достаточно просты. Они пользуются диспетчерами узлов класса и определяют требуемый диспетчер на основе значения уровня. Для метода выделения узла уровень передается в качестве входного параметра, для метода уничтожения оно определяется исходя из значения, полученного из освобождаемого узла.

Листинг 6.21. Выделение и уничтожение узлов в списке с пропусками

```

class function TtdSkipList.slAllocNode(aLevel : integer) : PskNode;
begin
  Result := SLNodeManager[aLevel].AllocNode;
  Result^.sknLevel := aLevel;
end;

procedure TtdSkipList.slFreeNode(aNode : PskNode);
begin
  if (aNode <> nil) then begin
    if Assigned(FDispose) then
      FDispose(aNode^.sknData);
    SLNodeManager[aNode^.sknLevel].FreeNode(aNode);
  end;
end;

class procedure TtdSkipList.slGetNodeManagers;
var
  i : integer;

```



```

begin
  {если диспетчеры узлов еще не созданы, создать их}
  if (SLNodeManager[0] = nil) then
    for i := 0 to pred(tdcMaxSkipLevels) do
      SLNodeManager[i] := TtdNodeManager.Create(NodeSize[i]);
end;

```

Обратите внимание, что метод уничтожения освобождает узлы только в том случае, когда список с пропусками создан в качестве владельца данных.

Остальные методы класса списка с пропусками еще проще – все они содержат всего несколько строк кода.

Листинг 6.22. Остальные методы класса списка с пропусками

```

procedure TtdSkipList.Delete;
begin
  {начальный и конечный узлы удалять нельзя}
  if (FCursor = FHead) or (FCursor = FTail) then
    slError(tdeListCannotDelete, 'Delete');
  {удалить узел в позиции курсора}
  Remove(FCursor^.sknData);
end;

function TtdSkipList.Examine : pointer;
begin
  Result := FCursor^.sknData;
end;

function TtdSkipList.IsAfterLast : boolean;
begin
  Result := FCursor = FTail;
end;

function TtdSkipList.IsBeforeFirst : boolean;
begin
  Result := FCursor = FHead;
end;

function TtdSkipList.IsEmpty : boolean;
begin
  Result := Count = 0;
end;

procedure TtdSkipList.MoveAfterLast;
begin
  FCursor := FTail;
end;

procedure TtdSkipList.MoveBeforeFirst;
begin
  FCursor := FHead;
end;

```

```
procedure TtdSkipList.MoveNext;  
begin  
  if (FCursor<>FTail) then  
    FCursor := FCursor^.sknNext[0];  
end;  
  
procedure TtdSkipList.MovePrior;  
begin  
  if (FCursor<>FHead) then  
    FCursor := FCursor^.sknPrev;  
end;
```

С использованием набора диспетчеров узлов для списка с пропусками связана одна проблема, о которой мы еще не говорили. Она не так очевидна для связных списков. А заключается она в пробуксовке. Проблема пробуксовки становится все более заметной при увеличении количества узлов до миллионов. Дело в том, что в списке с пропусками соседние узлы, скорее всего, будут находиться в разных страницах памяти. Поэтому при последовательном прохождении по списку от начала до конца на пути будут попадаться узлы разного размера, находящиеся в разных страницах памяти. Это приводит к подкачке страниц. К сожалению, мы никак не можем устранить свопинг (при использовании списков с несколькими миллионами узлов данные узлов в любом случае могут находиться в разных страницах). Проблему можно немного смягчить за счет использования стандартного диспетчера кучи Delphi. Тем не менее, даже в этом случае не исключается возможность возникновения пробуксовки.

Резюме

Эта глава была посвящена исследованию проблемы случайных чисел с нескольких точек зрения: с точки зрения генерирования последовательности случайных чисел и их применения для создания структуры данных не с прогнозируемыми, но вероятностными характеристиками.

Были приведены несколько методов генерации случайных чисел, распределенных по равномерному закону, в частности, мультипликативный конгруэнтный генератор, комбинационный и аддитивный генераторы, а также тасующий генератор. Для всех этих генераторов были представлены методы статистической оценки генерируемых ими последовательностей случайных чисел, которые позволяют оценить случайность получаемых результатов. Кроме того, были описаны два алгоритма генерации случайных чисел с другими распределениями: нормальным и экспоненциальным.

И, наконец, был рассмотрен список с пропусками – структура данных, используемая для хранения данных в отсортированном порядке. Было показано, каким образом случайные числа позволяют повысить характеристики быстродействия списков с пропусками.



Хеширование и хеш-таблицы

В главе 4 были рассмотрены алгоритмы поиска элемента в массиве (например, **TList**) или в связном списке. Наиболее быстрым из рассмотренных методов был бинарный поиск, для выполнения которого требовался отсортированный контейнер. Бинарный поиск представляет собой алгоритм класса $O(\log(n))$. Так, чтобы установить наличие или отсутствие заданного элемента в списке из 1000 элементов, требуется выполнить приблизительно 10 сравнений (поскольку $2^{10} = 1024$). Возможен ли еще более эффективный подход?

Если бы для выявления элемента обязательно нужно было использовать функцию сравнения, ответ на этот вопрос был бы отрицательным. Бинарный поиск — наиболее эффективный метод, который можно было бы использовать в этом случае.

Однако если бы элемент можно было связать с уникальным индексом, его можно было бы найти посредством однонаправленного действия: просто извлекая элемент, расположенный в позиции **MyList[Index]**. Это пример поиска с использованием индексирования по ключу, когда ключ элемента преобразуется в индекс, и элемент извлекается из массива с помощью этого индекса. Такой подход кардинально отличается от бинарного поиска, при котором, по существу, ключ элемента используется для перемещения по структуре с применением метода, в основе которого лежит сравнение.

Преобразование ключа элемента в значение индекса называется *хешированием* (*hashing*) и оно выполняется с помощью *функции хеширования* (*hash function*). Массив, используемый для хранения элементов, с которым используется значение индекса, называют *хеш-таблицей* (*hash table*).

Чтобы можно было выполнить поиск с использованием хеширования, требуется реализация двух отдельных алгоритмов. Первый – процесс хеширования, при помощи которого ключ элемента преобразуется в массив значений индекса. В идеальном случае различные ключи должны были бы хешироваться в различные значения индекса, но это нельзя гарантировать, и зачастую два различных ключа будут представлены одним и тем же значением индекса. Поэтому требуется второй алгоритм, определяющий наши действия в подобных случаях. Отображение двух или более ключей на один и тот же индекс по вполне понятной причине называют *конфликтом*, или *коллизией* (*collision*), а второй алгоритм, необходимый для исправления этой ситуации, называется *разрешением конфликтов* (*collision resolution*).

Хеш-таблица – прекрасный пример достижения компромисса между быстродействием и занимаемым объемом памяти. Если бы ключи элементов были уникальными значениями типа **word**, нужно было бы всего лишь создать 65536 элементов, и при этом можно было бы гарантировать нахождение элемента с конкретным ключом в результате выполнения одной операции. Однако если нужно хранить, скажем, не более 100 элементов, подобный подход оказывается чрезмерно расточительным. Да, возможно, этот метод работает достаточно быстро, но 99.85% области памяти массива пребывает пустой. Впадая в другую крайность, можно было бы выделить только необходимый объем памяти, выделяя массив требуемого размера, храня элементы в отсортированном порядке и используя бинарный поиск. Согласен, этот метод работает медленнее, но зато отсутствует бесполезно расходуемая память. Хеширование и хеш-таблицы позволяют выбрать золотую середину между этими двумя диаметрально противоположными подходами. Хеш-таблицы будут занимать больше места, причем некоторые элементы окажутся пустыми, тем не менее, использование функции хеширования позволяет найти элемент в результате очень небольшого числа обращений – обычно одного при тщательном выполнении хеширования.

Время от времени, с хеш-таблицами придется выполнять следующие операции: вставлять элементы в хеш-таблицу; выяснять, содержит ли хеш-таблица определенный элемент (хеш-таблицы обеспечивают очень быстрое выполнение поиска, чему собственно и посвящен этот раздел); удалять элементы из хеш-таблицы. Кроме того, желательно, чтобы при необходимости можно было расширять хеш-таблицу – т.е. требуется, чтобы размер хеш-таблицы можно было увеличивать с целью помещения в нее большего количества элементов, нежели предполагалось вначале.

Обратите внимание, что в приведенном описании функционирования хеш-таблиц ничего не говорится об извлечении записей в порядке следования ключей. Мы всего лишь пытаемся создать структуру данных, обеспечивающую очень быстрый доступ к конкретной записи с заданным ключом или очень быстрое выяснение того, что данный ключ отсутствует в структуре. Понятно, что нужно иметь возможность вставлять новые записи и их ключи, и, возможно, удалять существующие записи. Это все.

Если также необходимо, чтобы структура данных возвращала записи в порядке следования ключей, следует обратиться к деревьям бинарного поиска, к спискам с пропусками или к `TStringList`. Хеш-таблицы не обеспечивают извлечение в порядке следования ключей.

Однако вначале давайте проведем исследование функций хеширования, которые делают возможным выполнение указанных операций.

Функции хеширования

Алгоритм, который необходимо рассмотреть в первую очередь, – функция хеширования. Это подпрограмма, которая будет принимать ключ элемента и магическим образом преобразовывать его в значение индекса. Очевидно, что если в хеш-таблице предусмотрено место для n элементов, то функция хеширования должна создавать значения индексов, лежащие в диапазоне от 0 до $n - 1$ (как обычно, мы будем предполагать, что значения индексов начинаются с 0).

Поскольку ничего не говорилось о том, каким может быть тип ключа элемента, читателям должно быть понятно, что для различных типов ключей будут использоваться различные функции хеширования. Функция хеширования, предназначенная для целочисленного ключа, будет отличаться от предназначенной для строкового ключа. В идеале функция хеширования должна создавать значения индексов, которые внешне никак не связаны с ключами. Иначе говоря, в определенном смысле функция хеширования должна быть подобной функции рандомизации. Следовательно, очень похожие ключи должны были бы приводить к созданию совершенно различных хеш-значений.

Но все приведенные рассуждения являются чисто теоретическими. Чтобы получить представление о том, что хорошо, а что плохо, рассмотрим ряд функций хеширования.

Простейший случай – использование целочисленных ключей, когда элемент уникально идентифицируется целочисленным значением. Простейшей функцией хеширования, которую можно было бы использовать в этом случае, является операция деления по модулю. Если хеш-таблица содержит n элементов, хеш-значение ключа k вычисляется путем вычисления k по модулю n (если результат этой операции оказывается отрицательным, нужно просто добавить к нему n). Например, если n равно 16, то ключу 6 будет соответствовать индекс 6, ключу 44 – индекс 12 и т.д. В случае равномерного распределения значений ключей эта функция вполне подходила бы для работы, но в общем случае множество значений ключей не столь равномерно распределенное, и поэтому в качестве размера хеш-таблицы необходимо использовать простое число.

На практике можно сформулировать следующее правило создания хеш-таблиц: количество записей в хеш-таблице всегда должно быть равно простому числу. Для ознакомления с полным математическим обоснованием этого утверждения обратитесь к [13].

Для строковых ключей следует использовать метод, заключающийся в преобразовании строки в целочисленное значение с последующим применением операции деления по модулю для получения значения индекса, лежащего в диапазоне от 0 до $n - 1$.

Так как же преобразовать строку в целочисленное значение? Один из возможных способов предполагает использование длины строкового ключа. Преимущество применения этого метода состоит в простоте и высокой скорости выполнения. Однако его недостатком является генерирование множества конфликтов. На практике таких конфликтов возникает слишком много. Например, предположим, что нужно создать хеш-таблицу, которая должна содержать названия альбомов коллекции компакт-дисков. В частности, в принадлежащей автору коллекции компакт-дисков, насчитывающей несколько сот наименований, названия подавляющего большинства альбомов содержат от 2 до 20 символов. Использование длины названия альбома привело бы к возникновению множества конфликтов: альбом *Bilingual* в исполнении Pet Shop Boys конфликтовал бы с *Technique* в исполнении New Order и с *Mind Bomb* в исполнении The The. Таким образом, подобная функция хеширования совершенно неприемлема.

Более подходящей функцией хеширования было бы преобразование первых двух символов ключа в значение типа **word**. Затем для создания индекса можно было бы выполнить деление по модулю этого значения на длину хеш-таблицы. Такой подход вполне приемлем применительно к коллекции компакт-дисков рок- или поп-произведений, но не особенно подходит для коллекции компакт-дисков с классическими произведениями: все симфонии Бетховена преобразовывались бы в одно и то же хеш-значение, которое совпадало бы со значением для всех симфоний Рахманинова и для большинства симфоний Вогана-Вильямса.

Эту идею можно несколько развить и в качестве функции хеширования использовать деление по модулю суммы всех ASCII-значений символов в ключе на размер хеш-таблицы. Для коллекции компакт-дисков эта функция вполне подходит. К сожалению, во многих приложениях ключи могут быть анаграммами друг друга и, естественно, применение этой схемы приводило бы возникновению конфликтов.

Простая функция хеширования для строк

Похоже, что приведенные в предыдущем разделе рассуждения наталкивают на мысль о необходимости использования весовых коэффициентов, соответствующих позиции каждого символа в строке во избежание конфликтов при использовании анаграмм в качестве ключей. Это приводит к следующей реализации (исходный код можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHshBse.pas**).

Листинг 7.1. Простая функция хеширования строковых ключей

```
function TDSimpleHash( const aKey : string;  
                      aTableSize : integer) : integer;  
  
var  
  i : integer;  
  Hash : longint;  
begin  
  Hash := 0;  
  for i := 1 to length(aKey) do  
    Hash := ((Hash * 17) + ord(aKey[i])) mod aTableSize;  
  Result := Hash;  
  if (Result < 0) then  
    inc(Result, aTableSize);  
end;
```

Подпрограмма принимает два параметра. Первый из них – строка, хеш-значение которой требуется получить. Второй – размер хеш-таблицы (который, как мы приняли, должен быть простым числом). Алгоритм поддерживает постоянно изменяющееся хеш-значение, первоначально установленное равным нулю. Это хеш-значение изменяется для каждого символа в строке путем его умножения на небольшое простое число (17), добавления следующего символа и деления по модулю на размер хеш-таблицы.

Эта подпрограмма достаточно удачна. В ней для каждого символа выполняется всего несколько арифметических операций – к сожалению, в их числе и операция деления – и поэтому она достаточно эффективна. В реальных ситуациях строковые ключи оказываются в значительной степени подобными друг другу (вспомните, например, названия классических музыкальных произведений), а подпрограмма из похожих входных значений создает хеш-значения, которые выглядят случайными. Заключительный оператор **If** требуется потому, что промежуточное значение переменной **Hash** может быть отрицательным (такова неприятная "особенность" операции деления по модулю Delphi), а программа, вызывающая эту подпрограмму, будет ожидать результат, значение которого лежит в диапазоне от 0 до **TableSize-1**.

Функции хеширования PJW

В разделе, посвященном хеш-таблицам, книги *"Compilers: Principles, Techniques, and Tools"* ("Компиляторы: принципы, технологии, инструменты"), Ахо (Aho) и других, которая была издана Addison-Wesley [2], описана функция хеширования, созданная П. Дж. Вайнбергером (P. J. Weinberger). Эту подпрограмму называют также хешем Executable and Linking Format (формат исполняемых и компонуемых модулей), или ELF-хешем. Используемый в ней алгоритм аналогичен тому, что применяется в подпрограмме листинга 7.1. Единственное исключение состоит в том, что в этом алгоритме реализован эффект рандомизации, когда операция XOR

снова загружает старший полубайт действующей рабочей переменной хеша (полубайт, который должен исчезнуть в результате переполнения при выполнении следующей операции умножения), если он не равен нулю, в младшую часть переменной. Затем алгоритм устанавливает значение старшего полубайта равным нулю, в результате чего конечное хеш-значение всегда будет неотрицательным. (Исходный код функции можно найти на Web-сайте издательства, в разделе материалов. После выгрузки материалов отыщите среди них файл **TDHshBse.pas.**)

Листинг 7.2. Функция PJW хеширования строковых ключей

```
function TDPJWHash( const aKey : string;
                    aTableSize : integer) : integer;

var
  G : longint;
  i : integer;
  Hash : longint;
begin
  Hash := 0;
  for i := 1 to length(aKey) do begin
    Hash := (Hash shl 4) + ord(aKey[i]);
    G := Hash and longint($F0000000);
    if (G <> 0) then
      Hash := (Hash xor (G shr 24)) xor G;
  end;
  Result := Hash mod aTableSize;
end;
```

По ряду параметров эта функция превосходит простую функцию хеширования. Во-первых, благодаря описанному эффекту рандомизации. Во-вторых, для каждого символа выполняются только операции поразрядного сдвига и быстро выполняемые логические операции AND, OR, NOT и XOR (хотя функция и завершается операцией деления по модулю – похоже, что это неизбежно). Вероятно, в общем случае эта функция хеширования является наилучшей.

Мы не будем подробно останавливаться на других основных типах данных, поскольку в целом они успешно могут быть сведены к случаю целочисленных или строковых ключей. В качестве примера давайте рассмотрим хеширование дат, хранящихся в переменных **TDateTime**. В подавляющем большинстве приложений значения будут ограничиваться более поздними датами, чем заданная (например, 1 января 1975 года). В этом случае достаточно подходящей функцией хеширования была бы функция, выполняющая вычитание 1 января 1975 года из значения даты, для которого требуется получить хеш-значение, тем самым определяющая количество дней, истекших с момента начальной даты. Затем следует выполнить деление по модулю на размер хеш-таблицы.

Итак, мы подробно рассмотрели общие функции хеширования и выяснили, что иногда они будут генерировать одинаковые хеш-значения для различных ключей.

Но предположим, что у нас имеется известный список 100 строковых ключей. Существует ли какая-либо функция хеширования, которая будет генерировать уникальное хеш-значение для каждого из этих известных ключей, чтобы можно было разработать хеш-функцию, содержащую ровно 100 элементов? Функции хеширования такого типа называют совершенными. Безусловно, теоретически это возможно. Существует очень много таких функций (по существу, это равнозначно определению перестановок исходных ключей). Но как найти одну из таких функций? К сожалению, ответ на данный вопрос выходит за рамки этой книги. Даже Кнут (Knuth) [13] обходит эту тему. На практике совершенные функции хеширования представляют лишь теоретический интерес. Как только возникает потребность в другом ключе, совершенная функция хеширования разрушается и нам приходится разрабатывать следующую. Значительно удобнее считать, что никаких совершенных функций хеширования не существует, и иметь дело с неизбежными конфликтами, которые будут периодически возникать.

Разрешение конфликтов посредством линейного зондирования

Если количество элементов, которые, скорее всего, должна содержать хеш-таблица, известно, можно выделить место для хеш-таблицы, содержащей это количество элементов и небольшое число свободных ячеек "на всякий случай". Было разработано несколько алгоритмов, которые позволяют хранить элементы в таблице, используя пустые ячейки таблицы для хранения элементов, которые конфликтуют с уже имеющимися. Этот класс алгоритмов называют *схемами с открытой адресацией (open-addressing schemes)*. Простейшая схема с открытой адресацией – это *линейное зондирование (linear probing)*.

Поясним это на простом примере. Предположим, что мы вставляем фамилии в хеш-таблицу. До сих пор еще не описывалось, как выглядит хеш-таблица, но пока будем считать, что она представляет собой простой массив указателей элементов. Предположим, что существует функция хеширования того или иного вида.

Для начала вставим в пустую хеш-таблицу фамилию "Smith" (т.е. вставим элемент, ключом которого является "Smith"). Выполним хеширование ключа Smith с помощью функции хеширования и получим значение индекса, равное 42. Установим значение 42-го элемента хеш-таблицы равным Smith. Теперь записи хеш-таблицы вблизи этого элемента выглядят следующим образом:

Элемент 41: <пусто>
Элемент 42: Smith
Элемент 43: <пусто>

Это было достаточно просто. Теперь вставим фамилию "Jones". Необходимо выполнить те же действия, что и в предыдущем случае: следует вычислить хеш-значение ключа Jones, а затем вставить значение Jones по результирующему индексу. К сожалению, используемая функция хеширования имеет неизвестное проис-

хождение и для фамилии Jones генерирует хеш-значение, которое также равно 42. Если теперь обратиться к хеш-таблице, выясняется, что имеет место конфликт: ячейка 42 уже занята фамилией Smith. Что же делать? Используя линейное зондирование, мы проверяем следующую ячейку, чтобы выяснить, пуста ли она. Если да, то мы устанавливаем значение 43-го элемента хеш-таблицы равным Jones. (Если бы 43-я ячейка оказалась занятой, пришлось бы проверить следующую ячейку и т.д., возвращаясь к началу хеш-таблицы по достижении ее конца. Со временем мы нашли бы пустую ячейку либо вернулись бы к исходному состоянию, выяснив, что таблица заполнена.) Действие по проверке ячейки в хеш-таблице называется *зондированием* (*probing*), отсюда и название самого алгоритма – линейное зондирование.

Теперь хеш-таблица вблизи интересующей нас области выглядит следующим образом:

Элемент 41: <пусто>
Элемент 42: Smith
Элемент 43: Jones
Элемент 44: <пусто>

Вставив два элемента в гипотетическую хеш-таблицу, посмотрим, можно ли их снова найти. Выполним расчет хеш-значения для "Smith", в результате чего получаем индекс, равный 42. Обратившись к 42-му элементу, мы видим, что элемент Smith находится именно здесь. Выполнив расчет хеш-значения для Jones и получив индекс, равный 42, обратимся к 42-й ячейке. В ней находится элемент Smith, являющийся не тем, который мы ищем. Теперь нужно поступить так же, как и при вставке: обратиться к следующему элементу хеш-таблицы для выяснения того, совпадает ли он с искомым. В данном случае это так.

А как насчет поиска элемента, который отсутствует в таблице? Выполним поиск элемента "Brown". Реализуем хеширование, в результате чего будет получено значение индекса, равное 43. При обращении к 43-му элементу выясняется, что он соответствует элементу Jones. При переходе к следующему, 44-му, элементу выясняется, что он пуст. Теперь можно сделать вывод, что элемент Brown в хеш-таблице отсутствует.

Преимущества и недостатки линейного зондирования

В общем случае, если в хеш-таблице занято небольшое количество ячеек, можно надеяться, что для реализации большинства поисков, успешных или безрезультатных, придется выполнить всего одну-две операции зондирования. Однако когда таблица существенно заполнена элементами, количество пустых ячеек будет невелико, и в этом случае следует ожидать, что для выполнения безрезультатного поиска потребуются очень много операций зондирования (вплоть до $n-1$ зондирования при наличии только одной пустой ячейки). На практике, при использовании схемы с открытой адресацией, подобной линейному зондированию, имеет смысл обеспе-

читать *невозможность* перегрузки хеш-таблицы. В противном случае последовательности зондирования окажутся невероятно длинными.

Все сказанное не слишком сложно. Однако по поводу линейного зондирования стоит привести несколько соображений. Прежде всего, если хеш-таблица содержит n элементов, в нее можно вставить только n элементов (фактически, это справедливо по отношению к *любой* схеме с открытой адресацией). Способы расширения хеш-таблицы, в которой используется открытая адресация, мы рассмотрим чуть позже. Такие динамические хеш-таблицы позволили бы избежать длинных последовательностей зондирования, которые значительно снижают эффективность.

Второй момент – проблема кластеризации. При использовании линейного зондирования выясняется, что элементы имеют тенденцию к образованию непрерывных групп, или кластеров, занятых ячеек. Добавление новых элементов приводит к увеличению размеров групп, в результате чего конфликт вставленных элементов с элементом в кластере становится все более вероятным. И, конечно, с увеличением вероятности конфликта размеры кластеров также увеличиваются.

Это можно подтвердить математически, используя идеальную функцию хеширования, которая выполняет рандомизацию входных данных. Вставим элемент в пустую хеш-таблицу. Предположим, что в результате генерируется индекс x . Вставим еще один элемент. Поскольку результат действия функции хеширования по существу является случайным, вероятность попадания нового элемента в любую данную ячейку равна $1/n$. В частности, вероятность его конфликта с индексом x и вставки в ячейку $x + 1$ равна $1/n$. Кроме того, новый элемент может попасть непосредственно в ячейку $x - 1$ или $x + 1$. Вероятность обеих этих ситуаций также равна $1/n$, и, следовательно, вероятность того, что второй элемент образует кластер из двух ячеек, равна $3/n$.

После вставки второго элемента возможны три ситуации: два элемента образуют кластер, два элемента разделены одной пустой ячейкой или два элемента разделены более чем одной пустой ячейкой. Вероятности этих трех ситуаций соответственно равны $3/n$, $2/n$ и $(n - 5)/n$.

Вставим третий элемент. В первом случае это может привести к увеличению размера кластера с вероятностью $4/n$. Во втором случае это может привести к образованию кластера с вероятностью $5/n$. В третьем случае это может привести к образованию кластера с вероятностью $6/n$. Продолжая такие логические рассуждения, мы приходим к выводу, что вероятность образования кластера после вставки трех элементов равна $6/n - 8/n^2$, что приблизительно в два раза больше предыдущего значения вероятности. Можно было бы продолжить вычисление вероятностей для все большего количества элементов, но это лишено особого смысла. Вместо этого обратите внимание, что при вставке элемента и при наличии кластера из двух элементов вероятность увеличения этого кластера равна $4/n$. При наличии кластера с тремя элементами вероятность его увеличения возрастает до $5/n$ и т.д.

Как видите, после образования кластеров вероятность их увеличения все время возрастает.

Кластеры влияют на среднее количество зондирований, требуемых как для обнаружения существующего элемента (*попадания*), так и для выяснения того, что элемент в хеш-таблице отсутствует (*промаха*). Кнут показал, что среднее количество зондирований для обнаружения попадания приблизительно равно $\frac{1}{2}(1 + 1/(1 - x))$, где x – количество элементов в хеш-таблице, деленное на размер хеш-таблицы (эту величину называют *коэффициентом загрузки (load factor)*), а среднее количество зондирований для обнаружения промаха приблизительно равно $\frac{1}{2}(1 + 1/(1 - x)^2)$ [13]. Несмотря на простоту этих выражений, математические выкладки, приводящие к их получению, весьма сложны.

Используя приведенные формулы, можно показать, что если хеш-таблица заполнена примерно наполовину, для обнаружения попадания требуется в среднем приблизительно 1.5 зондирования, а для обнаружения промаха – 2.5 зондирования. Если же таблица заполнена на 90%, для обнаружения попадания требуется в среднем 5.5 зондирований, а для обнаружения промаха – 55.5 зондирований. Как видите, при использовании хеш-таблицы, в которой в качестве схемы разрешения конфликтов применяется линейное зондирование, таблица должна быть заполнена не более чем на две трети, чтобы эффективность оставалась приемлемой. Если это удастся, мы снизим влияние, которое кластеризация оказывает на эффективность хеш-таблицы.

Описанная особенность очень важна для хеш-таблиц, в которых в качестве метода разрешения конфликтов применяется линейное зондирование. *Нельзя допускать, чтобы хеш-таблица заполнялась в значительной степени. В противном случае длина последовательности зондирований становится чрезмерно большой. На протяжении многих лет я использую "две трети" в качестве предела заполнения хеш-таблиц, и этот критерий работает весьма успешно. Советую не допускать превышения указанного значения, но в любом случае стоит поэкспериментировать с меньшими значениями, например, с заполнением таблицы наполовину.*

Удаление элементов из хеш-таблицы с линейным зондированием

Прежде чем приступить к рассмотрению конкретного кода, рассмотрим удаление элементов из хеш-таблицы. Эта задача кажется достаточно простой: необходимо выполнить хеширование ключа элемента, который нужно удалить, найти его (используя необходимое количество зондирований), а затем пометить ячейку как пустую. К сожалению, применение этого упрощенного метода приводит к возникновению ряда проблем.

Предположим, что функция хеширования для ключей Smith, Jones и Brown создает следующие хеш-значения: 42, 42 и 43. Их добавление в хеш-таблицу в указанном порядке приводит к возникновению ситуации, показанной ниже:

Элемент 41: <пусто>
Элемент 42: Smith
Элемент 43: Jones
Элемент 44: Brown
Элемент 45: <пусто>

Иначе говоря, элемент Smith вставляется непосредственно в ячейку 42, элемент Jones вступает в конфликт с элементом Smith и попадает в ячейку 43, а элемент Brown вступает в конфликт с элементом Jones и попадает в ячейку 44.

Удалим элемент Jones, используя предложенный алгоритм удаления. В результате возникнет следующая ситуация:

Элемент 41: <пусто>
Элемент 42: Smith
Элемент 43: <пусто>
Элемент 44: Brown
Элемент 45: <пусто>

Теперь возникает проблема: попытайтесь найти элемент Brown. Ему соответствует индекс 43. Однако при просмотре ячейки 43 она оказывается пустой и, в соответствии с применяемым алгоритмом поиска, это означает, что элемент Brown в хеш-таблице отсутствует. Разумеется, это неверно.

Следовательно, при удалении элемента из хеш-таблицы, в которой применяется линейное зондирование, ячейку нельзя помечать как пустую: она может быть частью последовательности линейного зондирования. Вместо этого ячейку необходимо пометить как "удаленную" и слегка изменить алгоритм поиска, чтобы поиск продолжался при обнаружении удаленной ячейки.

Необходимо также слегка изменить и алгоритм вставки. В настоящее время, чтобы вставить элемент, мы осуществляем его поиск (т.е. выполняем хеширование ключа элемента и зондируем результирующий индекс и, возможно, последующие ячейки) до тех пор, пока не найдем элемент или не наткнемся на первую пустую ячейку. Обнаружив пустую ячейку, мы вставляем в нее новый элемент. (При обнаружении элемента можно либо сгенерировать сообщение об ошибке, либо просто заменить существующий элемент.)

Теперь же, ради эффективности, необходимо пометить первую удаленную ячейку, встретившуюся в ходе выполнения последовательности зондирования. Наличие пустой ячейки свидетельствует об отсутствии элемента. Однако мы не вставляем элемент в нее. Вместо этого, мы возвращаемся назад и вставляем элемент в первую пропущенную удаленную ячейку.

Возможность удаления элементов имеет одно важное следствие: слишком частое выполнение этой операции приведет к тому, что хеш-таблица будет заполнена ячейками, которые помечены как удаленные. Это, в свою очередь, увеличит среднее количество зондирований, требуемое для обнаружения попадания или промаха, тем самым снижая эффективность хеш-таблицы. Если количество удаленных ячеек

становится слишком большим, весьма желательно выделить новую хеш-таблицу и скопировать все элементы в нее.

Итак, если принять, что удаление элементов приведет к снижению эффективности хеш-таблицы, нельзя ли воспользоваться каким-то другим алгоритмом? Ответ, как это ни удивительно, положителен. Таким алгоритмом может быть следующий. Удалим элемент в соответствии с упрощенной схемой удаления; иначе говоря, пометим ячейку как пустую. Как только это выполнено, последующие элементы могут быть недоступны для этой операции, – точнее говоря, не *все* последующие элементы, а только те, которые находятся *в том же кластере*, что и только что удаленный элемент. Таким образом, мы всего лишь временно удаляем все элементы кластера, которые располагаются за полностью удаленным элементом, и снова их вставляем. Понятно, что обработка этих элементов выполняется по одному. При создании кода программы, нужно было бы начать с ячейки, расположенной за той, которая только что была помечена как пустая, и выполнять цикл до тех пор, пока не встретится пустая ячейка (обратите внимание, что в данном случае не следует беспокоиться о возникновении бесконечного цикла – известно, что с момента создания хеш-таблицы в ней появилась, по меньшей мере, одна пустая ячейка). Мы помечаем ячейку каждого элемента как пустую, а затем повторяем его вставку.

В заключение рассмотрим возможность преобразования хеш-таблицы в динамическую хеш-таблицу. Эта задача достаточно проста, хотя и трудоемка. Если коэффициент загрузки становится слишком большим, мы выделяем новую хеш-таблицу, которая больше старой (скажем, в два раза), переносим элементы исходной хеш-таблицы в новую (обратите внимание, что хеш-значения изменятся, поскольку новая хеш-таблица больше) и, наконец, освобождаем старую хеш-таблицу. Это все. Единственное небольшое "но" заключается в том, что в идеале желательно, чтобы размер новой хеш-таблицы был простым числом, как и размер исходной таблицы.

Класс хеш-таблиц с линейным зондированием

В листинге 7.3 приведен код интерфейса для хеш-таблицы с линейным зондированием (полный исходный код этого класса можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHshLnP.pas**). По поводу этой реализации следует привести ряд замечаний. Во-первых, мы принимаем соглашение, что ключом элемента является строка, отдельная от самого элемента. Это существенно упрощает как понимание кода, так и разработку и использование хеш-таблицы. В подавляющем большинстве случаев ключи все равно будут строками, а преобразование других типов данных в строки обычно не представляет особой сложности.

Второе соглашение состоит в том, что хотя класс будет допускать использование любой функции хеширования, функция должна иметь тип **TtdHashFunc**.

type

```
TtdHashFunc = function ( const aKey : string;
                        aTableSize : integer) : integer;
```

Если вы еще раз взглянете на листинги 7.1 и 7.2, то убедитесь, что в обоих случаях функции имеют этот тип.

Листинг 7.3. Хеш-таблица линейного зондирования TtdHashTableLinear

```
type
TtdHashTableLinear = class
  {хеш-таблица, в которой для разрешения конфликтов используется линейное
    зондирование}
private
  FCount : integer;
  FDispose: TtdDisposeProc;
  FHashFunc : TtdHashFunc;
  FName : TtdNameString;
  FTable : TtdRecordList;
protected
  procedure htlAlterTableSize(aNewTableSize : integer);
  procedure htlError(aErrorCode : integer;
                    const aMethodName : TtdNameString);
  procedure htlGrowTable;
  function htlIndexOf( const aKey : string;
                     var aSlot : pointer) : integer;
public
  constructor Create(aTableSize : integer;
                    aHashFunc : TtdHashFunc;
                    aDispose : TtdDisposeProc);
  destructor Destroy; override;
  procedure Delete(const aKey : string);
  procedure Empty;
  function Find(const aKey : string;
               var aItem : pointer) : boolean;
  procedure Insert(const aKey : string; aItem : pointer);
  property Count : integer
    read FCount;
  property Name : TtdNameString
    read FName write FName;
end;
```

С этим общедоступным интерфейсом не связаны какие-то неожиданности. Он содержит метод для вставки элемента вместе с его ключом, удаления элемента посредством использования его ключа и поиска элемента по его известному ключу. Метод **Clear** позволяет освободить хеш-таблицу от всех элементов.

Как видите, для хранения самой хеш-таблицы будет использоваться экземпляр **TtdRecordList**. Интерфейс класса не дает никакого представления о структуре

элементов хеш-таблицы, т.е. ячеек. Эта информация скрыта в разделе реализации модуля.

```

type
  PHashSlot = ^THashSlot;
  THashSlot = packed record
    {$IFDEF Delphi1}
    hsKey : PString;
    {$ELSE}
    hsKey : string;
    {$ENDIF}
    hsItem : pointer;
    hsInUse: boolean;
  end;

```

Ячейка представляет собой запись с тремя полями: ключом, собственно элементом и состоянием ячейки (независимо от того, используется оно или нет). В Delphi 1 ключ – это указатель строки, в то время как в последующих версиях он является длинной строкой (которая, естественно, представляет собой замаскированный указатель).

Конструктор **Create** выделяет экземпляр списка записей, а деструктор **Destroy** освобождает его.

Листинг 7.4. Конструктор и деструктор класса TtdHashTableLinear

```

constructor TtdHashTableLinear.Create( aTableSize : integer;
                                         aHashFunc : TtdHashFunc;
                                         aDispose : TtdDisposeProc );

begin
  inherited Create;
  FDispose := aDispose;
  if not Assigned(aHashFunc) then
    htlError(tdeHashTblNoHashFunc, 'Create');
  FHashFunc := aHashFunc;
  FTable := TtdRecordList.Create(sizeof(THashSlot));
  FTable.Name := ClassName + ': hash table';
  FTable.Count := TDGetClosestPrime(aTableSize);
end;

destructor TtdHashTableLinear.Destroy;
begin
  if (FTable<>nil) then begin
    Clear;
    FTable.Destroy;
  end;
  inherited Destroy;
end;

```

Конструктор обеспечивает присвоение функции хеширования. Применение хеш-таблицы без функции хеширования бессмысленно. Экземпляр **FTable** определяется таким образом, чтобы количество содержащихся в нем элементов было равно простому числу, ближайшему к значению, переданному в переменной **TableSize**. Деструктор обеспечивает освобождение хеш-таблицы (возможно, вначале придется удалить содержащиеся в ней элементы) перед освобождением экземпляра **FTable**.

Рассмотрим вставку нового элемента. Метод **Insert** принимает ключ элемента и сам элемент и добавляет их в хеш-таблицу.

Листинг 7.5. Вставка элемента в хеш-таблицу с линейным зондированием

```

procedure TtdHashTableLinear.Insert(const aKey : string;
                                     aItem : pointer);

var
  Slot : pointer;
begin
  if (htlIndexof(aKey, Slot) <> -1) then
    htlError(tdeHashTblKeyExists, 'Insert');
  if (Slot = nil) then
    htlError(tdeHashTblIsFull, 'Insert');
  with PHashSlot(Slot) do begin
    { $IFDEF Delphi1 }
    hsKey := NewStr(aKey);
    { $ELSE }
    hsKey := aKey;
    { $ENDIF }
    hsItem := aItem;
    hsInUse := true;
  end;
  inc(FCount);
  { увеличить таблицу, если она заполнена более чем на 2/3 }
  if ((FCount * 3) > (FTable.Count * 2)) then
    htlGrowTable;
end;

```

В данном случае защищенные вспомогательные методы выполняют несколько задач. Первый из них – **htlIndexof**. Этот метод предпринимает попытку найти ключ в хеш-таблице и в случае успеха возвращает его индекс и указатель на ячейку, которая содержит элемент (метод **Insert** воспринимает это как ошибку). Если ключ *не был* найден, метод возвращает значение -1, на этот раз с указателем на ячейку, в которую можно поместить элемент, что, собственно, и выполняется на следующем шаге. (Существует также третья возможность: метод **htlIndexof** возвращает значение -1 для индекса и ничего для ячейки; это считается признаком того, что таблица заполнена.) В конце подпрограммы выполняется проверка того, не заполнена ли хеш-таблица более чем на две трети, что, как говорилось ранее, служит хорошим показателем необходимости расширения хеш-таблицы с целью

снижения коэффициента загрузки (новая расширенная хеш-таблица должна быть заполнена примерно на одну треть). Метод **htlGrowTable** выполняет это.

Метод **Delete** удаляет элемент и его ключ из хеш-таблицы. Как мы уже видели, метод должен разрывать любые цепочки линейного зондирования.

Листинг 7.6. Удаление элемента из хеш-таблицы с линейным зондированием

```

procedure TtdHashTableLinear.Delete(const aKey : string);
var
  Inx : integer;
  ItemSlot : pointer;
  Slot : PHashSlot;
  Key : string;
  Item : pointer;
begin
  {поиск ключа}
  Inx := htlIndexOf(aKey, ItemSlot);
  if (Inx = -1) then
    htlError(tdeHashTblKeyNotFound, 'Delete');
  {удалить элемент и его ключ из данной ячейки}
  with PHashSlot(ItemSlot) do begin
    if Assigned(FDispose) then
      FDispose(hsItem);
    {$IFDEF Delphi1}
    DisposeStr(hsKey);
    {$ELSE}
    hsKey := '';
    {$ENDIF}
    hsInUse := false;
  end;
  dec(FCount);
  {повторно вставить все последующие элементы, предшествующие пустой ячейке}
  inc(Inx);
  if (Inx = FTable.Count) then
    Inx := 0;
  Slot := PHashSlot(FTable[Inx]);
  while Slot^.hsInUse do begin
    {сохранить элемент и ключ; удалить ключ из ячейки}
    Item := Slot^.hsItem;
    {$IFDEF Delphi1}
    Key := Slot^.hsKey^;
    DisposeStr(Slot^.hsKey);
    {$ELSE}
    Key := Slot^.hsKey;
    Slot^.hsKey := '';
    {$ENDIF}
    {пометить ячейку как пустую}
    Slot^.hsInUse := false;
  
```



```

dec(FCount);
{повторно вставить элемент и его ключ}
Insert(Key, Item);
{перейти к следующей ячейке}
inc(Inx);
if (Inx = FTable.Count) then
    Inx := 0;
    Slot := PHashSlot(FTable[Inx]);
end;
end;

```

Как и в предыдущем листинге, мы вызываем метод **htlIndexOf**, хотя на этот раз ошибка генерируется, если ключ не был найден. В случае обнаружения ключа метод возвращает указатель на ячейку, что позволяет избавиться от элемента (если это необходимо) и ключа. Состояние ячейки определяется как "не используется".

Теперь мы выполняем повторную вставку всех элементов, которые следуют за удаленным и находятся в одном с ним кластере. Из-за необходимости обрабатывать строки ключей в посещаемых ячейках описанная процедура кажется несколько запутанной. Во избежание утечек памяти, необходимо обеспечить освобождение строк ключей. Метод **Insert** будет перераспределять строки, независимо от выполняемых нами действий.

Метод **Clear** очень похож на метод **Delete**. Он используется для удаления всех элементов из хеш-таблицы.

Листинг 7.7. Опустошение хеш-таблицы с линейным зондированием

```

procedure TtdHashTableLinear.Clear;
var
    Inx : integer;
begin
    for Inx := 0 to pred(FTable.Count) do begin
        with PHashSlot(FTable[Inx])^ do begin
            if hsInUse then begin
                if Assigned(FDispose) then
                    FDispose(hsItem);
                {$IFDEF Delphi1}
                DisposeStr(hsKey);
                {$ELSE}
                hsKey := "";
                {$ENDIF}
            end;
            hsInUse := false;
        end;
    end;
    FCount := 0;
end;

```

Поскольку мы избавляемся от всех элементов в хеш-таблице, состояние всех ячеек можно установить (как только мы избавились от ключей и элементов в тех ячейках, которые используются) как "не используется".

Поиск элемента по его ключу выполняется методом **Find**. Уверен, что после ознакомления с методами **Insert** и **Delete** читатели догадываются, что это – всего лишь вызовы пресловутого метода **htlIndexOf**.

Листинг 7.8. Поиск элемента в хеш-таблице по ключу

```
function TtdHashTableLinear.Find( const aKey : string;
                                var aItem : pointer) : boolean;
var
  Slot : pointer;
begin
  if (htlIndexOf(aKey, Slot) <> -1) then begin
    Result := true;
    aItem := PHashSlot(Slot)^.hsItem;
  end
  else begin
    Result := false;
    aItem := nil;
  end;
end;
```

Как видите, все достаточно просто.

Методы, которые выполняют увеличение хеш-таблицы, используют еще один метод – **htlAlterTableSize**. Код обоих методов выглядит следующим образом.

Листинг 7.9. Изменение размера хеш-таблицы с линейным зондированием

```
procedure TtdHashTableLinear.htlAlterTableSize(aNewTableSize : integer);
var
  Inx : integer;
  OldTable : TtdRecordList;
begin
  {сохранить старую таблицу}
  OldTable := FTable;
  {распределить память под новую таблицу}
  FTable := TtdRecordList.Create(sizeof(THashSlot));
  try
    FTable.Count := aNewTableSize;
    {считывать старую таблицу и перенести ключи и элементы}
    FCount := 0;
    for Inx := 0 to pred(OldTable.Count) do
      with PHashSlot(OldTable[Inx])^ do
        if (hsState = hssInUse) then begin
          { $IFDEF Delphi }
          Insert(hsKey^, hsItem);
```

```

        DisposeStr (hsKey);
        { $ELSE}
        Insert (hsKey, hsItem);
        hsKey := '';
        { $ENDIF}
    end;
except
    {при возникновении исключения попытаться очистить хеш-таблицу и оставить
     ее в непротиворечивом состоянии}
    FTable.Free;
    FTable := OldTable;
    raise;
end;
{и, наконец, освободить старую таблицу}
OldTable.Free;
end;

procedure TtdHashTableLinear.htlGrowTable;
begin
    {увеличить размер таблицы приблизительно в два раза по сравнению с
     предыдущим}
    htlAlterTableSize (GetClosestPrime (succ (FTable.Count * 2)));
end;

```

Метод **htlAlterTableSize** содержит код выполнения этих операций. Он работает, сохраняя текущую хеш-таблицу (т.е. экземпляр списка записей), распределяя память под новую таблицу и, затем, просматривая все элементы в старой таблице (которые находятся в ячейках, помеченных как "используемые") и вставляя их в новую таблицу. В заключение, метод освобождает старую таблицу. Обратите внимание, что блок **Try..except** предпринимает попытку сохранить непротиворечивое состояние хеш-таблицы в случае возникновения исключения. Естественно, при этом предполагается, что в момент вызова метода хеш-таблица находилась в именно таком состоянии.

Излишне говорить, что расширение хеш-таблицы – довольно-таки трудоемкая операция (которая требует очень большого дополнительного объема свободной памяти – вдвое больше того, который уже был выделен). Всегда желательно приблизительно оценить общее количество строк, которые нужно вставить в хеш-таблицу, и добавить, скажем, еще половину этого количества строк. Результирующее значение можно использовать в качестве расчетного размера хеш-таблицы при ее создании. Такая оценка обеспечит нам определенную свободу действий при использовании хеш-таблицы.

Теперь пора разобраться с последним фрагментом головоломки: рассмотреть "закулисный" метод **htlIndexOf** – примитив, используемый методами **Insert**, **Delete** и **Find**.

Листинг 7.10. Примитив поиска ключа в хеш-таблице

```

function TtdHashTableLinear.htlIndexOf(const aKey : string;
                                     var aSlot : pointer) : integer;
var
  Inx : integer;
  CurSlot : PHashSlot;
  FirstInx : integer;
begin
  {вычислить хеш-значение строки, запомнить его, чтобы можно было установить,
   когда будет (если вообще будет) выполнен просмотр всех записей таблицы}
  Inx := FHashFunc(aKey, FTable.Count);
  FirstInx := Inx;
  {выполнить без каких-либо ограничений – при необходимости, выход из цикла
   можно будет осуществить всегда}
  while true do begin
    {для текущей ячейки}
    CurSlot := PHashSlot(FTable[Inx]);
    with CurSlot^ do begin
      if not hsInUse then begin
        {ячейка "пуста"; необходимо прекратить линейное зондирование
         и вернуть эту ячейку}
        aSlot := CurSlot;
        Result := -1;
        Exit;
      end
      else begin
        {ячейка "используется"; необходимо проверить, совпадает ли
         она с искомым ключом. Если да, то необходимо осуществить
         выход, возвращая индекс и ячейку}
        {$IFDEF Delphi1}
        if (hsKey^ = aKey) then begin
          {$ELSE}
          if (hsKey = aKey) then begin
            {$ENDIF}
            aSlot := CurSlot;
            Result := Inx;
            Exit;
          end;
        end;
      end;
    end;
    {на этот раз ключ или пустая ячейка не были найдены, поэтому необходимо
     увеличить значение индекса (при необходимости выполнив циклический воз-
     врат) и осуществить выход в случае возврата к начальной ячейке}
    inc(Inx);
  end;
end;

```

```
if (Inx = FTable.Count) then
  Inx := 0;
if (Inx = FirstInx) then begin
  aSlot := nil; {это сигнализирует о том, что таблица заполнена}
  Result := -1;
  Exit;
end;
end; {бесконечный цикл}
end;
```

После выполнения простой инициализации метод **htlIndexOf** вычисляет хеш-значение (т.е. значение индекса) для переданного ему ключа. Метод сохраняет это значение, чтобы можно было определить ситуацию, когда необходимо выполнить полный циклический возврат в хеш-таблице.

Метод определяет указатель на начальную ячейку. Мы просматриваем ячейку и выполняем различные операции, зависящие от состояния ячейки. Первый случай – когда ячейка пуста. Достижение этой точки означает, что ключ не был найден, поэтому метод возвращает указатель именно на эту ячейку. Естественно, в этом случае возвращаемое значение функции равно -1, что означает "ключ не найден".

Второй случай – когда ячейка используется. Для выяснения того, совпадают ли ключи, мы сравниваем ключ, хранящийся в ячейке, с ключом, переданным методу (обратите внимание, что мы выполняем поиск точного совпадения, т.е. сравнение с учетом регистра; если хотите выполнить сравнение без учета регистра, нужно использовать ключи, преобразованные в прописные буквы). Совпадение ключей свидетельствует об обнаружении искомого элемента. Поэтому программа возвращает указатель ячейки и устанавливает результат функции равным индексу ячейки.

Если в результате выполнения описанных операций сравнения выход из метода не был осуществлен, необходимо проверить следующую ячейку. Поэтому значение индекса **Inx** увеличивается, гарантируя циклический возврат и повторное выполнение цикла.

Обратите внимание, что проверка того, была ли посещена каждая отдельная ячейка, является несколько излишней. Хеш-таблица является динамической, и значение коэффициента загрузки будет поддерживаться между одной шестой и одной третей. То есть, в таблице всегда должны существовать ячейки, которые не используются. Однако, выполнение проверки – хорошая практика программирования, которая учитывает возможность изменения хеш-таблицы в будущем и того, что какой-либо код может привести к возникновению подобной ситуации.

Полный вариант кода класса **TtdHashTableLinear** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHshLnP.pas**.

Другие схемы открытой адресации

Хотя описанный класс хеш-таблиц был разработан для решения основной проблемы, возникающей при использовании схемы с открытой адресацией линейного зондирования (тенденции к кластеризации занятых ячеек), мы кратко рассмотрим несколько других схем с открытой адресацией.

Квадратичное зондирование

Первая из таких схем – *квадратичное зондирование* (*quadratic probing*). При использовании этого алгоритма контроль и предотвращение создания кластеров осуществляется путем проверки не следующей по порядку ячейки, а ячеек, которые расположены все дальше от исходной. Если первое зондирование оказывается безрезультатным, мы проверяем следующую ячейку. В случае неудачи этой попытки мы проверяем ячейку, которая расположена через четыре ячейки. Если и *эта* попытка неудачна, мы проверяем ячейку, расположенную через девять ячеек – и т.д., причем последующие зондирования выполняются для ячеек, расположенных через 16, 25, 36 и так далее ячеек. Этот алгоритм позволяет предотвратить образование кластеров, которые могут появляться в результате применения линейного зондирования, однако он может приводить и к ряду нежелательных проблем. Во-первых, если для многих ключей хеширование генерирует один и тот же индекс, все их последовательности зондирования должны будут выполняться вдоль одного и того же пути. В результате они образуют кластер, но такой, который кажется распределенным по хеш-таблице. Однако вторая проблема значительно серьезнее: квадратичное зондирование не гарантирует посещение всех ячеек. Максимум в чем можно быть уверенным, если размер таблицы равен простому числу, это в том, что квадратичное зондирование обеспечит посещение, по меньшей мере, половины ячеек хеш-таблицы. Таким образом, образом, можно говорить о выполнении задачи-минимум, но не задачи-максимум.

В этом легко убедиться. Начнем квадратичное зондирование с 0-й ячейки хеш-таблицы, содержащей 11 ячеек, и посмотрим, какие ячейки будут посещены при этом. Последовательность посещений выглядит следующим образом: 0, 1, 5, 3, 8, после чего зондирование снова начинается с ячейки 0. Мы никогда не посещаем ячейки 2, 4, 7, 9. По-моему, одной этой проблемы достаточно, чтобы в любом случае избегать применения квадратичного зондирования, хотя ее можно было бы избежать, не позволяя хеш-таблице заполняться более чем на половину.

Псевдослучайное зондирование

Следующая возможность – применение *псевдослучайного зондирования* (*pseudorandom probing*). Этот алгоритм требует использования генератора случайных чисел, который можно сбрасывать в определенный момент. Применительно к рассматриваемому алгоритму, из числа рассмотренных в 6 главе генераторов наиболее подошел бы минимальный стандартный генератор случайных чисел, по-

скольку его состояние однозначно определяется одним характеристическим значением – начальным числом. Алгоритм определяет следующую последовательность действий. Выполните хеширование ключа для получения хеш-значения, но *не* выполняйте деление по модулю на размер таблицы. Установите начальное значение генератора равным этому хеш-значению. Сгенерируйте первое случайное число с плавающей точкой (в диапазоне от 0 до 1) и умножьте его на размер таблицы для получения целочисленного значения в диапазоне от 0 до размера таблицы минус 1. Эта точка будет точкой первого зондирования. Если ячейка занята, сгенерируйте следующее случайное число, умножьте его на размер таблицы и снова выполните зондирование. Продолжайте выполнять упомянутые действия до тех пор, пока не найдете свободную ячейку. Поскольку при одном и том же заданном начальном значении генератор случайных чисел будет генерировать одни и те же случайные числа в одной и той же последовательности, для одного и того же хеш-значения всегда будет создаваться одна и та же последовательность зондирования.

Все это звучит достаточно обнадеживающе. Ценой ряда сложных и продолжительных вычислений, необходимых для получения случайного числа, этот алгоритм предотвратит образование кластеров, возникающих в результате линейного зондирования. Однако при этом возникает одна небольшая проблема: нет никакой гарантии, что рандомизированная последовательность обеспечит посещение каждой ячейки таблицы.

Нельзя не согласиться, что вероятность постоянного пропуска пустой ячейки достаточно низка, но это возможно, если таблица заполнена в значительной степени. Еще хуже то, что последовательность зондирований может стать очень большой до попадания в пустую ячейку. Следовательно, имеет смысл обеспечить невозможность значительного заполнения таблицы и изменение ее размера, если это происходит. С этого момента можно также продолжить использовать линейное зондирование с применением автоматически расширяемой хеш-таблицы. Это проще и быстрее.

Двойное хеширование

В заключение рассмотрим *двойное хеширование* (*double hashing*). На практике эта схема оказывается наиболее удачной из всех альтернативных схем с открытой адресацией. Итак, выполним хеширование ключа элемента в значение индекса. Назовем его h_1 . Выполним зондирование этой ячейки. Если она занята, выполним хеширование ключа путем применения совершенно иного и независимого алгоритма хеширования для получения другого значения индекса. Назовем его h_2 . Выполним зондирование ячейки $h_1 + h_2$. Если она занята, выполним зондирование ячейки $h_1 + 2h_2$, затем $h_1 + 3h_2$ и так далее (понятно, что все вычисления выполняются с делением по модулю на размер таблицы). Обоснование этого алгоритма следующее: если первая функция хеширования для двух ключей генерирует один и тот же индекс, очень маловероятно, что вторая функция хеширования сгенерирует для

них то же самое значение. Таким образом, два ключа, которые первоначально хешируются в одну и ту же ячейку, затем не будут соответствовать одной и той же последовательности зондирования. В результате мы можем ликвидировать "неизбежную" кластеризацию, сопряженную с линейным зондированием. Если размер таблицы равен простому числу, последовательность зондирования обеспечит посещение всех ячеек, прежде чем начнется сначала, что позволит избежать проблем, связанных с квадратичным и псевдослучайным зондированием. Единственная реальная проблема, возникающая при использовании двойного хеширования, – если не принимать во внимание необходимость вычисления дополнительного хеш-значения – состоит в том, что вторая функция хеширования по понятным причинам никогда не должна возвращать значение, равное 0. На практике эту проблему легко решить, выполняя деление по модулю на размер таблицы минус 1 (в результате мы получим значение в диапазоне от 0 до **TableSize-2**), а затем добавляя к результату единицу.

Например, при использовании строковых ключей можно было бы вызвать функцию хеширования Вайнбергера **TDPJWHash** для вычисления основных хеш-значений, а затем вызвать простую функцию хеширования **TDSimpleHash** для вычисления хеш-значений, которые будут использоваться для пропуска ячеек. Я предлагаю читателям самостоятельно выполнить это простое упражнение по реализации такой хеш-таблицы двойного хеширования.

Разрешение конфликтов посредством связывания

Если мы готовы использовать дополнительные ячейки, кроме тех, которые требуются самой хеш-таблице, можно воспользоваться другой эффективной схемой разрешения конфликтов – схемой с закрытой адресацией. Этот метод называется *связыванием* (*chaining*). В его основе лежит очень простой принцип: хеширование ключа элемента для получения значения индекса. Но вместо того, чтобы хранить элемент в ячейке, которая определяется значением индекса, мы сохраняем его в односвязном списке, помещенном в эту ячейку.

Поиск элемента достаточно прост. Мы хешируем ключ с целью получения соответствующего индекса, а затем выполняем поиск требуемого элемента в связном списке, помещенном в этой ячейке.

При выборе места вставки элемента в связный список доступно несколько возможностей. Его можно сохранить в начале связного списка или в конце, или же можно обеспечить, чтобы связные списки были упорядочены, и сохранить элемент в соответствующей позиции сортировки. Все три варианта имеют свои преимущества. Первый вариант означает, что недавно вставленные элементы будут найдены первыми в случае их поиска (имеет место своего рода эффект стека). Следовательно, этот метод наиболее подходит для тех приложений, в которых, скорее всего, поиск новых элементов будет выполняться чаще, нежели поиск старых. Вторым вариантом означает противоположное: первыми будут найдены "наиболее

старые" элементы (имеет место эффект типа очереди). Следовательно, он больше подходит для тех случаев, когда вероятность поиска более старых элементов больше вероятности поиска новых. Третий вариант предназначен для тех случаев, когда не существует предпочтений в отношении поиска более старых или новых элементов, но любой элемент нужно найти максимально быстро. В этом случае для облегчения поиска в связном списке можно прибегнуть к бинарному поиску. В действительности, если верить результатам выполненных мною тестов, третий вариант обеспечивает заметное преимущество только при наличии большого количества элементов в каждом связном списке. На практике лучше ограничить среднюю длину связных списков, при необходимости расширяя хеш-таблицу. Некоторые программисты экспериментировали, применяя деревья бинарного поиска к каждой ячейке (см. главу 8), а не к связным спискам. Однако полученные при этом преимущества оказались не особенно большими.

Первый упомянутый выше вариант вставки элемента в связный список имеет одно замечательное следствие. При успешном поиске элемента его можно переместить в начало связного списка, исходя из предположения, что если мы искали элемент, то, вероятно, довольно скоро будем искать его снова. Таким образом, элементы, поиск которых выполняется наиболее часто, будут перемещаться в верхнюю часть соответствующих связных списков.

Удаление элемента до смешного просто, если сравнить его с бегом по кругу, имевшим место при удалении элемента из хеш-таблицы с линейным зондированием. Достаточно найти элемент в соответствующем связном списке и разорвать связь с ним. Выполнение этих действий для односвязного списка описано в главе 3.

Преимущества и недостатки связывания

Преимущества связывания достаточно очевидны. Во-первых, в таблице, использующей связывание, никогда не возникнет ситуация нехватки места. Мы сколько угодно можем продолжать добавлять элементы в хеш-таблицу, и при этом будет происходить только увеличение связных списков. Реализация вставки и удаления крайне проста – действительно, большая часть работы была проделана в главе 3.

Несмотря на простоту, связывание имеет один важный недостаток. Он заключается в том, что никогда не возникает ситуация нехватки места! Проблема в том, что длина связных списков все больше и больше увеличивается. При этом время поиска в связных списках также увеличивается, а поскольку *любая* имеющая смысл операция, которую можно выполнять с хеш-таблицами, предполагает поиск элемента (вспомните пресловутый метод `htlIndexOf` класса хеш-таблиц с линейным зондированием), большая часть рабочего времени будет тратиться на поиск в связных списках.

Стоит отметить еще ряд обстоятельств. При использовании алгоритма разрешения конфликтов линейного зондирования мы сознательно старались минимизировать количество выполняемых зондирований, расширяя хеш-таблицу, когда ее коэффициент загрузки начинал превышать две третьих. Как следует из результатов

анализа, в этой ситуации для успешного поиска должно в среднем требоваться два зондирования, а для безрезультатного – пять. Подумайте, что означает зондирование. По существу, это сравнение ключей. Весь смысл применения хеш-таблицы заключался в уменьшении количества сравнений ключей до одного или двух. В противном случае вполне можно было бы выполнить бинарный поиск в отсортированном массиве строк. Что ж, при использовании связывания для разрешения конфликтов каждый раз, когда мы спускаемся по связному списку, пытаюсь найти конкретный ключ, для этого мы используем сравнение. Если прибегнуть к терминологии метода с открытой адресацией, то каждое сравнение можно сравнить с "зондированием". Так сколько же зондирований в среднем требуется для успешного поиска при использовании связывания? Для алгоритма связывания коэффициент загрузки по-прежнему вычисляется как число элементов, деленное на число ячеек (хотя на этот раз оно может иметь значение больше 1.0), и его можно представить средней длиной связных списков, присоединенных к ячейкам хеш-таблицы. Если коэффициент загрузки равен F , то среднее число зондирований для успешного поиска составит $F/2$. Для безрезультатного поиска среднее число зондирований равно F . (Эти результаты справедливы для несортированных связных списков. Если бы связные списки были отсортированы, значения были бы меньше – исходя из теории, оба значения нужно разделить на $\log_2(F)$). Как это ни удивительно, хотя на первый взгляд связывание кажется более удачным решением, нежели открытая адресация, при более внимательном рассмотрении этот метод оказывается не столь уж хорошим.

Суть всех выше приведенных рассуждений состоит в том, что в идеале необходимо увеличивать также хеш-таблицу, которая использует метод связывания для разрешения конфликтов. Использование методологии перемещения наиболее недавно использованных элементов в верхнюю часть соответствующих связных списков также обеспечивает существенный выигрыш в производительности.

Класс связных хеш-таблиц

Теперь пора рассмотреть какой-нибудь код. Общедоступный интерфейс к классу **TtdHashTableChained** в общих чертах не отличается от такового для класса **TtdHashTableLinear**. Различия между двумя этими классами проявляются в разделах **private** и **protected**.

Листинг 7.11. Класс TtdHashTableChained

```
type
    TtdHashChainUsage = ( {Применение цепочек хеш-таблицы..}
                          hcuFirst, {..вставка в начало}
                          hcuLast); {..вставка в конце}

type
    TtdHashTableChained = class
        {хеш-таблица, в которой для разрешения конфликтов используется связывание}
```

```

private
  FChainUsage : TtdHashChainUsage;
  FCount : integer;
  FDispose : TtdDisposeProc;
  FHashFunc : TtdHashFunc;
  FName : TtdNameString;
  FTable : TList;
  FNodeMgr : TtdNodeManager;
  FMaxLoadFactor : integer;
protected
  procedure htcSetMaxLoadFactor(aMLF : integer);
  procedure htcAllocHeads(aTable : TList);
  procedure htcAlterTableSize(aNewTableSize : integer);
  procedure htcError(aErrorCode : integer;
                    const aMethodName : TtdNameString);
  function htcFindPrim(const aKey : string;
                      var aInx : integer;
                      var aParent : pointer) : boolean;
  procedure htcFreeHeads(aTable : TList);
  procedure htcGrowTable;
public
  constructor Create(aTableSize : integer;
                    aHashFunc : TtdHashFunc;
                    aDispose : TtdDisposeProc);
  destructor Destroy; override;
  procedure Delete(const aKey : string);
  procedure Clear;
  function Find(const aKey : string;
               var aItem : pointer) : boolean;
  procedure Insert(const aKey : string; aItem : pointer);
  property Count : integer
    read FCount;
  property MaxLoadFactor : integer
    read FMaxLoadFactor write htcSetMaxLoadFactor;
  property Name : TtdNameString
    read FName write FName;
  property ChainUsage : TtdHashChainUsage
    read FChainUsage write FChainUsage;
end;

```

Мы объявили небольшой перечислимый тип **TtdHashChainUsage** для указания того, выполняется ли вставка элементов в начало или в конец связанного списка. Класс содержит свойство **ChainUsage**, которое указывает, какой метод следует использовать.

Свойство **MaxLoadFactor** служит для выполнения еще одной настройки. Оно определяет среднюю максимальную длину связанных списков, хранящихся в каждой из ячеек. Если средняя длина связанных списков становится слишком большой, класс увеличит внутреннюю хеш-таблицу, используемую для хранения элементов, и повторит их вставку.

Использование свойства **MaxLoadFactor** может оказаться затруднительным. Какое значение оно должно иметь? Вспомните, что его можно считать равным средней длине связанных списков, хранящихся в каждой из ячеек. Если придерживаться правила, применяемого для линейного зондирования, в соответствии с которым коэффициент загрузки выбирается так, чтобы для обнаружения промаха при поиске требовалось в среднем пять зондирований, то значение **MaxLoadFactor** должно быть равно пяти.

Однако необходимо учитывать еще одно соображение. При каждом зондировании выполняется сравнение искомого ключа с ключом элемента в хеш-таблице. Если сравнение занимает длительное время, как при поиске длинной строки, значение **MaxLoadFactor** должно быть меньше. Если сравнение выполняется значительно быстрее (например, в случае поиска короткой строки или целого числа), значение **MaxLoadFactor** может быть больше. Как и в случае любых настроек, чтобы добиться наилучших результатов, потребуется провести некоторый объем экспериментов.

Если внимательно присмотреться к коду, то мы увидим, что в нем используется хорошо известный нам класс **TtdNodeManager** (как именно – будет показано вскоре). Конструктор **Create**, как и **TList**, будет выделять один экземпляр этого класса. Деструктор **Destroy** будет освобождать оба эти экземпляра.

Листинг 7.12. Конструктор и деструктор класса **TtdHashTableChained**

```
constructor TtdHashTableChained.Create(aTableSize : integer;
                                     aHashFunc : TtdHashFunc;
                                     aDispose : TtdDisposeProc);
begin
  inherited Create;
  FDispose := aDispose;
  if not Assigned(aHashFunc) then
    htcError(tdeHashTblNoHashFunc, 'Create');
  FHashFunc := aHashFunc;
  FTable := TList.Create;
  FTable.Count := TDGetClosestPrime(aTableSize);
  FNodeMgr := TtdNodeManager.Create(sizeof(THashedItem));
  htcAllocHeads(FTable);
  FMaxLoadFactor := 5;
end;
```



```

destructor TtdHashTableChained.Destroy;
begin
  if (FTable<>nil) then begin
    Clear;
    htcFreeHeads(FTable);
    FTable.Destroy;
  end;
  FNodeMgr.Free;
  inherited Destroy;
end;

```

Созданный нами диспетчер узлов предназначен для работы с узлами **THashItem**. Он определяет структуру записей этого типа. Эта структура во многом аналогична структуре записей класса **TtdHashLinear**, за исключением того, что требуется связное поле и не требуется поле "используется" (все элементы в связном списке "используются" по определению; удаленные из хеш-таблицы элементы в связном списке отсутствуют).

```

type
  PHashedItem = ^THashedItem;
  THashedItem = packed record
    hiNext : PHashedItem;
    hiItem : pointer;
    {$IFDEF Delphi1}
    hiKey : PString;
    {$ELSE}
    hiKey : string;
    {$ENDIF}
  end;

```

Конструктор вызывает метод **htcAllocHeads** для создания первоначально пустой хеш-таблицы. Вот что должно при этом происходить. Каждая ячейка в хеш-таблице будет содержать указатель на односвязный список (поскольку каждая ячейка содержит только указатель, для хранения хеш-таблицы можно воспользоваться **TList**). Для упрощения вставки и удаления элементов мы выделяем заглавные узлы для каждого возможного связного списка, как было описано в главе 3. Естественно, деструктор должен освобождать эти заглавные узлы – данная операция выполняется при помощи метода **htcFreeHeads**.

Листинг 7.13. Выделение и освобождение заглавных узлов связных списков

```

procedure TtdHashTableChained.htcAllocHeads(aTable : TList);
var
  Inx : integer;
begin
  for Inx := 0 to pred(aTable.Count) do
    aTable.List^[Inx] := FNodeMgr AllocNodeClear;
  end;

```

```

procedure TtdHashTableChained.htcFreeHeads(aTable : TList);
var
  Inx : integer;
begin
  for Inx := 0 to pred(aTable.Count) do
    FNodeMgr.FreeNode(aTable.List^[Inx]);
end;

```

Теперь посмотрим, как выполняется вставка нового элемента и его строкового ключа в хеш-таблицу, которая использует связывание.

Листинг 7.14. Вставка нового элемента в хеш-таблицу со связыванием

```

procedure TtdHashTableChained.Insert(const aKey : string;
                                     aItem : pointer );
var
  Inx : integer;
  Parent : pointer;
  NewNode : PHashedItem;
begin
  if htcFindPrim(aKey, Inx, Parent) then
    htcError(tdeHashTblKeyExists, 'Insert');
    NewNode := FNodeMgr.AllocNodeClear;
    { $IFDEF Delphi1 }
    NewNode^.hiKey := NewStr(aKey);
    { $ELSE }
    NewNode^.hiKey := aKey;
    { $ENDIF }
    NewNode^.hiItem := aItem;
    NewNode^.hiNext := PHashedItem(Parent)^.hiNext;
    PHashedItem(Parent)^.hiNext := NewNode;
    inc(FCount);
    { увеличить таблицу, если значение коэффициента загрузки превышает
      максимальное значение }
    if (FCount > (FMaxLoadFactor * FTable.Count)) then
      htcGrowTable;
end;

```

Прежде всего, мы вызываем подпрограмму **htcFindPrim**. Она выполняет такую же операцию, как и **htl1IndexOf**, при использовании линейного зондирования: предпринимает попытку найти ключ и возвращает индекс ячейки, в которой он был найден. Однако этот метод создан с учетом применения связных списков. Если поиск ключа выполняется успешно, метод возвращает значение "истина", а также индекс ячейки хеш-таблицы и указатель на родительский узел элемента в связном списке. Почему родительский? Что ж, если вспомнить сказанное в главе 3, основные операции с односвязным списком предполагают вставку и удаление узла за данным узлом. Следовательно, целесообразнее, чтобы метод **htcFindPrim** возвращал родительский узел узла, в который выполняется вставка.

Если ключ не найден, метод **htcFindPrim** возвращает значение "ложь" и индекс ячейки, в которую нужно вставить элемент, и родительский узел, за которым его можно успешно вставить.

Итак, вернемся к методу **Insert**. Естественно, если ключ был найден, метод генерирует ошибку. В противном случае мы выделяем новый узел, устанавливаем элемент и ключ, а затем вставляем элемент непосредственно за данным узлом.

Если при этом коэффициент загрузки хеш-таблицы достигает максимального значения, мы расширяем хеш-таблицу.

Как легко догадаться, метод **Delete** работает аналогично.

Листинг 7.15. Удаление элемента из хеш-таблицы со связыванием

```
procedure TtdHashTableChained.Delete(const aKey : string);
var
  Inx : integer;
  Parent : pointer;
  Temp : PHashedItem;
begin
  {поиск ключа}
  if not htcFindPrim(aKey, Inx, Parent) then
    htcError(tdeHashTblKeyNotFound, 'Delete');
  {удалить элемент и ключ из данного узла}
  Temp := PHashedItem(Parent)^.hiNext;
  if Assigned(FDispose) then
    FDispose(Temp^.hiItem);
  {$IFDEF Delphi1}
  DisposeStr(Temp^.hiKey);
  {$ELSE}
  Temp^.hiKey := '';
  {$ENDIF}
  {разорвать связь с узлом и освободить его}
  PHashedItem(Parent)^.hiNext := Temp^.hiNext;
  FNodeMgr.FreeNode(Temp);
  dec(FCount);
end;
```

Мы предпринимаем попытку найти ключ (если он не найден, метод генерирует ошибку), а затем избавляемся от содержимого возвращенного элемента и удаляем его из связного списка. Обратите внимание на простоту кода обоих методов, **Insert** и **Delete**, обусловленную наличием заглавного узла в каждом связном списке. Не нужно беспокоиться о том, является ли родительский узел нулевым. Метод **htcFindPrim** всегда будет возвращать допустимый родительский узел.

Метод **Clear** очень похож на метод **Delete**, за исключением того, что мы просто стандартным образом удаляем все узлы из каждого связного списка (естественно, за исключением заглавных узлов).

Листинг 7.16. Очистка хеш-таблицы TtdHashTableChained

```

procedure TtdHashTableChained.Clear;
var
  Inx : integer;
  Temp, Walker : PHashedItem;
begin
  for Inx := 0 to pred(FTable.Count) do begin
    Walker := PHashedItem(FTable.List^[Inx]).hiNext;
    while (Walker <> nil) do begin
      if Assigned(FDispose) then
        FDispose(Walker^.hiItem);
      {$IFDEF Delphi1}
      DisposeStr(Walker^.hiKey);
      {$ELSE}
      Walker^.hiKey := '';
      {$ENDIF}
      Temp := Walker;
      Walker := Walker^.hiNext;
      FNodeMgr.FreeNode(Temp);
    end;
    PHashedItem(FTable.List^[Inx]).hiNext := nil;
  end;
  FCount := 0;
end;

```

Метод **Find** прост, поскольку основная часть работы выполняется вездесущим методом **htcFindPrim**.

Листинг 7.17. Поиск элемента в хеш-таблице со связыванием

```

function TtdHashTableChained.Find(const aKey : string;
                                   var aItem : pointer) : boolean;
var
  Inx : integer;
  Parent : pointer;
begin
  if htcFindPrim(aKey, Inx, Parent) then begin
    Result := true;
    aItem := PHashedItem(Parent).hiNext^.hiItem;
  end
  else begin
    Result := false;
    aItem := nil;
  end;
end;

```

Единственная небольшая сложность состоит в том, что метод **htcFindPrim** возвращает родительский узел действительно интересующего нас узла.

Увеличение хеш-таблицы – вовсе не то, что требуется, поскольку при этом необходимо выполнить очень много операций по перемещению данных. Однако класс содержит автоматическую операцию увеличения таблицы. Свойство **MaxLoadFactor** управляет тем, когда это происходит, вызывая метод **htcGrowTable** в случае вставки слишком большого количества элементов.

Листинг 7.18. Увеличение хеш-таблицы со связыванием

```
procedure TtdHashTableChained.htcGrowTable;
begin
    {увеличить размер таблицы примерно в два раза по сравнению с предыдущим
    размером}
    htcAlterTableSize(TDGetClosestPrime(succ(FTable.Count * 2)));
end;

procedure TtdHashTableChained.htcAlterTableSize(aNewTableSize : integer);
var
    Inx : integer;
    OldTable : TList;
    Walker, Temp : PHashedItem;
begin
    {сохранить старую таблицу}
    OldTable := FTable;
    {распределить новую таблицу}
    FTable := TList.Create;
    try
        FTable.Count := aNewTableSize;
        htcAllocHeads(FTable);
        {считывать старую таблицу и перенести ключи и элементы в новую таблицу
        посредством их вставки}
        FCount := 0;
        for Inx := 0 to pred(OldTable.Count) do begin
            Walker := PHashedItem(OldTable.List^[Inx]).hiNext;
            while (Walker<>nil) do begin
                {$IFDEF Delphi1}
                Insert(Walker^.hiKey^, Walker^.hiItem);
                {$ELSE}
                Insert(Walker^.hiKey, Walker^.hiItem);
                {$ENDIF}
                Walker := Walker^.hiNext;
            end;
        end;
    except
        {предпринять попытку очистки и сохранения хеш-таблицы в непротиворечивом
        состоянии в случае возникновения исключения}
        Clear;
        htcFreeHeads(FTable);
        FTable.Free;
        FTable := OldTable;
        raise;
    end;
```

```

{теперь новая таблица полностью заполнена всеми элементами и их ключами,
 поэтому необходимо уничтожить старую таблицу и ее связанные списки}
for Inx := 0 to pred(OldTable.Count) do begin
  Walker := PHashedItem(OldTable.List^[Inx]).hiNext;
  while (Walker<>nil) do begin
    {$IFDEF Delphi1}
    DisposeStr(Walker^.hiKey);
    {$ELSE}
    Walker^.hiKey := '';
    {$ENDIF}
    Temp := Walker;
    Walker := Walker^.hiNext;
    FNodeMgr.FreeNode(Temp);
  end;
  PHashedItem(OldTable.List^[Inx]).hiNext := nil;
end;
htcFreeHeads(OldTable);
OldTable.Free;
end;

```

В этом классе реализация метода **htcAlterTableSize** оказывается значительно сложнее, нежели в классе с линейным зондированием. Чтобы обеспечить корректное восстановление после возникновения исключительных состояний, возникающих при увеличении таблицы, увеличение выполняется в два этапа. Вначале элементы и их ключи копируются в новую таблицу большего размера. Затем, сразу по завершении первого этапа, мы избавляемся от узлов в меньшей таблице.

В заключение рассмотрим основной метод, используемый многими методами класса хеш-таблицы – **htcFindPrim** (листинг 7.19).

Листинг 7.19. Примитив для поиска элемента в хеш-таблице со связыванием

```

function TtdHashTableChained.htcFindPrim(const aKey : string;
                                         var aInx : integer;
                                         var aParent : pointer) : boolean;
var
  Inx : integer;
  Head, Walker, Parent : PHashedItem;
begin
  {вычислить хеш-значение для строки}
  Inx := FHashFunc(aKey, FTable.Count);
  {предположить, что связанный список существует в Inx-ой ячейке}
  Head := PHashedItem(FTable.List^[Inx]);
  {начать просмотр связанного списка с целью поиска ключа}
  Parent := Head;
  Walker := Head^.hiNext;

```



```

while (Walker<>nil) do begin
  { $IFDEF Delphi1 }
  if (Walker^.hiKey^ = aKey) then begin
    { $ELSE }
    if (Walker^.hiKey = aKey) then begin
      { $ENDIF }
      if (ChainUsage = hcuFirst) and (Parent Head) then begin
        Parent^.hiNext := Walker^.hiNext;
        Walker^.hiNext := Head^.hiNext;
        Head^.hiNext := Walker;
        Parent := Head;
      end;
      aInx := Inx;
      aParent := Parent;
      Result := true;
      Exit;
    end;
    Parent := Walker;
    Walker := Walker^.hiNext;
  end;
  { достижение этой точки свидетельствует о том, что ключ не найден }
  aInx := Inx;
  if ChainUsage = hcuLast then
    aParent := Parent
  else
    aParent := Head;
  Result := false;
end;

```

Работа метода начинается с хеширования переданного ему ключа. В результате мы получаем индекс ячейки, в которой найден заголовок связного списка. Мы перемещаемся вниз по связному списку до тех пор, пока не найдем искомый элемент или не встретим указатель **nil**, обозначающий конец списка. В ходе этого мы поддерживаем родительскую переменную, поскольку вызывающему методу нужно вернуть этот узел, а не указатель на узел элемента.

Если ключ не был найден, мы возвращаем узел в конце списка или заглавный узел – это определяется свойством **ChainUsage**. Если его значение установлено равным **hcuLast**, мы возвращаем последний узел, если оно установлено равным **hcuFirst** – заглавный узел. Таким образом, если вызывающим методом был метод **Insert**, можно быть уверенным, что новый элемент будет вставлен в требуемое место. Метод возвращает также индекс ячейки.

Если ключ был найден и значением свойства **ChainUsage** является **hcuFirst**, необходимо воспользоваться методологией "перемещения в начало" и переместить найденный элемент в первую позицию связного списка. Конечно, в случае использования односвязного списка эта операция проста и эффективна. И, наконец, мы возвращаем родительский узел и индекс ячейки.

Полный исходный код класса **TtdHashTableChained** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHshChn.pas**.

Разрешение конфликтов посредством группирования

Существует разновидность метода связывания для разрешения конфликтов, которая носит название *группирования в блоки (bucketing)*. Вместо помещения связанного списка в каждую ячейку, в нее помещается *группа*, которая по существу представляет собой массив элементов фиксированного размера. При создании хеш-таблицы необходимо выделить группу для каждой ячейки и пометить все элементы в каждой группе как "пустые".

Чтобы вставить элемент, мы хешируем ключ элемента с целью определения номера ячейки. Затем мы просматриваем все элементы в группе, пока не обнаружим элемент, помеченный как пустой, и присваиваем его элементу, который пытаемся вставить (понятно, что в случае присутствия элемента в группе генерируется ошибка).

Но что делать, если в группе больше нет пустых элементов? В этом случае доступны две возможности. Первая соответствует применению подхода линейного зондирования, а вторая – использованию групп переполнения.

Если в нужной группе не хватает места, первая возможность заключается в просмотре группы в следующей ячейке и проверке наличия в ней свободного места. Мы продолжаем выполнять эти действия, пока не отыщем пустой элемент, после чего вставляем в него элемент. Этот метод является прямой аналогией алгоритма линейного зондирования (действительно, если длина всех групп равна одному элементу, этот метод является методом линейного зондирования). Следовательно, он сопряжен с такими же проблемами. Например, удаление элементов из хеш-таблицы требует разрыва цепочек зондирования. Если группа не заполнена полностью, можно просто удалить из нее элемент и по одному переместить вверх элементы в группе. Если группа заполнена полностью, элементы этой группы могут вызывать переполнение, переходя в следующую, поэтому мы вынуждены либо помечать элемент как удаленный, либо повторять вставку последующих элементов, включая элементы в следующих группах, пока не встретится пустой элемент группы.

Вторая возможность заключается в использовании групп переполнения. В этом случае хеш-таблица содержит дополнительную группу, которая не используется при обычном применении хеш-таблицы. Эту группу называют *группой переполнения (overflow bucket)*. Если при вставке элемента в группе места под него не оказывается, мы ищем пустой элемент в группе переполнения и вставляем элемент туда. Таким образом, группа переполнения содержит элементы переполнения всех обычных групп. Если сама группа переполнения заполняется, мы просто выделяем еще одну группу и продолжаем выполнять описанные операции. Поиск элемента в

этой структуре данных предполагает просмотр каждого элемента в группе, в которую был хеширован ключ, и, если она заполнена, – просмотр каждого элемента в каждой группе переполнения, пока не будет найден пустой элемент. Удаление элемента из такой хеш-таблицы настолько не эффективно, что может оказаться вообще невозможным. Единственный целесообразный метод удаления – пометка элементов как удаленных. В противном случае, при необходимости удалить элемент из правильно заполненной группы придется повторно вставить каждый элемент, который присутствует в группах переполнения.

Так зачем же вообще рассматривать группирование? Что ж, вероятно, это лучшая структура данных для хеш-таблиц, хранящихся на диске.

Хеш-таблицы на диске

Контроллеры для таких устройств постоянного хранения данных, как жесткие и гибкие диски, дисководы Iomega Zip и ленточные накопители разработаны для поблочного считывания и записи данных. Обычно размер этих блоков равен какой-то степени двойки, например, 512, 1024 или 4096 байт. Поскольку контроллер должен выполнить считывание всего блока даже в том случае, когда требуется всего несколько байт, имеет смысл попытаться извлечь выгоду из подобного поведения.

Предположим, что требуется создать приложение, в котором используется большое количество записей, хранящихся на диске. Записи должны быть доступны в произвольном порядке по ключу. При этом каждая запись имеет отдельный уникальный строковый ключ. Это – идеальное применение для хеш-таблицы, однако записи столь многочисленны и велики, что невозможно выполнить их одновременное считывание в память. Действительно, делать это не имеет смысла, поскольку можно предположить, что большинство из них не будет требоваться в ходе любого отдельного сеанса работы программы.

Примером такого применения служит система пункта продажи в большом продуктовом супермаркете. В магазине могут продаваться сотни тысяч различных наименований товаров, из которых средний покупатель приобретает, скажем, не больше сотни (а то и десятка). Это идеальное применение для хеш-таблицы: каждый товар в магазине известен по его всемирному шифру продукта (UPC – Universal Product Code), т.е. 12-значному строковому значению, которое представляет собой уникальный ключ каждого товара. С учетом этого, приложение в кассовом пункте использует сканированный универсальный код товара с целью его хеширования в хеш-таблицу, а затем в запись, соответствующую товару.

Однако обратите внимание, что хранящаяся на диске хеш-таблица подходит только для обработки типа извлечения данных: получив ключ, она возвращает запись. Подобно своему аналогу, хранящемуся в памяти, хеш-таблица на диске не подходит для последовательного извлечения записей.

Прежде всего, создадим файл данных, состоящий из множества записей одинакового размера, каждая из которых описывает отдельный элемент. Естественно, для этого мы будем использовать класс **TtdRecordFile**, описанный в главе 2.

Файл индексов – это, по сути дела, второй файл базы данных хеш-информации. Как и в предыдущем случае, нам не нужно считывать в память весь файл индексов. Например, если бы каждый ключ содержал 10 цифр, а связанный с каждым ключом номер записи имел бы длину, равную 4 байтам, для хранения одного ключа требовалось бы 15 байт (исходя из предположения, что ключ содержит либо ноль в качестве символа-ограничителя, либо байт-префикс, определяющий его длину). Если бы хеш-таблица содержала 100 000 элементов, то для хранения ее индексов в памяти потребовалось бы минимум 1 500 000 байт. Разумеется, мы еще и выделяем дополнительную память под хранение строк ключей хеш-таблицы в куче, что приведет к еще большим накладным расходам (например, в 32-разрядной системе каждая строка кучи содержит три дополнительных символа типа **longint**). Значительно целесообразнее было бы считывать фрагменты индекса, когда в них возникает необходимость.

Применим метод группирования. В индексе хеш-таблицы мы используем группы фиксированного размера, чтобы при наличии ключа его можно было хешировать с целью получения требуемого номера группы, выполнить его считывание из файла индекса, а затем выполнить поиск требуемого ключа в группе. Эта методика выглядит достаточно простой, но, естественно, при этом необходимо предусмотреть действия на случай переполнения группы.

Расширяемое хеширование

Алгоритм, который нам нужно использовать, называется *расширяемым хешированием* (*extendible hashing*), и чтобы им можно было воспользоваться, необходимо вернуться к функции хеширования.

При использовании исходного метода мы знали размер хеш-таблицы, и поэтому, выполнив хеширование ключа, нужно было немедленно разделить его по модулю на размер таблицы и использовать результат как индекс в хеш-таблице. С другой стороны, в случае применения расширяемого хеширования размер хеш-таблицы не известен, поскольку при необходимости она будет увеличиваться во избежание переполнения. В ранее рассмотренных версиях хеш-таблиц при необходимости мы увеличивали их размер, следуя принципу повторного хеширования всех видимых элементов. В случае хеш-таблиц, хранящихся на диске, этот метод оказывается чересчур уж радикальным, поскольку большая часть времени тратилась бы на выполнение операций дискового ввода/вывода. При использовании расширяемого хеширования мы реорганизуем лишь небольшую часть хеш-таблицы – в основном, только группу переполнения.

Теперь функция хеширования будет возвращать значение типа **longint**. Если вернуться к первоначальной хеш-функции PJW, можно убедиться, что она вычис-

ляла 32-разрядное хеш-значение (фактически, 28-разрядное значение, поскольку значения четырех старших разрядов всегда устанавливались равными 0), а затем выполнялось деление по модулю этого значения на размер таблицы. При использовании расширяемого хеширования заключительное деление по модулю не выполняется. Вместо этого мы используем все хеш-значение полностью.

Означает ли это, что мы получаем хеш-таблицу с 268 миллионами ячеек? Нет, и это вполне согласуется со здравым смыслом. Мы используем только несколько разрядов хеш-значения, и по мере того, как таблица заполняется, мы начинаем использовать все больше разрядов хеш-значения.

Посмотрим, как работает этот алгоритм, на примере заполнения гипотетической хеш-таблицы. Первоначально в таблице имеется одна группа. Предположим, что каждая группа будет содержать 10 хеш-значений и номер записи каждого хеш-значения, чтобы ее можно было извлечь. Обратите внимание, что мы не помещаем в группы сами ключи. При использовании 28-разрядных хеш-значений, маловероятно, чтобы два ключа хешировались в одно и то же значение. (Фактически это будет происходить настолько редко, что для проверки ключей можно извлечь саму запись без заметного замедления всего процесса. Естественно, при этом предполагается, что используемая хеш-функция успешно справляется с рандомизацией.)

Начнем вставлять в таблицу хеш-значения вместе с номерами их записей. При наличии только одной группы их можно вставить только в одно место, поэтому после 10 вставок группа заполняется. Разобьем заполненную группу на две группы одинаковых размеров и повторим вставку всех элементов исходной группы в две новые группы. Причем все элементы, которые завершаются нулевым разрядом, поместим в одну группу, а завершающиеся единичным разрядом – в другую. Эти две группы имеют так называемую *разрядную глубину (bit-depth)*, равную одному разряду. Теперь при каждой вставке пары хеш-значение/номер записи она будет помещаться в первую или во вторую группу, в зависимости от последнего разряда хеш-значения.

Со временем мы заполним еще одну группу. Предположим, что это группа, в которую мы вставляли все хеш-значения, завершающиеся 0. Снова разобьем группу на две отдельные группы. На этот раз все элементы, хеш-значения которых заканчиваются двумя нулевыми разрядами, т.е. 00, будут помещаться в первую группу, а завершающиеся разрядами 10 – во вторую группу. Обе группы имеют разрядную глубину, равную 2. Поэтому для определения места вставки необходимо проверять два младших разряда хеш-значения. Теперь у нас имеются три группы: в первую вставляются элементы, завершающиеся разрядами 00, во вторую – разрядами 10, а в третью – просто 1.

Предположим, что мы продолжаем вставку и заполняем группу 10. Мы снова разбиваем заполненную группу на две и повторяем вставку ее элементов в две новые группы. На этот раз две новые группы будут принимать элементы, завершаю-

щиеся разрядами 010 и 110. Таким образом, теперь у нас имеются четыре группы: одна с разрядной глубиной, равной 1, в которую выполняется вставка хеш-значений, завершающихся 1, одна с разрядной глубиной равной 2, содержащая хеш-значения, которые завершаются разрядами 00, и две группы с разрядной глубиной, равной 3, которые предназначены для хеш-значений, завершающихся разрядами 010 и 110.

Почему-то есть уверенность, что читатели уже получили представление о работе расширяемого хеширования, – все остальное не представляет сложности.

Для поддержания отображения того, какие хеш-значения помещаются в те или иные группы, используется структура, называемая *каталогом* (*catalogue*). По существу каталог содержит список всех возможных окончаний групп и связанных с ними номеров групп. Вместо того чтобы поддерживать какой-либо причудливый набор значений разрядной глубины и номеров групп, выбранный методом проб и ошибок, каталог поддерживает собственное значение разрядной глубины, равное максимальной разрядной глубине группы, и имеет ячейку для каждого значения этой разрядной глубины.

В рассмотренном нами примере максимальная разрядная глубина группы была равна 3, поэтому разрядная глубина каталога также равна этому значению. Три разряда позволяют образовать восемь комбинаций разрядов: 000, 001, 010, 011, 100, 101, 110 и 111. Все комбинации, которые завершаются 1 (т.е. вторая, четвертая, шестая и восьмая), указывают на одну и ту же группу, принимающую элементы, хеш-значения которых завершаются 1. Аналогично, записи каталога для значений 000 и 100 указывают на одну и ту же группу, в которую помещаются элементы с хеш-значениями, завершающимися разрядами 00.

Однако эта схема не учитывает ряд особенностей. Две записи каталога, которые указывают на группу для элементов, хеш-значения которых завершаются разрядами 00, разделены тремя другими записями. Аналогично единственной группе, принимающей все элементы, хеш-значения которых завершаются 1, соответствуют четыре записи, равномерно распределенные по каталогу. При разбиении группы дополняющие друг друга группы не будут размещаться в каталоге по соседству. Для дальнейших рассуждений было бы проще предположить, что записи каталога, соответствующие одной группе, располагаются по соседству, чтобы при разбиении группы дополняющая первую группа помещалась непосредственно за ней.

Для достижения этого следует *инвертировать* последние разряды хеш-значения при вычислении индексной записи каталога. Так, например, если хеш-значение завершается разрядами 001, при поиске мы обратимся не к записи 001 каталога, а к записи 100 (4, которая соответствует инвертированному значению 001). В результате использование каталога значительно упрощается. В нашем примере хеш-значения, которые завершаются разрядами 00, помещаются в запись каталога 000 (0) или 001 (1). Хеш-значения, которые завершаются разрядами 010, помещаются в запись каталога 010 (2). Хеш-значения, которые завершаются разрядами 011, по-

мещаются в запись каталога 011 (3). И, наконец, хеш-значения, которые завершаются разрядом 1, помещаются в записи 100, 101, 110 или 111 (4, 5, 6, 7).

Вернемся немного назад, и вставим элементы в пустую хеш-таблицу, как это было сделано ранее. Выполняемые при этом действия показаны на рис. 7.1. Мы начинаем с каталога только с одной записью с индексом 0 (а). Принято считать, что в подобной ситуации разрядная глубина равна 0. Мы заполняем единственную группу (назовем ее А) и теперь ее нужно разбить. Вначале мы увеличиваем разрядную глубину каталога до 1. Иначе говоря, теперь он будет содержать две записи (b). В результате будут созданы две группы, на первую из которых указывает запись 0 (исходная запись А), а на вторую – запись 1, В (c). Все элементы, хеш-значения которых завершаются разрядом 0, помещаются в группу А, а остальные – в группу В. Снова заполним группу А. Теперь разрядную глубину каталога необходимо увеличить с 1 до 2, чтобы получить четыре группы, доступных для вставки. Перед разделением заполненной группы записи каталога 00 и 01 будут указывать на исходную группу А, а записи 10 и 11 – на группу В (d). Группа А разбивается на группу, которая принимает хеш-значения с окончанием 00 (снова А), и группу, которая принимает хеш-значения с окончанием 10, С. На группу А будет указывать запись 00 каталога, а на группу С – запись 01 (e). И, наконец, группа С (на которую указывает запись 01 каталога) заполняется. Нужно снова увеличить разрядную глубину каталога, на этот раз до трех разрядов.

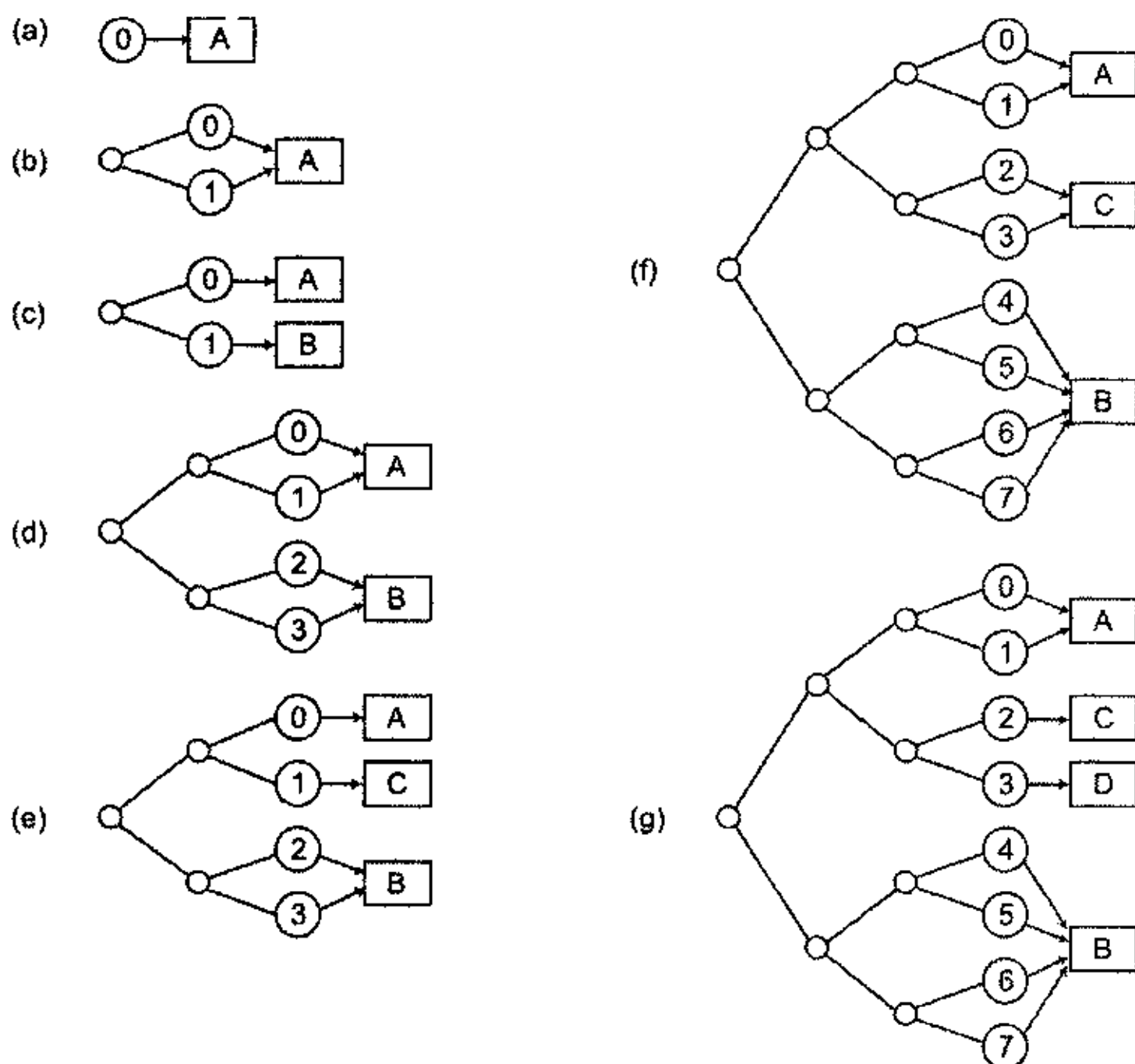


Рисунок 7.1.
Вставка в
расширяемую
хеш-таблицу

Теперь записи 000 и 001 указывают на запись А, записи 010 и 011 – на группу С, а 100, 101, 110 и 111 – на группу В (f). Мы создаем новую группу D и повторяем вставку всех элементов группы С в группы С и D, причем первая группа, которой соответствует запись каталога 010 (2), принимает хеш-значения с окончанием 010, а вторая, которой соответствует запись каталога 011 (3), – хеш-значения с окончанием 110 (g).

Теперь, когда мы рассмотрели основной алгоритм, пора применить его на практике. Прежде всего, отметим следующее: все фрагменты расширяемой хеш-таблицы хранятся в отдельных файлах: каталога, группы и записей. Для хранения групп и записей мы используем класс **TtdRecordStream** (в действительности мы будем использовать производный от него класс **TtdRecordFile**, ориентированный на использование файлов, но внутри программы мы будем считать, что применительно к расширяемой хеш-таблице этот класс является простым потоком). Каталог может храниться и извлекаться из любого класса, производного от **TStream**, но понятно, что длительного хранения целесообразно использовать класс **TFileStream**.

Извлечение и реализация каталога – следующая по сложности задача. Код интерфейса для ее выполнения приведен в листинге 7.20.

Листинг 7.20. Интерфейс класса **TtdHashDirectory**

```

type
  TtdHashDirectory = class
  private
    FCount : integer;
    FDepth : integer;
    FList : TList;
    FName : TtdNameString;
    FStream : TStream;
  protected
    function hdGetItem(aInx : integer) : longint;
    procedure hdSetItem(aInx : integer; aValue : longint);
    function hdErrorMsg(aErrorCode : integer;
                        const aMethodName : TtdNameString;
                        aIndex : integer) : string;
    procedure hdLoadFromStream;
    procedure hdStoreToStream;
  public
    constructor Create(aStream : TStream);
    destructor Destroy; override;
    procedure DoubleCount;
    property Count : integer read FCount;
    property Depth : integer read FDepth;
    property Items[aInx : integer] : longint
      read hdGetItem write hdSetItem; default;
    property Name : TtdNameString
      read FName write FName;
  end;
```

Для выполнения поставленной задачи этого общедоступного интерфейса вполне достаточно. Мы можем удвоить количество элементов в каталоге, используя метод **DoubleCount**, и можем получать текущие номера элементов (свойство **Count**) и разрядную глубину каталога (свойство **Depth**). Теоретически, мы могли бы обойтись только одним свойством, поскольку **Count = 2Depth**. Но поддержание обоих свойств – менее трудоемкая задача по сравнению с вычислением степени двух, когда это потребуется. И, наконец, мы можем обратиться к отдельным элементам, хранящимся в каталоге в виде значений типа длинных целых. Естественно, эти значения будут номерами групп.

Разделы **private** и **protected** содержат еще несколько методов и полей. Во-первых, это методы **set** и **get** свойства **Items**, а, во-вторых, – два метода, предназначенные для выполнения считывания и записи каталога в и из потока. Кроме того, как мы видим, реальным контейнером записей каталога является экземпляр **TList**.

В листинге 7.21 конструктор создает экземпляр каталога хеш-таблицы, внутренний объект **TList** и при необходимости выполняет автоматическое считывание из потока.

Листинг 7.21. Создание экземпляра класса **TtdHashDirectory**

```

constructor TtdHashDirectory.Create(aStream : TStream);
begin
    Assert(sizeof(pointer) = sizeof(longint),
    hdErrorMsg(tdePointerLongSize, 'Create', 0));
    {создать предка}
    inherited Create;
    {создать каталог как TList}
    FList := TList.Create;
    FStream := aStream;
    {если поток не содержит никаких данных, то инициализировать каталог с одной
    записью и глубиной равной 0}
    if (FStream.Size = 0) then begin
        FList.Count := 1;
        FCount := 1;
        FDepth := 0;
    end
    {в противном случае выполнить загрузку из потока}
    else
        hdLoadFromStream;
end;
procedure TtdHashDirectory.hdLoadFromStream;
begin
    FStream.Seek(0, soFromBeginning);
    FStream.ReadBuffer(FDepth, sizeof(FDepth));
    FStream.ReadBuffer(FCount, sizeof(FCount));
    FList.Count := FCount;
    FStream.ReadBuffer(FList.List^, FCount * sizeof(longint));
end;

```

Я оставил оператор **Assert** в конструкторе **Create**. Он проверяет равенство размера указателя размеру значения **longint**. Это связано с тем, что я немного "схитрил", сохраняя значения каталога непосредственно в **TList** в виде однотипных указателей. При изменении размера указателя или **longint**, используемый метод работать не будет. Поэтому, просто на всякий случай, я поместил здесь это утверждение. Если впоследствии компилятор будет генерировать сообщение об ошибке, это можно будет исправить. Если же нет, то во время выполнения будет выводиться сообщение о нарушении утверждения.

А пока что **LoadFromStream** выполняет минимальную проверку для проверки наличия допустимого каталога в потоке. Поскольку считывание выполняется непосредственно из потока в буфер фиксированного размера, в будущем, возможно, имеет смысл несколько усовершенствовать процесс, включив сигнатуру в поток или добавив проверку с применением циклического избыточного кода и т.п.

Уничтожение экземпляра каталога хеш-таблицы (листинг 7.22) требует считывания его текущего содержимого обратно в поток и освобождения внутреннего объекта **TList**.

Листинг 7.22. Уничтожение экземпляра класса **TtdHashDirectory**

```
destructor TtdHashDirectory.Destroy;
begin
    hdStoreToStream;
    FList.Free;
    inherited Destroy;
end;

procedure TtdHashDirectory.hdStoreToStream;
begin
    FStream.Seek(0, soFromBeginning);
    FStream.WriteBuffer(FDepth, sizeof(FDepth));
    FStream.WriteBuffer(FCount, sizeof(FCount));
    FStream.WriteBuffer(FList.List^, FCount * sizeof(longint));
end;
```

Методы предка (листинг 723) свойства **Items** просто извлекают данные, однотипные **longint**, из внутреннего объекта **TList**.

Листинг 7.23. Установка и извлечение значений каталога

```
function TtdHashDirectory.hdGetItem(aInx : integer) : longint;
begin
    Assert((0 <= aInx) and (aInx < FList.Count),
        hdErrorMsg(tdeIndexOutOfBounds, 'hdGetItem', aInx));
    Result := longint(FList.List^[aInx]);
end;
```

```

procedure TtdHashDirectory.hdSetItem(aInx : integer;
                                     aValue : longint );
begin
  Assert((0 <= aInx) and (aInx < FList.Count),
    hdErrorMsg(tdeIndexOutOfBounds, 'hdGetItem', aInx));
  FList.List^[aInx] := pointer(aValue);
end;

```

И, наконец, в листинге 7.24 приведен код интересного метода класса, который вдвое увеличивает размер каталога.

Листинг 7.24. Увеличение вдвое количества записей в каталоге

```

procedure TtdHashDirectory.DoubleCount;
var
  Inx : integer;
begin
  {удвоить значение счетчика, увеличить глубину}
  FList.Count := FList.Count * 2;
  FCount := FCount * 2;
  inc(FDepth);
  {теперь каждая запись в исходном каталоге удваивается; например, значению в
   записи 0 старого каталога теперь соответствует значение для записей 0 и 1
   нового каталога}
  for Inx := pred(FList.Count) downto 1 do
    FList.List^[Inx] := FList.List^[Inx div 2];
end;

```

Во-первых, этот метод удваивает значение счетчика элементов во внутреннем объекте **TList**. Реализация **TList** гарантирует установку новых элементов в нулевые значения, хотя, как вскоре будет показано, это не имеет никакого значения. Метод удваивает значение внутреннего счетчика и увеличивает значение разрядной глубины. Затем мы копируем и удваиваем все элементы объекта **TList** (чтобы убедиться, что цикл работает правильно, советуем во время изучения этого материала обращаться к рисункам 7.1 (e) и 7.1 (f)).

Этот класс выполняет ряд важных подготовительных действий, необходимых для работы нашего основного класса **TtdHashTableExtendible**, код интерфейса которого приведен в листинге 7.25.

Листинг 7.25. Интерфейс класса TtdHashTableExtendible

```

type
  TtdHashTableExtendible = class
    private
      FCompare : TtdCompareRecordKey;
      FCount : longint;
      FDirectory: TtdHashDirectory;
      FHashFunc : TtdHashFuncEx;
      FName : TtdNameString;

```

```

FBuckets : TtdRecordStream;
FRecords : TtdRecordStream;
FRecord : pointer;
protected
  procedure hteCreateNewHashTable;
  procedure hteError(aErrorCode : integer;
                     const aMethodName : TtdNameString);
  function hteErrorMsg(aErrorCode : integer;
                       const aMethodName : TtdNameString) : string;
  function hteFindBucket(const aKey : string;
                         var aFindInfo) : boolean;
  procedure hteSplitBucket(var aFindInfo);
public
  constructor Create(aHashFunc : TtdHashFuncEx;
                    aCompare : TtdCompareRecordKey;
                    aDirStream : TStream;
                    aBucketStream : TtdRecordStream;
                    aRecordStream : TtdRecordStream);
  destructor Destroy; override;
  function Find(const aKey : string;
                var aRecord) : boolean;
  procedure Insert(const aKey : string; var aRecord);
  property Count : longint read FCount;
  property Name : TtdNameString read FName write FName;
end;
```

Этот класс поддерживает обычные методы конструктора и деструктора, а также возможность вставки записи и ее ключа и последующего поиска записи по ее ключу.

Как показано в листинге 7.26, конструктору **Create** передаются три потока и два указателя на функции. Три потока предназначены для каталога, групп и записей. Первая функция – обычная функция хеширования (хотя для этой хеш-таблицы функции хеширования должны создавать 32-разрядные значения; в данном случае никакое деление по модулю на размер таблицы не выполняется). Вторая функция – функция сравнения значения ключа **Key** с записью, которая считывается из потока записей.

Листинг 7.26. Создание экземпляра класса **TtdHashTableExtendible**

```

constructor TtdHashTableExtendible.Create(
    aHashFunc : TtdHashFuncEx;
    aCompare : TtdCompareRecordKey;
    aDirStream : TStream;
    aBucketStream : TtdRecordStream;
    aRecordStream : TtdRecordStream);
```



```

begin
  {создать предка}
  inherited Create;
  {создать каталог}
  FDirectory := TtdHashDirectory.Create(aDirStream);
  {сохранить параметры}
  FHashFunc := aHashFunc;
  FCompare := aCompare;
  FBuckets := aBucketStream;
  FRecords := aRecordStream;
  {получить буфер для любой записи, которую нужно считать}
  GetMem(FRecord, FRecords.RecordLength);
  {если поток групп пуст, создать первую группу}
  if (FBuckets.Count = 0) then
    hteCreateNewHashTable;
end;
procedure TtdHashTableExtendible.hteCreateNewHashTable;
var
  NewBucket : TBucket;
begin
  FillChar(NewBucket, sizeof(NewBucket), 0);
  FDirectory[0] := FBuckets.Add(NewBucket);
end;

```

Конструктор создает каталог, передавая его потоку каталогов и сохраняя параметры во внутренних полях. Если поток групп еще не содержит групп, конструктор вызывает защищенный метод **hteCreateNewHashTable** для определения новой таблицы. Этот метод добавляет первую пустую группу в поток групп, и сохраняет номер группы в качестве первой записи каталога.

Деструктор просто выполняет очистку, как показано в листинге 7.27

Листинг 7.27. Уничтожение экземпляра класса TtdHashTableExtendible

```

destructor TtdHashTableExtendible.Destroy;
begin
  FDirectory.Free;
  if (FRecord<>nil) then
    FreeMem(FRecord, FRecords.RecordLength);
  inherited Destroy;
end;

```

Теперь рассмотрим метод **Find** и его вспомогательный защищенный метод **hteFindBucket**, который, как обычно и все вспомогательные подпрограммы, выполняет большую часть работы. Из листинга 7.28 видно, что метод **Find** действительно всего лишь вызывает метод **hteFindBucket**, и, если тот возвращает значение "истина", копирует запись из внутреннего буфера и, в свою очередь, возвращает значение "истина". Если метод возвращает значение "ложь", это свидетельствует, что запись не была найдена, и метод **Find** также возвращает значение "ложь".

Листинг 7.28. Поиск записи по ее ключу

```

type
  THashElement = packed record
    heHash : longint;
    heItem : longint;
  end;
  PBucket = ^TBucket;
  TBucket = packed record
    bkDepth : longint;
    bkCount : longint;
    bkHashes : array [0..pred(tdcBucketItemCount)] of THashElement;
  end;
  PFindItemInfo = ^TFindItemInfo;
  TFindItemInfo = packed record
    fiiHash : longint; {хеш-значение параметра ключа}
    fiiDirEntry : integer; {запись каталога}
    fiiSlot : integer; {ячейка в группе}
    fiiBucketNum : longint; {номер группы в потоке}
    fiiBucket : TBucket; {группа}
  end;

function TtdHashTableExtendible.Find(const aKey : string;
                                     var aRecord) : boolean;

var
  FindInfo : TFindItemInfo;
begin
  if hteFindBucket(aKey, FindInfo) then begin
    Result := true;
    Move(FRecord^, aRecord, FRecords.RecordLength);
  end
  else
    Result := false;
  end;

function TtdHashTableExtendible.hteFindBucket(const aKey : string;
                                              var aFindInfo) : boolean;

var
  FindInfo : PFindItemInfo;
  Inx : integer;
  IsDeleted : boolean;
begin
  FindInfo := PFindItemInfo(@aFindInfo);
  with FindInfo^ do begin
    {вычислить хеш-значение для строки}
    fiiHash := FHashFunc(aKey);
    {вычислить запись в каталоге для этого хеш-значения, которая соответствует
     номеру группы}
    fiiDirEntry := ReverseBits(fiiHash, FDirectory.Depth);
    fiiBucketNum := FDirectory[fiiDirEntry];
  end;

```

```

{извлечь группу}
FBuckets.Read(fiiBucketNum, fiiBucket, IsDeleted);
if IsDeleted then
    hteError(tdeHashTblDeletedBkt, 'hteFindBucket');
{выполнить поиск хеш-значения в группе, причем предполагается, что этот
поиск будет безуспешным}
Result := false;
with fiiBucket do begin
    for Inx := 0 to pred(bkCount) do begin
        {если хеш-значение совпадает..}
        if (bkHashes[Inx].heHash = fiiHash) then begin
            {считать запись}
            FRecords.Read(bkHashes[Inx].heItem, FRecord^, IsDeleted);
            if IsDeleted then
                hteError(tdeHashTblDeletedRec, 'hteFindBucket');
            {сравнить запись с ключом}
            if FCompare(FRecord^, aKey) then begin
                Result := true;
                fiiSlot := Inx;
                Exit;
            end;
        end;
    end;
end;
end;
end;

```

Метод **hteFindBucket** представляет наибольший интерес. Вначале, подобно "обычной" хеш-таблице, он вычисляет хеш-значение ключа. Затем он вычисляет запись каталога, к которой это хеш-значение относится. Как упоминалось ранее, для этого необходимо инвертировать соответствующее количество младших разрядов. Требуемое количество разрядов равно разрядной глубине каталога и эту задачу выполняет небольшая подпрограмма **ReverseBits**.

Листинг 7.29. Вычисление записи каталога

```

function ReverseBits(aValue : longint; aBitCount : integer) : longint;
var
    i : integer;
begin
    Result := 0;
    for i := 0 to pred(aBitCount) do begin
        Result := (Result shl 1) or (aValue and 1);
        aValue := aValue shr 1;
    end;
end;

```

Как только запись каталога определена, можно выполнить ее считывание, чтобы получить номер группы. Сразу после этого можно реализовать считывание группы из потока групп. А затем выполняется поиск среди хеш-значений группы с целью нахождения хеш-значения, соответствующего данному ключу. Если это значение будет найдено, мы получим номер требуемой записи и сможем выполнить ее считывание из потока записей.

Как уже говорилось ранее, методы **Insert** и **Find** не делают никаких предположений о порядке следования хеш-номеров в группах. Поэтому мы используем последовательный поиск. Выполнив небольшой объем дополнительной работы, можно было бы обеспечить сортировку элементов в группе в порядке следования хеш-значений и, в результате, иметь возможность воспользоваться бинарным поиском.

Однако это сопряжено с одной проблемой: ничто не мешает функции хеширования сгенерировать одно и то же хеш-значение для двух и более ключей. В этом случае они были бы добавлены в одну группу, и, следовательно, чтобы обнаружить требуемую запись, нам пришлось бы гарантировать посещение всех записей, имеющих одинаковые хеш-значения.

Если запись была найдена, метод **hteFindBucket** возвращает хеш-значение, запись каталога, номер группы, саму группу и ячейку в группе, в которой было найдено хеш-значение. В настоящее время вся эта информация не используется. Последующая версия класса **TtdHashTableExtendible** будет поддерживать удаление, и эта дополнительная информация понадобится.

Если запись не была найдена, метод возвращает все ранее перечисленные данные, за исключением номера ячейки. Использование этих данных показано на примере метода **Insert**, код которого показан в листинге 7.30.

Листинг 7.30. Вставка пары ключ/запись в хеш-таблицу

```
procedure TtdHashTableExtendible.Insert(const aKey : string;
                                       var aRecord);

var
  FindInfo : TFindItemInfo;
  RRN : longint;
begin
  if hteFindBucket(aKey, FindInfo) then
    hteError(tdeHashTblKeyExists, 'Insert');
  { выполнить проверку для выяснения наличия достаточного места в данной
    группе; если нет, разбить группу и повторить процесс поиска места для
    вставки элемента; процесс продолжается до момента обнаружения достаточного
    свободного места }
  while (FindInfo.fiiBucket.bkCount >= tdcBucketItemCount) do begin
    hteSplitBucket(FindInfo);
    if hteFindBucket(aKey, FindInfo) then
      hteError(tdeHashTblKeyExists, 'Insert');
  end;
```

```

{добавить запись в поток записей для получения номера записи}
RRN := FRecords.Add(aRecord);
{добавить хеш-значения в конец списка, обновить группу}
with FindInfo, FindInfo.fiiBucket do begin
    bkHashes[bkCount].heHash := fiiHash;
    bkHashes[bkCount].heItem := RRN;
    inc(bkCount);
    FBuckets.Write(fiiBucketNum, fiiBucket);
end;
{имеется еще одна запись}
inc(FCount);
end;

```

При вставке, прежде всего, следует попытаться найти пару ключ/запись. Если это удастся, генерируется ошибка. Если же нет – метод **hteFindBucket** вернет разнообразную информацию: хеш-значение ключа (чтобы его не нужно было повторно вычислять), запись каталога, номер группы и саму группу, в которой должно находиться хеш-значение ключа.

Мы проверяем, заполнена ли группа. Пока предположим, что это не так. Мы добавляем запись в поток записей – что позволяет получить номер записи, – а затем добавляем пару хеш-значение/номер записи в конец группы, увеличивая значения обычно применяемых в таких случаях счетчиков.

Если группа заполнена, ее нужно разбить. Это делает другой скрытый защищенный метод **hteSplitBucket**. Как только он осуществляет возврат, необходимо повторить попытку поиска элемента, чтобы определить необходимую информацию таким образом, чтобы можно было легко добавить пару ключ/запись. Хотя и был включен код проверки на предмет обнаружения пары ключ/запись и генерирования ошибки в случае возникновения упомянутой ситуации, хеш-таблица действительно тщательно очищена – как мы уже убедились, подобная ситуация не возникает.

Итак, рассмотрим последний метод – **hteSplitBucket**. На данный момент он является наиболее сложным методом класса. Листинг 7.31 содержит подробные комментарии, но чтобы работа метода была понятнее, рекомендуем снова обратиться к рисунку 7.1.

Листинг 7.31. Разбиение группы

```

procedure TtdHashTableExtendible.hteSplitBucket(var aFindInfo);
var
    FindInfo : PFindItemInfo;
    Inx : integer;
    NewBucket : TBucket;
    Mask : longint;
    OldValue : longint;
    OldInx : integer;
    NewInx : integer;

```

```

NewBucketNum : longint;
StartDirEntry : longint;
NewStartDirEntry : longint;
EndDirEntry : longint;
begin
  FindInfo := PFindItemInfo(@aFindInfo);
  {если разбиваемая группа имеет такую же разрядную глубину, как каталог,
   удвоить емкость каталога}
  if (FindInfo^.fiiBucket.bkDepth = FDirectory.Depth) then begin
    FDirectory.DoubleCount;
    {обновить элемент каталога новой группой, которая была разбита}
    FindInfo^.fiiDirEntry := FindInfo^.fiiDirEntry * 2;
  end;
  {вычислить диапазон записей каталога, указывающих на исходную группу, и
   диапазон для новой группы}
  StartDirEntry := FindInfo^.fiiDirEntry;
  while (StartDirEntry >= 0) and
    (FDirectory[StartDirEntry] = FindInfo^.fiiBucketNum) do
    dec(StartDirEntry);
  inc(StartDirEntry);
  EndDirEntry := FindInfo^.fiiDirEntry;
  while (EndDirEntry < FDirectory.Count) and
    (FDirectory[EndDirEntry] = FindInfo^.fiiBucketNum) do
    inc(EndDirEntry);
  dec(EndDirEntry);
  NewStartDirEntry := (StartDirEntry + EndDirEntry + 1) div 2;
  {увеличить разрядную глубину разбиваемой группы}
  inc(FindInfo^.fiiBucket.bkDepth);
  {инициализировать новую группу; она будет иметь такую же разрядную глубину,
   как и разбиваемая группа}
  FillChar(NewBucket, sizeof(NewBucket), 0);
  NewBucket.bkDepth := FindInfo^.fiiBucket.bkDepth;
  {вычислить маску AND, которая будет использоваться для выяснения места
   помещения хеш-записей}
  Mask := (1 shl NewBucket.bkDepth) - 1;
  {вычислить значение, полученное в результате применения маски AND,
   для хеш-записей старой группы}
  OldValue := ReverseBits(StartDirEntry, FDirectory.Depth) and Mask;
  {считать старую группу и перенести в новую группу принадлежащие ей
   хеш-значения}
  OldInx := 0;
  NewInx := 0;
  with FindInfo^.fiiBucket do
    for Inx := 0 to pred(bkCount) do begin
      if (bkHashes[Inx].heHash and Mask) = OldValue then begin
        bkHashes[OldInx] := bkHashes[Inx];
        inc(OldInx);
      end

```



```

else begin
    NewBucket.bkHashes[NewInx] := bkHashes[Inx];
    inc(NewInx);
end;
end;
{установить счетчики для обеих групп}
FindInfo^.fiiBucket.bkCount := OldInx;
NewBucket.bkCount := NewInx;
{добавить новую группу в поток групп, обновить старую группу}
NewBucketNum := FBuckets.Add(NewBucket);
FBuckets.Write(FindInfo^.fiiBucketNum, FindInfo^.fiiBucket);
{установить все записи в новом диапазоне каталога в соответствие с новой группой}
for Inx := NewStartDirEntry to EndDirEntry do
    FDirectory[Inx] := NewBucketNum;
end;

```

Прежде всего, выполняется проверка, равна ли разрядная глубина разбиваемой группы разрядной глубине каталога. Если да, то необходимо вдвое увеличить размер каталога и обеспечить обновление отслеживаемого значения записи каталога. Например, если запись **FindInfo^.fiiDirEntry** имела значение, равное 3, и мы вдвое увеличили размер каталога, то теперь она должна иметь значение, равное 6 (или, если быть точным, 7, поскольку обе новые записи каталога указывают на одну и ту же группу).

Теперь нужно выяснить диапазон записей каталога, которые указывают на разбиваемую группу. В соответствии с рисунком 7.1 (g), если бы пришлось разбивать запись *B*, диапазон был бы 4–7. Разбиваемая группа должна остаться в первой половине этого диапазона, а новая группа, которую предстоит заполнить, будет занимать вторую половину диапазона записей каталога.

Поскольку мы разбиваем группу, ее разрядную глубину следует увеличить (мы уже обеспечили, чтобы это можно сделать без превышения разрядной глубины каталога). Поскольку новая группа дополняет данную, она будет иметь такую же разрядную глубину.

Теперь элементы в заполненной группе потребуется разделить между ней и новой группой. Если бы для этого мы пошли окольным путем, то скопировали бы элементы во временный массив, очистили заполненную группу, обновили записи каталога, а затем снова добавили элементы в группу. При этом для каждого элемента пришлось бы получать хеш-значение и вычислять инвертированные разряды для определения записи каталога, чтобы можно было определить, в какую группу должен быть добавлен тот или иной элемент. Этот метод работает очень надежно, но, как уже было сказано, является слишком трудоемким.

Желательно разработать метод, с помощью которого можно было бы непосредственно определить, в какую группу должно помещаться хеш-значение. Предположим, что имеет место следующая ситуация: разрядная глубина каталога равна 3,

но разрядная глубина группы равна 2. Записи 4 и 5 каталога указывают на группу А, которая заполнена, а записи 6 и 7 – на пустую группу В. Куда должно быть помещено данное хеш-значение? Прежде всего, следует осознать, что группа А содержит только те хеш-значения, последними разрядами которых являются 001, 101, 011 или 111 (чтобы убедиться в этом, проинвертируйте разряды для получения записей 4, 5, 6 и 7 каталога). Если хеш-значение имеет окончание 001 или 101, оно будет помещено в группу А. Если оно имеет окончание 011 или 111, оно будет помещено в группу В. Все еще не понятно, не так ли? Что ж, первые две комбинации заканчиваются разрядами 01, в то время как две вторые комбинации – разрядами 11. Почему учитываются только два разряда? Вспомним, что разрядная глубина группы равна 2. Идея состоит в том, чтобы вычислить запись каталога, соответствующую началу диапазона (которое нам известно), проинвертировать разряды, соответствующие разрядной глубине каталога, и выполнить операцию AND для полученного результата и маски, которая была сгенерирована из значения разрядной глубины группы. Затем эту маску можно использовать для разбиения хеш-значений по категориям. Именно эти действия и выполняются в среднем разделе подпрограммы.

Все остальные действия являются тривиальными и особого интереса не представляют – выполняется проверка правильности значений счетчиков групп, добавление новой группы, обновление исходной группы и обеспечение того, чтобы записи каталога, которые требуют изменения, указывали на новую группу.

Полный код класса **TtdHashTableExtendible** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHshExt.pas**.

Резюме

В этой главе были рассмотрены хеш-таблицы – структуры данных, которые пытаются предоставить максимально быстрый доступ к своим элементам, при этом они подпадают под категорию $O(1)$.

Мы рассмотрели различные хранящиеся в памяти таблицы, включая две наиболее важных – хеш-таблицу, использующую линейное зондирование, и хеш-таблицу, в которой применяется связывание. Мы ознакомились с преимуществами и недостатками каждого из этих методов и со способами их настройки.

И, наконец, мы выяснили, как поддерживать хеш-таблицы на диске, минимизируя при этом количество обращений к диску. Мы рассмотрели алгоритм группирования и его реализацию для создания базы данных, в основе которой лежит хеширование.



Глава 8

Бинарные деревья

Подобно массивам и связным спискам, деревья того или иного вида -- это структуры данных, которые используются программистами практически повсеместно. В главе 3 были рассмотрены односвязные списки, в которых существовала единственная связь, соединяющая узлы друг с другом (двухсвязные списки имели также связь, указывающую в противоположном направлении). Обычно связные списки рассматриваются как горизонтальные структуры (в целях экономии места на бумаге!), в которых начальный узел располагается слева, а сам связный список простирается направо. Теперь представим, что этот связный список повернут на 90 градусов по часовой стрелке, чтобы начальный узел располагался вверху, а конечный внизу. Этот случай представляет собой особый пример многопутевого дерева, в котором каждый узел имеет только один *дочерний* узел, расположенный непосредственно под ним. Аналогично, каждый узел имеет один *родительский* узел, который расположен непосредственно над ним. Естественно, такая классификация охватывает целое семейство деревьев. Примем соглашение, что самый нижний узел имеет нулевую связь, т.е. не имеет дочернего узла. Поскольку каждый узел имеет максимум один дочерний узел, односвязный список можно было бы называть унарным деревом.

Многопутевое дерево является обобщением этой концепции. Оно представляет собой коллекцию узлов, организованных так, чтобы все узлы кроме корневого (мы будем называть узел в верхушке дерева *корневым*, а узел, который не имеет дочерних узлов -- *листовым*, или просто *листом*) имели только *один* родительский узел и могли иметь ноль или больше дочерних узлов. Таким образом, связный список -- это особое многопутевое дерево, в котором каждый узел (кроме самого ниж-

него) имеет только один дочерний узел. Если каждый узел может иметь максимум n дочерних узлов, такое дерево называется n -арным деревом.

Теперь рассмотрим случай, когда каждый узел имеет до двух дочерних узлов. Иначе говоря, для каждого узла суще-

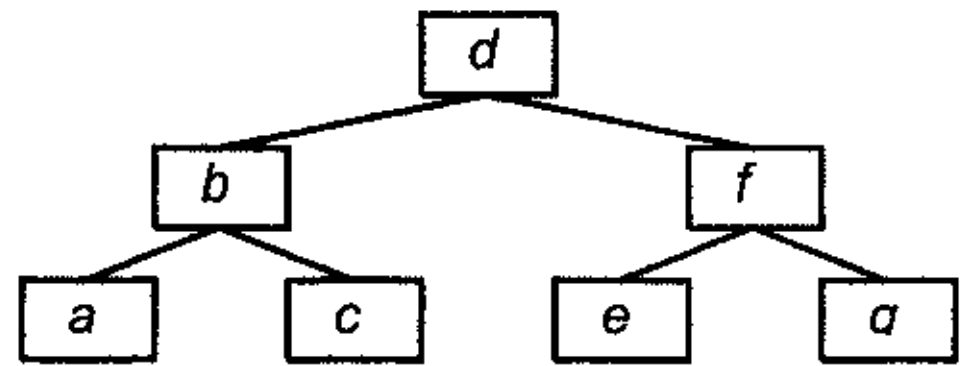


Рисунок 8.1. Бинарное дерево

ствует максимум две связи с узлами следующего нижнего уровня. Эта структура называется *бинарным деревом*. Согласно принятому соглашению, два дочерних узла данного узла называются левым и правым дочерними узлами, поскольку при рисовании дерева с расположением его корня в вершине, дочерние узлы выстраиваются горизонтально под ним, один левее от другого. Классическое представление бинарного дерева показано на рис. 8.1.

Из приведенных рассуждений ясно, что при определении используемого узла бинарного дерева в Delphi-программе нам требуются две связи (т.е. указатели) с его дочерними узлами, связь с его родительским узлом (эта связь необязательна, но, как мы увидим, ее применение упрощает некоторые алгоритмы, работающие с деревьями) и фактические данные, которые должны храниться в узле. С целью упрощения задачи примем, что данные в узле могут быть представлены указателем, подобно **TList** и структурам данных, которые уже были рассмотрены в этой книге. Поскольку узел имеет фиксированный размер, мы снова воспользуемся описанным в главе 3 диспетчером узлов, когда дело дойдет до создания класса бинарного дерева. Код, приведенный в листинге 8.1, определяет расположение записей узлов.

Листинг 8.1. Расположение узлов в бинарном дереве

```

type
  TtdChildType = (
    ctLeft,    {.. левый дочерний узел}
    ctRight);  {.. правый дочерний узел}
  TtdRBColor = ( {цвета для красно-черного дерева}
    rbBlack,   {.. черный}
    rbRed);    {.. красный}
  PtdBinTreeNode = ^TtdBinTreeNode;
  TtdBinTreeNode = packed record
    btParent : PtdBinTreeNode;
    btChild : array [TtdChildType] of PtdBinTreeNode;
    btData : pointer;
  case boolean of
    false : (btExtra : longint);
    true : (btColor : TtdRBColor);
  end;
end;
```

Обратите внимание, что две дочерние связи мы определили в виде двухэлементного массива. На первый взгляд, это может показаться излишним, но когда дело дойдет до реализации операций с бинарным деревом, такое определение существенно упростит нашу задачу. Кроме того, узел бинарного дерева объявляет дополнительное поле, которое не требуется для обычных бинарных деревьев, однако упрощает задачу для красно-черного варианта дерева бинарного поиска.

Создание бинарного дерева

Само по себе создание бинарного дерева тривиально. В простейшем случае корневой узел бинарного дерева определяет все бинарное дерево.

```
var  
  MyBinaryTree : PtBinTreeNode;
```

Если **MyBinaryTree** равен **nil**, никакого бинарного дерева не существует, поэтому это значение служит начальным значением бинарного дерева.

```
{инициализировать бинарное дерево}  
MyBinaryTree := nil;
```

На практике принято использовать фиктивный узел, аналогичный фиктивному заглавному узлу односвязного списка, чтобы каждый реальный узел дерева, включая корневой, имел родительский узел. Корневой узел может быть как левым, так и правым дочерним узлом фиктивного узла, но для определенности примем, что он является левым.

Вставка и удаление с использованием бинарного дерева

Если мы всерьез намереваемся использовать бинарное дерево, необходимо рассмотреть, как выполняется добавление в дерево элементов (т.е. узлов), удаление элементов из дерева и посещение всех элементов дерева. Последняя операция позволит выполнять поиск конкретного элемента. Поскольку выполнение последних двух операций невозможно без рассмотрения первой, начнем с рассмотрения вставки узла в бинарное дерево.

Чтобы иметь возможность вставить узел в бинарное дерево, необходимо выбрать родительский узел, к которому можно присоединить новый узел в качестве дочернего, и более того, этот узел не может уже иметь два дочерних узла. Мы должны также знать, каким дочерним узлом – левым или правым – должен стать новый узел.

При заданном родительском узле и указании дочерних узлов слева направо код для вставки узла очень прост. Мы создаем узел, устанавливаем в качестве значения его поля данных элемент, который добавляем в дерево, и определяем обе его дочерние связи как **nil**. Затем, во многом подобно вставке узла в двусвязный список, мы устанавливаем соответствующий дочерний указатель родительского узла так, чтобы он указывал на новый дочерний узел, а родительский указатель дочернего узла – на родительский узел.

Листинг 8.2. Вставка в бинарное дерево

```

function TtdBinaryTree.InsertAt(aParentNode : PtdBinTreeNode;
                                aChildType : TtdChildType;
                                aItem : pointer) : PtdBinTreeNode;
begin
    {если родительский узел является нулевым, считаем, что выполняется вставка
     корневого узла}
    if (aParentNode = nil) then begin
        aParentNode := FHead;
        aChildType := ctLeft;
    end;
    {выполнить проверку того, установлена ли уже дочерняя связь}
    if (aParentNode^.btChild[aChildType] <> nil) then
        btError(tdeBinTreeHasChild, 'InsertAt');
    {распределить новый узел и вставить в качестве требуемого дочернего узла
     родительского узла}
    Result := BTNodeManager.AllocNode;
    Result^.btParent := aParentNode;
    Result^.btChild[ctLeft] := nil;
    Result^.btChild[ctRight] := nil;
    Result^.btData := aItem;
    Result^.btExtra := 0;
    aParentNode^.btChild[aChildType] := Result;
    inc(FCount);
end;

```

Обратите внимание, что приведенный в листинге 8.2 код вначале проверяет, является ли добавляемый узел корневым. Если да, то переданный родительский узел равен **nil**. В этом случае метод инициализирует родительский узел значением внутреннего заглавного узла.

Кроме этой проверки метод **InsertAt** убеждается, что дочерняя связь, которую предполагается использовать для нового узла, действительно не используется. В противном случае это будет грубой ошибкой.

Обратите внимание, что класс бинарного дерева (составной частью которого является этот метод) использует диспетчер узлов для распределения и освобождения узлов. Поскольку все узлы имеют одинаковый размер, в этом, как было сказано в главе 3, заложен глубокий смысл.

А как выполняется удаление узлов? Эта задача несколько сложнее, поскольку узел может иметь один или два дочерних узла. Первое правило удаления может быть сформулировано следующим образом: листовой узел (т.е. не имеющий дочерних узлов) может быть удален без каких-либо нежелательных последствий. При этом мы выясняем, каким дочерним узлом родительского узла является лист, и устанавливаем соответствующую дочернюю связь равной **nil**. После этого узел может быть освобожден.

Второе правило удаления из бинарного дерева применяется в отношении случая, когда удаляемый узел имеет один дочерний узел. Эта задача также достаточно проста: мы просто перемещаем дочерний узел вверх по дереву, чтобы он стал тем же дочерним узлом родительского узла, каким является удаляемый узел.

Третье правило применяется к случаю, когда удаляемый узел имеет два дочерних узла. Как и можно было предположить, это правило звучит просто: узел не может быть удален. Попытка сделать это является ошибкой. Позже мы рассмотрим вариант бинарного дерева – дерево бинарного поиска, – который содержит достаточный объем дополнительной внедренной в дерево информации, чтобы можно было обойти это ограничение.

Листинг 8.3. Удаление из бинарного дерева

```

procedure TtdBinaryTree.Delete(aNode : PtdBinTreeNode);
var
    OurChildsType : TtdChildType;
    OurType : TtdChildType;
begin
    if (aNode = nil) then
        Exit;
    { выяснить, имеется ли единственный дочерний узел, и то, каким узлом он
      является; при наличии двух дочерних узлов сгенерировать ошибку }
    if (aNode^.btChild[ctLeft] <> nil) then begin
        if (aNode^.btChild[ctRight] <> nil) then
            btError(tdeBinTree2Children, 'Delete');
        OurChildsType := ctLeft;
    end
    else
        OurChildsType := ctRight;
    { выяснить, является ли дочерний узел левым или правым дочерним узлом данного
      родительского узла }
    OurType := GetChildType(aNode);
    { установить дочернюю связь данного родительского узла равной данной дочерней
      связи }
    aNode^.btParent^.btChild[OurType] :=
        aNode^.btChild[OurChildsType];
    if (aNode^.btChild[OurChildsType] <> nil) then
        aNode^.btChild[OurChildsType]^.btParent := aNode^.btParent;
    { освободить узел }
    if Assigned(FDispose) then
        FDispose(aNode^.btData);
    BTNodeManager.FreeNode(aNode);
    dec(FCount);
end;

```

В листинге 8.3 не учтен случай, когда удаляемый узел является нулевым. В любом случае в этой ситуации мало что можно сделать, а генерация исключения была бы излишней. Поэтому метод проверяет, чтобы удаляемый узел не имел двух

дочерних узлов. Однако он не разделяет два других случая удаления (т.е. случаи отсутствия дочерних узлов и наличия только одного дочернего узла), а объединяет их в один случай, когда один дочерний узел замещает узел, даже если дочерний узел является нулевым. **GetChildType** – это небольшая функция, которая возвращает информацию о том, является ли ее параметр узла левым или правым дочерним узлом родительского узла.

Перемещение по бинарному дереву

После того, как мы рассмотрели построение бинарного дерева, можно рассмотреть вопрос о том, как посетить все узлы такой структуры. Под *посещением* подразумевается выполнение той или иной обработки хранящегося в узле элемента. Такой обработкой могло бы быть как выполнение простой операции, подобной записи данных в узел, так и реализация более сложных действий.

В отличие от связанных списков, где перемещение по структуре определено однозначно (достаточно следовать всем указателям **Next** (следующий), пока не будет достигнут конец списка), в бинарном дереве в каждом узле можно выбрать один из двух путей, и поэтому процесс несколько усложняется. Процедуру перемещения по дереву называют *обходом* (*traversal*). Существуют четыре основных алгоритма обхода – *обходом в ширину* (*pre-order*), *симметричным обходом* (*in-order*), *обходом в глубину* (*post-order*) и *обходом по уровням* (*level-order*). Последний алгоритм – обход по уровням – наиболее прост для визуального представления, но наиболее сложен для кодирования. Этот алгоритм предполагает посещение каждого из узлов, начиная с корневого, и просмотр узлов сверху вниз, уровень за уровнем. На каждом уровне мы посещаем узлы слева направо. Таким образом, мы посещаем корневой узел, левый дочерний узел корневого узла, правый дочерний узел корневого узла, левый дочерний узел левого дочернего узла корневого узла, правый дочерний узел левого дочернего узла корневого узла и т.д. Снова обратившись к рисунку 8.1, мы видим, что при обходе по уровням посещение узлов выполнялось бы в следующем порядке: *d, b, f, a, c, e, g*.

Обход в ширину, симметричный обход и обход в глубину

Прежде чем приступить к описанию остальных трех алгоритмов обхода, которые взаимосвязаны, приведем несколько иное определение бинарного дерева. Бинарное дерево состоит из корневого узла, содержащего указатели на корневые узлы двух других бинарных деревьев, называемых дочерними. Указатели на любой или оба дочерних узла могут быть нулевыми. Это определение описывает бинарное дерево очень кратко, хотя и рекурсивно. Тем не менее, оно представляет собой идеальный способ описания остальных трех видов обхода.

При обходе в ширину вначале мы посещаем корневой узел, затем, используя алгоритм обхода в ширину, выполняем обход левого дочернего дерева, а затем таким же образом выполняем обход правого дочернего дерева. (Обход дерева,

изображенного на рис. 8.1, выполнялся бы в следующем порядке: d, b, a, c, f, e, g .) При симметричном обходе вначале выполняется обход левого дочернего дерева корневого узла с применением алгоритма симметричного обхода, а затем симметричный обход правого дочернего дерева. (В дереве, показанном на рис. 8.1, посещение узлов выполнялось бы в следующем порядке: a, b, c, d, e, f, g .) При обходе в глубину вначале выполняется обход в левого дочернего дерева с применением алгоритма обхода в глубину, затем таким же образом выполняется обход правого дочернего дерева, а затем посещается корневой узел. (В дереве, изображенном на рис. 8.1, посещение узлов выполнялось бы в следующем порядке: a, c, b, e, g, f, d .)

Обход в глубину чаще всего применяется для уничтожения всех узлов в бинарном дереве, когда процесс уничтожения можно было бы описать следующим образом: "чтобы уничтожить все узлы в бинарном дереве, необходимо уничтожить левое дочернее дерево корневого узла, затем правое дочернее дерево корневого узла, а затем сам коревой узел".

Создание кода, реализующего эти три алгоритма обхода, не представляет собой сложности: достаточно создать рекурсивную процедуру, которая вызывает сама себя для каждого узла. Пример простого кода выполнения рекурсивных обходов приведен в листинге 8.4.

Листинг 8.4. Обход в ширину, симметричный обход и обход в глубину

```

type
  TtdProcessNode = procedure (aNode : PtdBinaryNode);
  procedure PreOrderTraverse(aRoot : PtdBinaryNode;
                           aProcessNode : TtdProcessNode);
begin
  if (aNode<>nil) then begin
    aProcessNode(aRoot);
    PreOrderTraverse(aRoot^.bnChild[ciLeft], aProcessNode);
    PreOrderTraverse(aRoot^.bnChild[ciRight], aProcessNode);
  end;
end;

procedure InOrderTraverse(aRoot : PtdBinaryNode;
                        aProcessNode : TtdProcessNode);
begin
  if (aNode<>nil) then begin
    InOrderTraverse(aRoot^.bnChild[ciLeft], aProcessNode);
    aProcessNode(aRoot);
    InOrderTraverse(aRoot^.bnChild[ciRight], aProcessNode);
  end;
end;

```

```

procedure PostOrderTraverse(aRoot : PtdBinaryNode;
                           aProcessNode : TtdProcessNode);
begin
  if (aNode<>nil) then begin
    PostOrderTraverse(aRoot^.bnChild[ciLeft], aProcessNode);
    PostOrderTraverse(aRoot^.bnChild[ciRight], aProcessNode);
    aProcessNode(aRoot);
  end;
end;

```

Обратите внимание на то, как каждая рекурсивная процедура проверяет, не является ли переданный ей узел нулевым. В этом случае она не выполняет никаких действий, немедленно осуществляя выход. Следовательно, со временем рекурсивный вызов процедур завершится (поскольку дерево простирается не до бесконечности).

Однако в каждом случае применения рекурсивной процедуры следует оценить, сколько раз она должна будет выполняться в ходе последовательности рекурсивных вызовов. Дело в том, что рекурсивные процедуры хранят свое состояние в стеке программы, размер которого в общем случае ограничен. Если выясняется, что рекурсивная процедура может иметь слишком много уровней, следует подумать над тем, как избавиться от рекурсии за счет применения внешнего стека. Используя внешний стек вместо стека программы, можно быть уверенным, что при необходимости размер стека в куче можно будет увеличить (пока выделенный объем кучи не будет исчерпан, однако, в общем случае этот объем значительно превышает размер стека программы).

Мы используем стек, созданный на основе связанного списка класса **TtdStack**, который был описан в главе 3. Для выполнения обхода в ширину мы заталкиваем в стек корневой узел и выполняем цикл, который продолжается до тех пор, пока стек не опустеет. Мы выталкиваем из стека верхний узел и посещаем его. Если правая дочерняя связь этого узла является ненулевой, мы заталкиваем ее в стек. Затем заталкиваем в стек левую дочернюю связь узла, если она является ненулевой. (Заталкивание дочерних узлов в указанном порядке означает, что вначале из стека выталкивается левый дочерний узел.) Если стек не является пустым, цикл повторяется. Обход завершается немедленно после опустошения стека.

Листинг 8.5. Нерекурсивный обход в ширину

```

type
  TtdVisitProc = procedure ( aData : pointer; aExtraData : pointer;
                             var aStopVisits : boolean );
function TtdBinaryTree.btNoRecPreOrder(aAction : TtdVisitProc;
                                       aExtraData : pointer) : PtdBinTreeNode;
var
  Stack : TtdStack;
  Node : PtdBinTreeNode;
  StopNow : boolean;

```

```

begin
  {предположим, что мы не добрались до выбранного узла}
  Result := nil;
  StopNow := false;
  {создать стек}
  Stack := TtdStack.Create(nil);
  try
    {затолкнуть корневой узел}
    Stack.Push(FHead^.btChild[ctLeft]);
    {продолжать процесс до тех пор, пока стек не будет пуст}
    while not Stack.IsEmpty do begin
      {извлечь узел в начале очереди}
      Node := Stack.Pop;
      {выполнить с ним указанное действие; если в результате возвращаемое
       значение переменной StopNow равно true, вернуть этот узел}
      aAction(Node^.btData, aExtraData, StopNow);
      if StopNow then begin
        Result := Node;
        Stack.Clear;
      end
      {в противном случае продолжить цикл}
      else begin
        {затолкнуть правую дочернюю связь, если она не нулевая}
        if (Node^.btChild[ctRight] <> nil) then
          Stack.Push(Node^.btChild[ctRight]);
        {затолкнуть левую дочернюю связь, если она не нулевая}
        if (Node^.btChild[ctLeft] <> nil) then
          Stack.Push(Node^.btChild[ctLeft]);
      end;
    end;
  finally
    {уничтожить стек}
    Stack.Free;
  end;
end;

```

Касательно кода, приведенного в листинге 8.5, следует сделать несколько замечаний. Во-первых, мы используем процедуру действия, которая несколько сложнее применявшейся ранее. Процедура типа **TtdVisitProc** предоставляет пользователю метода обхода большую степень управления процессом, а именно — возможность остановить обход. Т.е. пользователь класса бинарного дерева может выполнять действия как для каждой записи (посещая все узлы), так и для первой найденной записи (т.е. для поиска первого узла, удовлетворяющего заданному условию). Значение третьего параметра процедуры действия, **aStopVisits**, устанавливается равным **false** вызывающей процедурой, а если процедуре действия нужно остановить обход, это значение может быть установлено равным **true** (в этом

случае метод обхода вернет элемент, который привел к возврату значения **true** процедурой действия).

Однако, важная особенность приведенного в листинге 8.5 кода состоит в том, что процедура считает дерево не пустым. Фактически эта процедура – внутренняя процедура класса бинарного дерева, возможного при определенных условиях, и она будет вызываться только для дерева, которое содержит, по меньшей мере, один узел.

Убедившись, насколько просто избавиться от рекурсии при обходе в ширину, можно было бы предположить, что это легко сделать и для остальных двух видов обхода. Однако, применяя это же подход к симметричному обходу и обходу в глубину, мы сталкиваемся с препятствием. Чтобы понять, о чем идет речь, рассмотрим исключение рекурсии для симметричного обхода тем же способом, который был применен для обхода в ширину. Теоретически в цикле нужно было бы затолкнуть в стек правый дочерний узел, затем сам узел, а затем левый дочерний узел. Далее, со временем, нужно было бы вытолкнуть узел из стека и выполнить его обработку. Но, вытолкнув узел из стека, как узнать, встречался ли он ранее? Если узел ранее встречался, его нужно посетить; если нет, его вместе с дочерними узлами необходимо затолкнуть в стек, но в правильном порядке.

В действительности нужно сделать следующие действия. Вытолкнуть узел из стека. Если ранее узел не встречался, нужно затолкнуть в стек правый дочерний узел, пометить узел как "встречавшийся", затолкнуть его, а затем затолкнуть в стек левый дочерний узел. Если ранее узел *встречался* (помните, что он уже помечен?), следует просто его обработать. Но как пометить узел? В конце концов, узел – это указатель, и в действительности не хотелось бы с ним возиться. Я предлагаю следующее решение: после заталкивания в стек "встречавшегося" узла нужно затолкнуть узел **n11**. В этом случае выталкивание из стека нулевого узла свидетельствует о том, что следующий узел в стеке является тем, который должен быть обработан.

Нерекурсивный алгоритм симметричного обхода работает следующим образом. Затолкните в стек корневой узел и войдите в цикл, который должен выполняться до момента опустошения стека. Вытолкните верхний узел из стека. Если он является нулевым, вытолкните из стека следующий узел и посетите его. Если вытолкнутый узел не является нулевым, затолкните в стек правый дочерний узел (если он является ненулевым), затем сам узел, затем затолкните нулевой указатель и в заключение затолкните в стек левый дочерний узел (если он является ненулевым). Снова выполните цикл.

Как и в случае обхода в ширину, метод предполагает, что дерево является не пустым, и что в нем присутствует, по меньшей мере, один узел. В данном случае это еще более важно, поскольку метод может работать совершенно не правильно, если нулевой узел заталкивается в стек, который не связан с алгоритмом.

Листинг 8.6. Нерекурсивный симметричный обход

```

function TtdBinaryTree.btNoRecInOrder(aAction : TtdVisitProc;
                                     aExtraData : pointer) : PtdBinTreeNode;
var
  Stack : TtdStack;
  Node : PtdBinTreeNode;
  StopNow : boolean;
begin
  {предположим, что мы не добрались до выбранного узла}
  Result := nil;
  StopNow := false;
  {создать стек}
  Stack := TtdStack.Create(nil);
  try
    {затолкнуть корневой узел}
    Stack.Push(FHead^.btChild[ctLeft]);
    {продолжать процесс до тех пор, пока стек не опустеет}
    while not Stack.IsEmpty do begin
      {извлечь узел в начале очереди}
      Node := Stack.Pop;
      {если он является нулевым, вытолкнуть из стека следующий узел и выполнить
      с ним указанное действие. Если в результате возвращается запрос на пре-
      кращение обхода, вернуть этот узел}
      if (Node = nil) then begin
        Node := Stack.Pop;
        aAction(Node^.btData, aExtraData, StopNow);
        if StopNow then begin
          Result := Node;
          Stack.Clear;
          end;
        end
        {в противном случае дочерние узлы этого узла в стек еще не заталкивались}
        else begin
          {затолкнуть правый дочерний узел, если он не нулевой}
          if (Node^.btChild[ctRight] <> nil) then
            Stack.Push(Node^.btChild[ctRight]);
          {затолкнуть узел, а за ним - нулевой указатель}
          Stack.Push(Node);
          Stack.Push(nil);
          {затолкнуть левый дочерний узел, если он не нулевой}
          if (Node^.btChild[ctLeft] <> nil) then
            Stack.Push(Node^.btChild[ctLeft]);
          end;
        end;
      finally
        {уничтожить стек}
        Stack.Free;
      end;
    end;
  end;

```

Нерекурсивный алгоритм обхода в глубину работает аналогично. Необходимо затолкнуть в стек корневой узел и войти в цикл, который будет выполняться до момента опустошения стека. В цикле необходимо вытолкнуть из стека верхний узел. Если он является нулевым, нужно вытолкнуть из стека следующий узел и выполнить его обработку. Если узел не является нулевым, следует затолкнуть в стек сам узел, затем нулевой указатель, затем правый дочерний узел (если он является ненулевым), а затем левый дочерний узел (если он является ненулевым). Затем необходимо снова выполнить цикл.

Листинг 8.7. Нерекурсивный обход в глубину

```
function TtdBinaryTree.btNoRecPostOrder(aAction : TtdVisitProc;
                                         aExtraData : pointer) : PtdBinTreeNode;

var
  Stack : TtdStack;
  Node : PtdBinTreeNode;
  StopNow : boolean;
begin
  {предположим, что мы не добрались до выбранного узла}
  Result := nil;
  StopNow := false;
  {создать стек}
  Stack := TtdStack.Create(nil);
  try
    {затолкнуть корневой узел}
    Stack.Push(FHead^.btChild[ctLeft]);
    {продолжать процесс до тех пор, пока стек не опустеет}
    while not Stack.IsEmpty do begin
      {извлечь узел в начале очереди}
      Node := Stack.Pop;
      {если он является нулевым, вытолкнуть из стека следующий узел и выполнить
      с ним указанное действие. Если в результате возвращается значение false
      (т.е. обход должен быть прекращен), вернуть этот узел}
      if (Node = nil) then begin
        Node := Stack.Pop;
        aAction(Node^.btData, aExtraData, StopNow);
        if StopNow then begin
          Result := Node;
          Stack.Clear;
        end;
      end
      {в противном случае дочерние узлы этого узла в стек еще не заталкивались}
      else begin
        {затолкнуть узел, а за ним – нулевой указатель}
        Stack.Push(Node);
        Stack.Push(nil);
        {затолкнуть правый дочерний узел, если он не нулевой}
        if (Node^.btChild[ctRight] <> nil) then
          Stack.Push(Node^.btChild[ctRight]);
```

```

    {затолкнуть левый дочерний узел, если он не нулевой}
    if (Node^.btChild[ctLeft] <> nil) then
        Stack.Push(Node^.btChild[ctLeft]);
    end;
end;
finally
    {уничтожить стек}
    Stack.Free;
end;
end;

```

Как и ранее, по тем же причинам, метод предполагает, что дерево является не пустым.

Обход по уровням

Мы еще не рассматривали обход по уровням, при котором вначале посещается корневой узел, затем слева направо посещаются два возможных узла на первом уровне, затем слева направо четыре возможных узла на втором уровне и т.д. Этот метод обхода кажется слишком сложным для кодирования, но в действительности он очень прост. Достаточно знать один прием. Он заключается в следующем применении очереди. Поместим корневой узел в очередь, и будем выполнять цикл до тех пор, пока очередь не опустеет. Удалим из очереди верхний узел. Посетим его. Если его левая дочерняя связь является ненулевой, поместим ее в очередь. Если правая дочерняя связь является ненулевой, поместим в очередь и ее. Если очередь не пуста, снова выполним цикл. Вот, собственно, и все.

Листинг 8.8. Обход по уровням

```

function TtdBinaryTree.btLevelOrder(aAction : TtdVisitProc;
                                     aExtraData : pointer) : PtdBinTreeNode;
var
    Queue : TtdQueue;
    Node : PtdBinTreeNode;
    StopNow : boolean;
begin
    {предположим, что мы не добрались до выбранного узла}
    Result := nil;
    StopNow := false;
    {создать очередь}
    Queue := TtdQueue.Create(nil);
    try
        {поместить корневой узел в очередь}
        Queue.Enqueue(FHead^.btChild[ctLeft]);
        {продолжать процесс до тех пор, пока очередь не опустеет}
        while not Queue.IsEmpty do begin
            {извлечь узел в начале очереди}
            Node := Queue.Dequeue;

```

```

{выполнить действия с ним. Если в результате возвращается запрос на
прекращение обхода, вернуть этот узел}
aAction(Node^.btData, aExtraData, StopNow);
if StopNow then begin
    Result := Node;
    Queue.Clear;
end
{в противном случае продолжить процесс}
else begin
    {поместить в очередь левый дочерний узел, если он не нулевой}
    if (Node^.btChild[ctLeft] <> nil) then
        Queue.Enqueue(Node^.btChild[ctLeft]);
    {поместить в очередь правый дочерний узел, если он не нулевой}
    if (Node^.btChild[ctRight] <> nil) then
        Queue.Enqueue(Node^.btChild[ctRight]);
    end;
end;
finally
    {уничтожить очередь}
    Queue.Free;
end;
end;

```

Подобно методам нерекурсивного обхода, метод **btLevelOrder** должен вызываться только для дерева, которое является непустым.

Реализация класса бинарных деревьев

Как и в случае остальных уже рассмотренных структур данных, мы реализуем стандартное бинарное дерево в виде класса. Действительно, мы уже положили начало такому подходу, рассмотрев различные методы готового класса.

В идеале, как, например, это было сделано для связных списков, желательно освободить пользователя класса от необходимости разбираться в структуре узлов (это позволит нам впоследствии изменять их структуру, не причиняя неудобств пользователю класса). Но в случае использования обычных бинарных деревьев приходится предполагать наличие у пользователя определенных знаний о структуре узлов, которые позволяют ему вставить новый узел (пользователь должен сообщить классу дерева, какой узел является родительским, и каким дочерним узлом становится новый узел). Поэтому наша реализация будет "черным ящиком" не совсем в той степени, в какой хотелось бы.

Класс бинарного дерева будет поддерживать такие стандартные операции, как вставка и удаление. Кроме того, его метод **Traverse** будет поддерживать различные виды обхода. Одним из методов, который мог бы обеспечить определенные преимущества при решении задач, подобных синтаксическому анализу выражений, была бы операция объединения двух деревьев в новый корневой узел.

Листинг 8.9. Интерфейс класса бинарного дерева

```
type
  TtdBinaryTree = class
    {класс бинарного дерева}
  private
    FCount : integer;
    FDispose : TtdDisposeProc;
    FHead : PtdBinTreeNode;
    FName : TtdNameString;
  protected
    procedure btError(aErrorCode : integer;
                      const aMethodName : TtdNameString);
    function btLevelOrder(aAction : TtdVisitProc;
                          aExtraData : pointer) : PtdBinTreeNode;
    function btNoRecInOrder(aAction : TtdVisitProc;
                           aExtraData : pointer) : PtdBinTreeNode;
    function btNoRecPostOrder(aAction : TtdVisitProc;
                              aExtraData : pointer) : PtdBinTreeNode;
    function btNoRecPreOrder(aAction : TtdVisitProc;
                             aExtraData : pointer) : PtdBinTreeNode;
    function btRecInOrder(aNode : PtdBinTreeNode;
                          aAction : TtdVisitProc;
                          aExtraData : pointer) : PtdBinTreeNode;
    function btRecPostOrder(aNode : PtdBinTreeNode;
                            aAction : TtdVisitProc;
                            aExtraData : pointer) : PtdBinTreeNode;
    function btRecPreOrder(aNode : PtdBinTreeNode;
                           aAction : TtdVisitProc;
                           aExtraData : pointer) : PtdBinTreeNode;
  public
    constructor Create(aDisposeItem : TtdDisposeProc);
    destructor Destroy; override;
    procedure Clear;
    procedure Delete(aNode : PtdBinTreeNode);
    function InsertAt(aParentNode : PtdBinTreeNode;
                     aChildType : TtdChildType;
                     aItem : pointer) : PtdBinTreeNode;
    function Root : PtdBinTreeNode;
    function Traverse(aMode : TtdTraversalMode;
                     aAction : TtdVisitProc;
                     aExtraData : pointer;
                     aUseRecursion : boolean) : PtdBinTreeNode;
    property Count : integer read FCount;
    property Name : TtdNameString read FName write FName;
  end;
```

Как обычно при использовании структур данных, рассмотренных в этой книге, мы убеждаемся, что класс владеет содержащимися в нем данными и, следовательно, может их при необходимости освобождать, или же предполагаем, что обработка данных выполняется из какого-то другого места, и в этом случае дерево не будет освобождать какие-либо данные. Поэтому конструктор **Create** принимает параметр, определяющий процедуру удаления элемента данных. Если этот параметр является нулевым, дерево не владеет данными и, следовательно, не будет их удалять. Если параметр **aDisposeItem** является адресом процедуры, эта процедура будет вызываться в каждом случае, когда требуется освободить элемент.

Листинг 8.10. Методы **Create** и **Destroy** класса бинарного дерева

```

constructor TtdBinaryTree.Create(aDisposeItem : TtdDisposeProc);
begin
    inherited Create;
    FDispose := aDisposeItem;
    {проверить, доступен ли диспетчер узлов}
    if (BTNodeManager = nil) then
        BTNodeManager := TtdNodeManager.Create(sizeof(TtdBinTreeNode));
    {выделить заглавный узел; со временем корневой узел дерева станет его левым
    дочерним узлом}
    FHead := BTNodeManager.AllocNodeClear;
end;

destructor TtdBinaryTree.Destroy;
begin
    Clear;
    BTNodeManager.FreeNode(FHead);
    inherited Destroy;
end;

```

Метод **Create** убеждается, что диспетчер узлов бинарного дерева активен, а затем выделяет фиктивный заглавный узел. Именно на месте левого дочернего узла этого узла находится корневой узел дерева. Метод **Destroy** убеждается, что дерево очищено (т.е. все узлы в дереве освобождены), а затем освобождает фиктивный заглавный узел.

Следующий метод, который мы рассмотрим – метод **Clear**. В данном случае требуется удалить все узлы дерева. Как упоминалось ранее, это выполняется за счет применения обхода всего дерева в глубину. В данном случае мы воспользовались нерекурсивным обходом, поскольку он выполняется быстрее.

Листинг 8.11. Очистка бинарного дерева

```

procedure TtdBinaryTree.Clear;
var
    Stack : TtdStack;
    Node : PtdBinTreeNode;

```



```

begin
  if (FCount = 0) then
    Exit;
    {создать стек}
  Stack := TtdStack.Create(nil);
  try
    {затолкнуть корневой узел}
    Stack.Push(FHead^.btChild[ctLeft]);
    {продолжать процесс до тех пор, пока стек не опустеет}
    while not Stack.IsEmpty do begin
      {извлечь узел в начале очереди}
      Node := Stack.Pop;
      {если он является нулевым, вытолкнуть из стека следующий узел и освободить его}
      if (Node = nil) then begin
        Node := Stack.Pop;
        if Assigned(FDispose) then
          FDispose(Node^.btData);
        BTNodeManager.FreeNode(Node);
      end
      {в противном случае дочерние узлы этого узла в стек еще не заталкивались}
      else begin
        {затолкнуть узел, а за ним – нулевой указатель}
        Stack.Push(Node);
        Stack.Push(nil);
        {затолкнуть правый дочерний узел, если он не нулевой}
        if (Node^.btChild[ctRight] <> nil) then
          Stack.Push(Node^.btChild[ctRight]);
        {затолкнуть левый дочерний узел, если он не нулевой}
        if (Node^.btChild[ctLeft] <> nil) then
          Stack.Push(Node^.btChild[ctLeft]);
      end;
    end;
  finally
    {уничтожить стек}
    Stack.Free;
  end;
  {внести изменения, отражающие то, что дерево пусто}
  FCount := 0;
  FHead^.btChild[ctLeft] := nil;
end;

```

Если сравнить этот код с кодом общего метода нерекурсивного обхода, приведенным в листинге 8.7, то несложно заметить, что они во многом совпадают. Единственное реальное различие состоит в том, что в коде отсутствует какая-либо процедура действия – мы уже знаем, что будет делаться с каждым узлом.

Метод **Traverse** действует всего лишь в качестве контейнера различных внутренних методов обхода, большинство из которых мы уже рассмотрели. Остальные методы представляют собой соответствующие рекурсивные методы обхода дерева.

Листинг 8.12. Обход в классе бинарного дерева

```
function TtdBinaryTree.btRecInOrder(aNode : PtdBinTreeNode;
                                   aAction : TtdVisitProc;
                                   aExtraData : pointer) : PtdBinTreeNode;

var
  StopNow : boolean;
begin
  Result := nil;
  if (aNode^.btChild[ctLeft] <> nil) then begin
    Result := btRecInOrder(aNode^.btChild[ctLeft],
                          aAction, aExtraData);
    if (Result <> nil) then
      Exit;
  end;
  StopNow := false;
  aAction(aNode^.btData, aExtraData, StopNow);
  if StopNow then begin
    Result := aNode;
    Exit;
  end;
  if (aNode^.btChild[ctRight] <> nil) then begin
    Result := btRecInOrder(aNode^.btChild[ctRight],
                          aAction, aExtraData);
  end;
end;

function TtdBinaryTree.btRecPostOrder(aNode : PtdBinTreeNode;
                                       aAction : TtdVisitProc;
                                       aExtraData : pointer) : PtdBinTreeNode;

var
  StopNow : boolean;
begin
  Result := nil;
  if (aNode^.btChild[ctLeft] <> nil) then begin
    Result := btRecPostOrder(aNode^.btChild[ctLeft], aAction, aExtraData);
    if (Result <> nil) then
      Exit;
  end;
  if (aNode^.btChild[ctRight] <> nil) then begin
    Result := btRecPostOrder(aNode^.btChild[ctRight],
                          aAction, aExtraData);
    if (Result <> nil) then
      Exit;
  end;
end;
```

```

    StopNow := false;
    aAction(aNode^.btData, aExtraData, StopNow);
    if StopNow then
        Result := aNode;
end;

function TtdBinaryTree.btRecPreOrder(aNode : PtdBinTreeNode;
                                     aAction : TtdVisitProc;
                                     aExtraData : pointer) : PtdBinTreeNode;

var
    StopNow : boolean;
begin
    Result := nil;
    StopNow := false;
    aAction(aNode^.btData, aExtraData, StopNow);
    if StopNow then begin
        Result := aNode;
        Exit;
    end;
    if (aNode^.btChild[ctLeft] <> nil) then begin
        Result := btRecPreOrder(aNode^.btChild[ctLeft], aAction, aExtraData);
        if (Result <> nil) then
            Exit;
    end;
    if (aNode^.btChild[ctRight] <> nil) then begin
        Result := btRecPreOrder(aNode^.btChild[ctRight], aAction, aExtraData);
    end;
end;

function TtdBinaryTree.Traverse(aMode : TtdTraversalMode;
                                aAction : TtdVisitProc;
                                aExtraData : pointer;
                                aUseRecursion : boolean) : PtdBinTreeNode;

var
    RootNode : PtdBinTreeNode;
begin
    Result := nil;
    RootNode := FHead^.btChild[ctLeft];
    if (RootNode <> nil) then begin
        case aMode of
            tmPreOrder :
                if aUseRecursion then
                    Result := btRecPreOrder(RootNode, aAction, aExtraData)
                else
                    Result := btNoRecPreOrder(aAction, aExtraData);
            tmInOrder :
                if aUseRecursion then
                    Result := btRecInOrder(RootNode, aAction, aExtraData)
                else
                    Result := btNoRecInOrder(aAction, aExtraData);
        end;
    end;
end;

```

```
tmPostOrder :  
  if aUseRecursion then  
    Result := btRecPostOrder(RootNode, aAction, aExtraData)  
  else  
    Result := btNoRecPostOrder(aAction, aExtraData);  
tmLevelOrder :  
  Result := btLevelOrder(aAction, aExtraData);  
end;  
end;  
end;
```

Как видно из кода внутренних рекурсивных процедур, возможность прекращения обхода в любой момент времени делает код несколько менее читабельным и более сложным.

Исходный код класса **TtdBinaryTree** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDBinTre.pas**.

Деревья бинарного поиска

Хотя бинарные деревья являются структурами данных, которые представляют интерес и сами по себе, на практике в основном используют бинарные деревья, содержащие элементы в сортированном виде. Такие бинарные деревья называют *деревьями бинарного поиска* (*binary search tree*).

В дереве бинарного поиска каждый узел имеет *ключ*. (В деревьях бинарного поиска, которые будут построены в этой главе, считается, что ключ является частью элемента, вставляемого в дерево. Для сравнения двух элементов, а, следовательно, и их ключей, мы будем использовать подпрограмму **TtdCompare**.) Упорядочение применяется ко всем узлам в дереве: для каждого узла ключ левого дочернего узла меньше или равен ключу узла, а этот ключ, в свою очередь, меньше или равен ключу правого дочернего узла. Если описанное упорядочение постоянно применяется во время вставки (как именно – будет показано чуть ниже), это также означает, что для каждого узла все ключи в левом дочернем дереве меньше или равны ключу узла, а все ключи в правом дочернем дереве больше или равны ключу узла.

Какие основные операции претерпевают изменения в случае использования дерева бинарного поиска вместо обычного бинарного дерева? Что ж, все алгоритмы обхода работают так же, как и ранее (фактически, при симметричном обходе все узлы в дереве бинарного поиска посещаются в порядке ключей – отсюда и английское название этого метода "in-order"). Однако операции вставки и удаления должны быть изменены, поскольку они могут нарушить порядок ключей в дереве бинарного поиска. Поиск элемента может быть выполнен значительно быстрее.

Алгоритм поиска в дереве бинарного поиска использует упорядоченность дерева. Поиск элемента выполняется следующим образом. Поиск начинается с корневого узла, и этот узел становится текущим. Затем ключ искомого элемента сравнивается с ключом текущего узла. Если они равны, дело сделано, поскольку мы нашли требуемый элемент в дереве. В противном случае, если ключ элемента меньше ключа текущего узла, мы делаем левый дочерний узел текущим. Если он больше, мы делаем текущим правый дочерний узел и возвращаемся к шагу выполнения сравнения. Со временем мы либо найдем нужный узел, либо встретим нулевой дочерний узел, что свидетельствует об отсутствии искомого элемента в дереве.

Следует отметить одну особенность этого алгоритма в случае наличия в дереве нескольких элементов с равными ключами: не существует никаких гарантий, что мы найдем какой-то конкретный элемент с соответствующим ключом. Им может оказаться первый элемент, последний или любой промежуточный. Фактически, в основном по тем же причинам, что и при использовании списка с пропусками, желательно гарантировать, чтобы все элементы в дереве бинарного поиска имели уникальные, различающиеся между собой ключи. Присутствие дублированных ключей не допускается. На практике это правило не создает особых трудностей: если можно различить два элемента, должно быть не трудно обеспечить их различие и в дереве бинарного поиска. Обычно это достигается за счет использования младших ключей (например, фамилия служит в качестве главного ключа, а имя – в качестве контрольного значения, когда фамилии совпадают). Таким образом, деревья бинарного поиска, рассмотренные в этой главе, будут подчиняться правилу недопустимости дублированных ключей. В результате определение дерева бинарного поиска будет формулироваться следующим образом: это дерево, в котором ключ левого дочернего узла строго меньше ключа данного узла, который, в свою очередь, строго меньше ключа правого дочернего узла.

Алгоритм поиска в дереве бинарного поиска имитирует стандартный бинарный поиск в массиве или в связном списке. В каждом узле мы принимаем решение, какой дочерней связью нужно следовать. При этом можно игнорировать все узлы, находящиеся в другом дочернем дереве. Если дерево *сбалансировано*, алгоритм поиска является операцией типа $O(\log(n))$. Другими словами, среднее время, затрачиваемое на поиск любого элемента, пропорционально $\log_2$ от числа элементов в дереве. Под сбалансированным мы будем понимать дерево, в котором длина пути от любого листа до корневого узла приблизительно одинакова, причем дерево имеет минимальное количество уровней, необходимое для данного количества присутствующих узлов.

Листинг 8.13. Поиск в дереве бинарного поиска

```

function TtdBinarySearchTree.bstFindItem(aItem : pointer;
    var aNode : PtdBinTreeNode; var aChild : TtdChildType) : boolean;
var
    Walker : PtdBinTreeNode;
    CmpResult : integer;
begin
    Result := false;
    {если дерево пусто, вернуть нулевой и левый узел для указания того, что новый
    узел, в случае его вставки, должен быть корневым}
    if (FCount = 0) then begin
        aNode := nil;
        aChild := ctLeft;
        Exit;
    end;
    {в противном случае перемещаться по дереву}
    Walker := FBinTree.Root;
    CmpResult := FCompare(aItem, Walker^.btData);
    while (CmpResult <> 0) do begin
        if (CmpResult < 0) then begin
            if (Walker^.btChild[ctLeft] = nil) then begin
                aNode := Walker;
                aChild := ctLeft;
                Exit;
            end;
            Walker := Walker^.btChild[ctLeft];
        end
        else begin
            if (Walker^.btChild[ctRight] = nil) then begin
                aNode := Walker;
                aChild := ctRight;
                Exit;
            end;
            Walker := Walker^.btChild[ctRight];
        end;
        CmpResult := FCompare(aItem, Walker^.btData);
    end;
    Result := true;
    aNode := Walker;
end;

function TtdBinarySearchTree.Find(aKeyItem : pointer) : pointer;
var
    Node : PtdBinTreeNode;
    ChildType : TtdChildType;
begin
    if bstFindItem(aKeyItem, Node, ChildType) then
        Result := Node^.btData
    else
        Result := nil;
end;

```

В коде, представленном в листинге 8.13, не используются отдельные ключи для каждого элемента. Вместо этого предполагается, что свойство упорядочения дерева бинарного поиска определяется функцией сравнения, подобно тому, как это делалось в отсортированных связных списках, списках с пропусками и т.п. Функция сравнения дерева бинарного поиска объявляется конструктором **Create**.

Метод **Find** использует внутренний метод **bstFindItem**. Этот метод должен вызываться для достижения двух различных целей. Во-первых, самим методом **Find**, и, во-вторых, методом, который вставляет новые узлы в дерево (этот метод мы рассмотрим несколько позже). Соответственно, если элемент не был найден, метод будет возвращать место, в которое он должен быть вставлен. Естественно, эта функция не требуется для простого поиска: нам нужно только знать, существует ли элемент, и если существует, то получить элемент целиком обратно.

В представленном коде следует также отметить, что класс используется внутренний экземпляр **TtdBinaryTree**, названный **FBinTree**, для хранения фактического бинарного дерева. Как будет показано, класс дерева бинарного поиска делегирует все операции бинарного дерева этому внутреннему бинарному дереву. Легко заметить, что от этого внутреннего объекта требуется получить только корневой узел. С этого момента остается только перемещаться по узлам.

Вставка в дерево бинарного поиска

Мы можем существенно упростить операцию вставки для пользователя дерева бинарного поиска: он должен предоставить только сам элемент. Пользователь не должен также беспокоиться о том, какой узел становится родительским, и в качестве какого дочернего узла добавляется новый узел. Все это, скрывая подробности, может выполнить дерево бинарного поиска, используя в качестве руководства к действию порядок элементов внутри дерева.

Фактически, вставить новый элемент в дерево бинарного поиска достаточно просто, и большая часть этого процесса уже была рассмотрена. Мы ищем элемент до тех пор, пока не достигаем точки, когда дальнейший спуск оказывается невозможен, поскольку дочерняя связь, которой нужно было бы следовать, является нулевой. К этому моменту мы знаем, где должен размещаться элемент, – в точке, где мы должны были остановиться. При этом известно, каким дочерним узлом должен быть элемент, и, естественно, мы останавливаемся на родительском узле нового узла. Обратите также внимание, что используемый алгоритм поиска места для вставки нового элемента гарантирует целостность порядка элементов в дереве бинарного поиска.

Тем не менее, алгоритм вставки сопряжен с одной проблемой. Хотя метод гарантирует создание допустимого дерева бинарного поиска после выполнения операции, созданное дерево может быть неоптимальным или неэффективным. Чтобы понять, о чем идет речь, вставьте элементы *a*, *b*, *c*, *d*, *e* и *f* в пустое дерево бинарного поиска. С элементом *a* все просто – он становится корневым узлом. Элемент

b добавляется в качестве правого дочернего узла элемента a . Элемент c добавляется в качестве правого дочернего узла элемента b и т.д. Результат показан слева на рис. 8.2: он представляет собой длинное вытянутое дерево, которое можно трактовать как связного списка. В идеале желательно, чтобы дерево было более сбалансированным. Для только что созданного вырожденного дерева время поиска пропорционально числу элементов в дереве ($O(n)$), а не $\log_2$ числа элементов ($O(\log(n))$). Возможны также другие случаи вырождения. Например, попытайтесь выполнить следующую последовательность вставок: a, f, b, e, c и d , в результате которой создается явно вырожденное дерево, показанное справа на рис. 8.2.

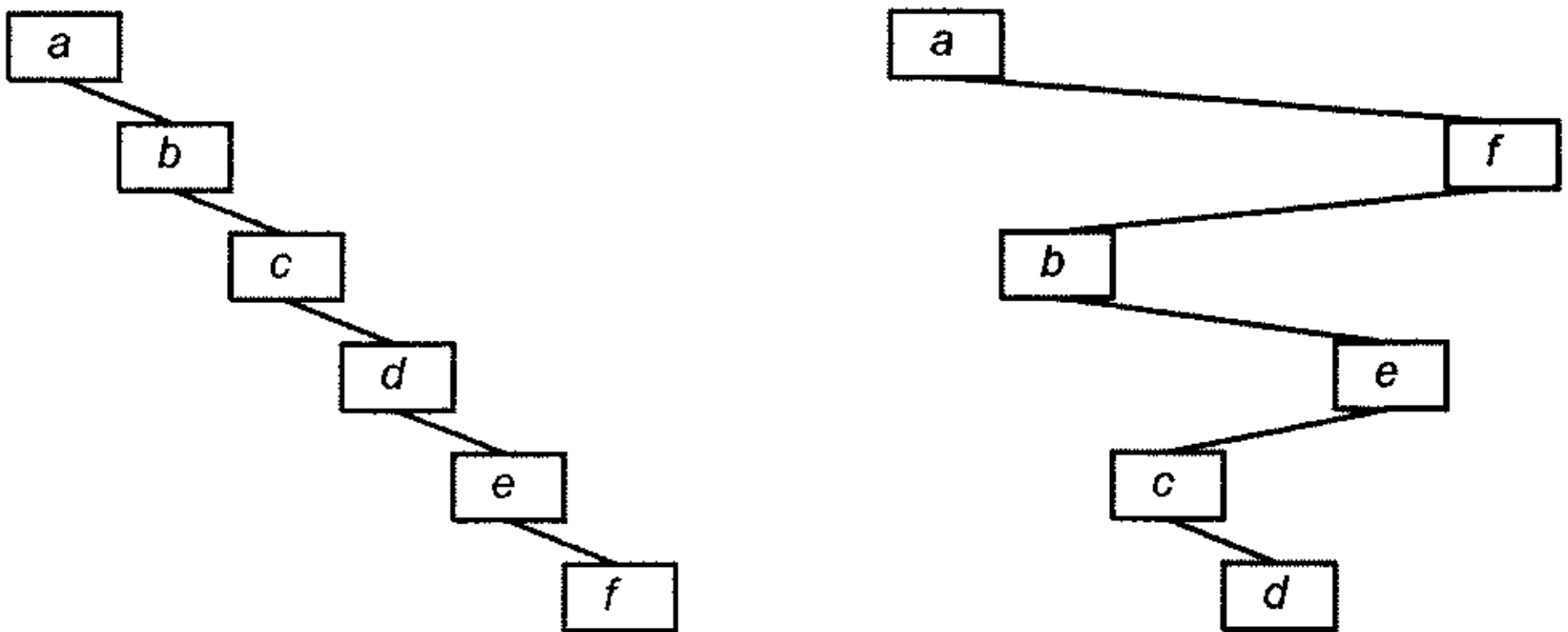


Рисунок 8.2. Вырожденные деревья бинарного поиска

В связи с возникновением описанных проблем, этот простой алгоритм вставки вряд ли будет применяться на практике. Если бы можно было гарантировать случайный порядок вставки ключей и элементов, или если бы общее количество элементов было очень небольшим, описанный алгоритм вставки оказался бы вполне приемлемым. Однако в общем случае подобную гарантию просто нельзя дать, и поэтому необходимо использовать более сложный алгоритм вставки, частью которого является попытка сбалансировать дерево бинарного поиска. Эта методика балансировки будет рассмотрена в ходе ознакомления с красно-черными деревьями (RB-деревьями).

Важно иметь в виду следующее. Рассмотренные алгоритмы вставки и удаления гарантированно создают допустимое дерево бинарного поиска. Однако при этом весьма вероятно, что дерево будет скошенным и несбалансированным. Для небольших деревьев бинарного поиска это не имеет особого значения (в конце концов, для малых значений n $\log(n)$ и n — величины более-менее одного порядка, поэтому выигрыш в значении O большого будет небольшим), тем не менее, для больших деревьев такое различие поистине огромно.

Возвращаясь к простому алгоритму вставки, мы видим, что для вставки n элементов в дерево бинарного поиска в среднем требуется время, пропорциональное $O(n \log(n))$ (другими словами, для каждой вставки используется алгоритм $O(\log(n))$ для выяснения места, в которое должен быть помещен новый элемент, а количество вставляемых элементов равно n). В случае вырождения вставка n элементов превращается в операцию типа $O(n^2)$.

Листинг 8.14. Вставка в дерево бинарного поиска

```
function TtdBinarySearchTree.bstInsertPrim(aItem : pointer;
      var aChildType : TtdChildType) : PtdBinTreeNode;
begin
  { вначале предпринять попытку найти элемент; если он найден, сгенерировать
    ошибку }
  if bstFindItem(aItem, Result, aChildType) then
    bstError(tdeBinTreeDupItem, 'bstInsertPrim');
  { эта операция возвращает узел, поэтому вставку потребуется выполнить здесь }
  Result := FBinTree.InsertAt(Result, aChildType, aItem);
  inc(FCount);
end;

procedure TtdBinarySearchTree.Insert(aItem : pointer);
var
  ChildType : TtdChildType;
begin
  bstInsertPrim(aItem, ChildType);
end;
```

Для выполнения большей части работы мы используем внутреннюю процедуру **bstInsertPrim**. Это делается для того, чтобы разделить код собственно вставки и код метода **Insert**, что впоследствии упростит нашу задачу при создании производных деревьев от дерева бинарного поиска для выполнения операции балансировки. Как видите, процедура **bstInsertPrim** возвращает вставленный узел и использует метод **bstFindItem**, который уже встречался в листинге 8.13.

Таким образом, фактическую вставку мы делегируем объекту бинарного дерева, который использует свой метод **InsertAt**.

Удаление из дерева бинарного поиска

Как и при выполнении предыдущей операции, большая часть проблем может быть скрыта от пользователя дерева бинарного поиска. Однако дерево должно выполнить определенную, более сложную задачу.

Естественно, первым шагом является поиск элемента в дереве с применением стандартного алгоритма. Если найти элемент не удастся, придется как-то сообщить о неудаче. В случае обнаружения элемента, поиск может быть прерван в узле одного из трех типов, как это имеет место в стандартном бинарном дереве.

Первый тип узла – узел без дочерних узлов, обе дочерние связи которого являются нулевыми. Иначе говоря, лист. Чтобы удалить узел этого типа, мы просто разрываем его связь с родительским узлом и удаляем его. Это удаление не нарушает порядок узлов в дереве – в конце концов, узел был листом и не имел дочерних узлов.

Второй тип узла – узел только с одним дочерним узлом. В случае стандартного бинарного дерева мы просто перемещаем дочерний узел на один уровень вверх, чтобы заменить удаляемый узел. Можно ли это же сделать в данном случае? Рассмотрим родительский узел узла, который должен быть удален. Удаленный узел является либо левым дочерним узлом (в этом случае его ключ меньше ключа родительского узла), либо правым дочерним узлом (в этом случае его ключ больше ключа родительского узла). Не только этот узел, но и все дочерние, "внучатые" и так далее узлы удаленного узла обладают тем же свойством. Все они будут либо меньше родительского узла, либо больше. Таким образом, до тех пор, пока речь идет о родительском узле, при замене узла одним из его дочерних узлов свойство упорядочения будет сохраняться. Если дочерний узел имеет свои дочерние узлы, это перемещение не сказывается на них или на их порядке. Следовательно, в случае дерева бинарного поиска мы по-прежнему можем выполнить эту простую операцию.

Третий тип узла – узел с двумя дочерними узлами. В стандартном дереве бинарного поиска мы считали попытку удаления узла этого типа ошибкой. Удаление не могло быть выполнено, поскольку не существовало никакого общего способа выполнения операции удаления, который имел бы смысл. В случае дерева бинарного поиска это не так: в данном случае можно воспользоваться свойством упорядочения дерева бинарного поиска.

Ситуация выглядит следующим образом: нам нужно удалить определенный узел (т.е. элемент в этом узле), но он имеет два дочерних узла (каждый из которых имеет собственные дочерние узлы). Алгоритм удаления несколько сложен, поэтому вначале он будет описан словесно, а затем будет показано, как он работает. На практике мы ищем узел, содержащий наибольший элемент, который меньше только того, который мы пытаемся удалить. Затем мы меняем местами элементы в этих двух узлах. И, наконец, мы удаляем второй узел. Он всегда будет соответствовать одному из ранее рассмотренных случаев удаления.

Первый шаг заключается в отыскании наибольшего элемента, меньшего того элемента, который мы пытаемся удалить. Понятно, что он находится в левом дочернем дереве (все элементы этого дерева меньше удаляемого элемента). Кроме того, он является наибольшим элементом этого дерева. Иначе говоря, все остальные элементы, которые могут находиться в левом дочернем дереве, меньше этого элемента. В действительности все элементы в *правом* дочернем дереве больше этого выбранного элемента (поскольку он меньше элемента, который должен быть удален, а этот элемент, в свою очередь, меньше всех элементов в правом дочер-

нем дереве). Следовательно, он вполне может заменить удаляемый элемент, и это действие не нарушит порядок элементов в дереве.

Но как насчет узла, с позиции которого он был перемещен, и который теперь нужно удалить? В отношении этого конкретного узла важно уяснить, что он не имеет *никакого правого дочернего узла*. Если бы он имел правый дочерний узел, элемент в дочернем узле должен был бы быть больше элемента, с которым мы меняли его местами, и, следовательно, первоначально выбранный элемент не мог бы быть наибольшим. Он может иметь левый дочерний узел, но независимо от этого мы знаем, как удалить узел, имеющий не более одного дочернего узла.

При этом все еще остается проблема обнаружения наибольшего элемента, который меньше исходного, предназначенного для удаления. По существу, мы выполняем перемещение по дереву. Начиная с элемента, который нужно удалить, мы переходим к левой дочерней связи. С этого места мы продолжаем перемещаться по правым дочерним связям до тех пор, пока не доберемся до узла, не имеющего никакой правой дочерней связи. Этот элемент гарантированно содержит наибольший элемент, меньший только того элемента, который мы пытаемся удалить.

Обратите также внимание, что удаление, как и вставка, может приводить к созданию вырожденного дерева. Эту проблему решают алгоритмы балансировки, которые мы рассмотрим при ознакомлении с красно-черным вариантом дерева бинарного поиска.

Листинг 8.15. Удаление из дерева бинарного поиска

```
function TtdBinarySearchTree.bstFindNodeToDelete(aItem : pointer)
    : PtdBinTreeNode;

var
    Walker : PtdBinTreeNode;
    Node : PtdBinTreeNode;
    Temp : pointer;
    ChildType : TtdChildType;
begin
    {попытаться найти элемент; если элемент не найден, сгенерировать признак
    ошибки}
    if not bstFindItem(aItem, Node, ChildType) then
        bstError(tdeBinTreeItemMissing, 'bstFindNodeToDelete');
    {если узел имеет два дочерних узла, найти наибольший узел, который меньше
    удаляемого, и поменять местами элементы}
    if (Node^.btChild[ctLeft] <> nil) and (Node^.btChild[ctRight] <> nil) then
        begin
            Walker := Node^.btChild[ctLeft];
            while (Walker^.btChild[ctRight] <> nil) do
                Walker := Walker^.btChild[ctRight];
            Temp := Walker^.btData;
            Walker^.btData := Node^.btData;
            Node^.btData := Temp;
            Node := Walker;
        end;
end;
```



```

    { вернуть узел, который нужно удалить }
    Result := Node;
end;

procedure TtdBinarySearchTree.Delete(aItem : pointer);
begin
    FBinTree.Delete(bstFindNodeToDelete(aItem));
    dec(FCount);
end;

```

Большая часть работы выполняется методом **bstFindNodeToDelete**. Он вызывает метод **bstFindItem**, чтобы найти элемент, который требуется удалить (естественно, если он не найден, генерируется ошибка), а затем проверяет, имеет ли найденный узел два дочерних узла. Если имеет, мы ищем узел с наибольшим элементом, который меньше удаляемого элемента. Мы меняем местами элементы в узлах и возвращаем второй элемент.

Реализация класса дерева бинарного поиска

Как обычно, дерево бинарного поиска будет реализовано в виде класса, хотя хотелось бы еще раз предупредить, что его следует использовать только в том случае, если есть уверенность, что вставляемые элементы являются в достаточной степени случайными или их количество достаточно мало, чтобы дерево не выродилось в длинную вытянутую структуру. Основное назначение класса дерева бинарного поиска – попытка сокрытия от пользователя внутренней структуры дерева. Это означает, что пользователь должен иметь возможность использовать класс для поддержания набора элементов в отсортированном порядке и выполнения их обхода без необходимости знания структуры внутренних узлов.

При реализации дерева бинарного поиска мы не будем использовать наследование от класса бинарного поиска, описанного в первой части этой главы. В основном, это обусловлено тем, что класс бинарного дерева открывает пользователю слишком много подробностей внутренней структуры узлов. Вместо этого мы делегируем функции вставки, удаления и обхода внутреннему объекту бинарного дерева. Просто на тот случай, если пользователю потребуется знание внутреннего объекта дерева, мы откроем его через соответствующее свойство.

Листинг 8.16. Интерфейс дерева бинарного поиска

```

type
    TtdBinarySearchTree = class
        { класс дерева бинарного поиска }
    private
        FBinTree : TtdBinaryTree;
        FCompare : TtdCompareFunc;
        FCount : integer;
        FName : TtdNameString;

```



```

protected
  procedure bstError(aErrorCode : integer;
                    const aMethodName : TtdNameString);
  function bstFindItem(aItem : pointer; var aNode : PtdBinTreeNode;
                    var aChild : TtdChildType) : boolean;
  function bstFindNodeToDelete(aItem : pointer) : PtdBinTreeNode;
  function bstInsertPrim(aItem : pointer; var aChildType : TtdChildType)
                    : PtdBinTreeNode;
public
  constructor Create(aCompare : TtdCompareFunc;
                    aDispose : TtdDisposeProc);
  destructor Destroy; override;
  procedure Clear;
  procedure Delete(aItem : pointer); virtual;
  function Find(aKeyItem : pointer) : pointer; virtual;
  procedure Insert(aItem : pointer); virtual;
  function Traverse(aMode : TtdTraversalMode;
                    aAction : TtdVisitProc;
                    aExtraData : pointer;
                    aUseRecursion : boolean) : pointer;
  property BinaryTree : TtdBinaryTree read FBinTree;
  property Count : integer read FCount;
  property Name : TtdNameString read FName write FName;
end;

```

Глядя на определение этого класса, легко убедиться, что мы уже встречались с большинством методов.

Исходный код класса **TtdBinarySearchTree** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDBinTre.pas**.

Перекомпоновка дерева бинарного поиска

В ходе рассмотрения дерева бинарного поиска неоднократно упоминалось, что добавление элементов в дерево бинарного поиска может сделать его крайне несбалансированным, а иногда даже привести к его вырождению в длинное вытянутое дерево, подобное связному списку.

Проблема этого вырождения заключается не в том, что дерево перестает корректно функционировать (элементы продолжают храниться в отсортированном порядке), а в том, что в данном случае эффективности древовидной структуры наносится, по сути, смертельный удар. Для идеально сбалансированного дерева (в котором все родительские узлы имеют по два дочерних узла, а все листья размещаются на одном уровне, плюс-минус один) время поиска, время вставки и время удаления соответствуют $O(\log(n))$. Иначе говоря, если для выполнения основной операции в дереве с 1000 узлов требуется время, равное t , для ее выполнения в дереве с 1000000 узлов потребуется время равное всего лишь $2t$. С другой сторо-

ны время выполнения базовых операций в вырожденном дереве пропорционально $O(n)$, и, следовательно, для выполнения этой же операции в дереве с 1 000 000 узлов потребовалось бы время, равное $1000t$.

Так каким же образом избежать этого вырождения деревьев? Ответ заключается в создании алгоритма, который осуществляет балансировку дерева бинарного поиска во время вставки и удаления элементов. Прежде чем действительно приступить к рассмотрению алгоритмов балансировки, давайте исследуем различные методы перекомпоновки деревьев бинарного поиска, а затем ими можно будет воспользоваться для балансировки деревьев.

Вспомните, что в дереве бинарного поиска все узлы в левом дочернем дереве данного узла меньше, а узлы в правом дочернем дереве больше его. (Естественно, под тем, что один узел меньше другого, подразумевается, что ключ элемента в одном узле меньше ключа элемента в другом узле. Просто проще написать, что "один узел меньше другого", нежели постоянно ссылаться на ключи узлов.) Немного проанализируем эту аксиому.

Взгляните на левый дочерний узел в дереве бинарного поиска. Что мы знаем о нем? Что ж, естественно, он имеет собственные левое и правое дочерние деревья. Он больше всех узлов в его левом дочернем дереве и меньше всех узлов в правом дочернем дереве. Более того, поскольку он является левым дочерним узлом, его родительский узел больше всех узлов в его правом дочернем дереве. Следовательно, если повернуть левый дочерний узел в позицию его родительского узла, чтобы его правое дочернее дерево стало новым левым дочерним деревом родительского узла, результирующее бинарное дерево останется допустимым. Этот поворот показан на рис. 8.3. На этом рисунке треугольники представляют дочерние деревья, которые содержат ноль или больше узлов – для алгоритма поворота точное их количество роли не играет.

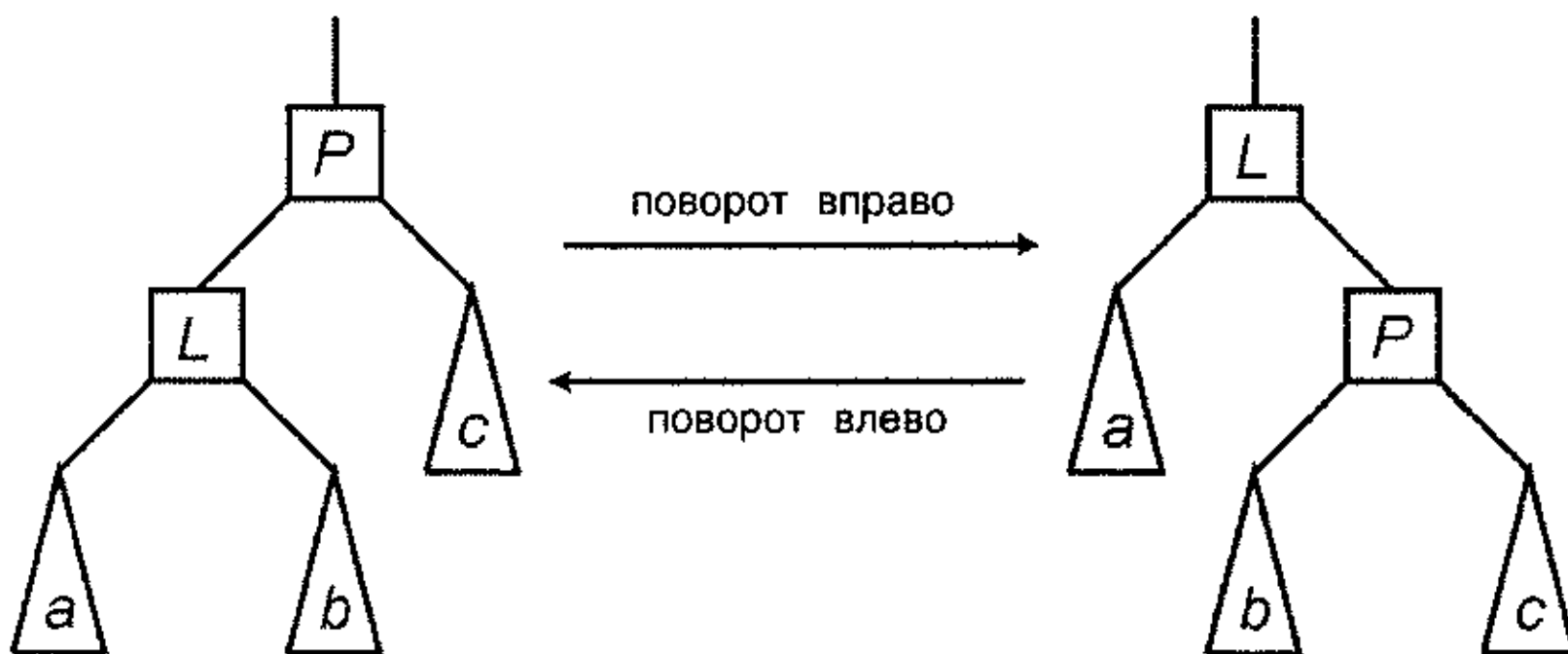


Рисунок 8.3. Повышение ранга левого дочернего узла (и наоборот)

Для исходного дерева можно было бы записать следующее неравенство: $(a < L < b) < P < c$. Для нового дерева имеем: $a < L < (b < P < c)$, что, конечно же, остается справедливым и при удалении круглых скобок, поскольку операция $<$ подчиняется коммуникативному закону. (Первое неравенство читается следующим образом: все узлы в дереве a меньше узла L , который меньше всех узлов в дереве b , а все это дерево в целом меньше узла P , который, в свою очередь, меньше всех узлов в дереве c . Подобным же образом можно интерпретировать и второе неравенство.)

Только что рассмотренную операцию называют *поворотом вправо* (*right rotation*). При этом говорят, что ранг левого дочернего узла L *повышается*, а ранг родительского узла P *понижается*. Иначе говоря, узел L перемещается на один уровень вверх, а узел P – на один уровень вниз. Такой поворот называется поворотом вокруг узла P .

Естественно, рассмотрев поворот вправо, легко предположить, как выполняется другой поворот, *поворот влево* (*left rotation*), который создал бы первое дерево из второго. Поворот влево повышает ранг правого дочернего узла P и понижает ранг родительского узла L . Код выполнения обоих видов поворота приведен в листинге 8.17, при этом кодирование выполняется с точки зрения того узла, ранг которого повышается.

Листинг 8.17. Повышение ранга узла

```
function TtdSplayTree.stPromote(aNode : PtdBinTreeNode) : PtdBinTreeNode;
var
    Parent : PtdBinTreeNode;
begin
    {поставить родительский узел того узла, ранг которого повышается}
    Parent := aNode^.btParent;
    {в обоих случаях необходимо разорвать и перестроить шесть связей: связь узла
     с его дочерним узлом и противоположную связь, связь узла с его родительским
     узлом и противоположную связь и связь родительского узла с его родительским
     узлом и противоположную связь; обратите внимание, что дочерний узел данного
     узла может быть пустым}
    {повысить ранг левого дочернего узла, т.е. выполнить поворот родительского
     узла вправо}
    if (Parent^.btChild[ctLeft] = aNode) then begin
        Parent^.btChild[ctLeft] := aNode^.btChild[ctRight];
        if (Parent^.btChild[ctLeft] <> nil) then
            Parent^.btChild[ctLeft]^.btParent := Parent;
        aNode^.btParent := Parent^.btParent;
        if (aNode^.btParent^.btChild[ctLeft] = Parent) then
            aNode^.btParent^.btChild[ctLeft] := aNode
        else
            aNode^.btParent^.btChild[ctRight] := aNode;
        aNode^.btChild[ctRight] := Parent;
        Parent^.btParent := aNode;
    end
```

```

{повысить ранг правого дочернего узла, т.е. выполнить поворот родительского
узла влево}
else begin
  Parent^.btChild[ctRight] := aNode^.btChild[ctLeft];
  if (Parent^.btChild[ctRight] <> nil) then
    Parent^.btChild[ctRight]^.btParent := Parent;
  aNode^.btParent := Parent^.btParent;
  if (aNode^.btParent^.btChild[ctLeft] = Parent) then
    aNode^.btParent^.btChild[ctLeft] := aNode
  else
    aNode^.btParent^.btChild[ctRight] := aNode;
  aNode^.btChild[ctLeft] := Parent;
  Parent^.btParent := aNode;
end;
{вернуть узел, ранг которого был повышен}
Result := aNode;
end;

```

Этот метод заимствован из класса скошенного дерева, который будет рассматриваться несколько позже. А пока важно отметить способ разрыва и преобразования связей, используемый для выполнения обоих типов повышения ранга. Поскольку переданный методу узел может быть левым или правым дочерним узлом, имеющим различные связи, которые необходимо разорвать и перестроить, по существу, этот метод представляет собой оператор **If**, учитывающий две возможности.

Два рассмотренных вида поворота реорганизуют дерево на локальном уровне, но основное свойство упорядоченности узлов дерева бинарного поиска остается неизменным. В случае выполнения поворота вправо все узлы дерева *a* перемещаются на один уровень ближе к корневому узлу, те, которые расположены в дереве *b*, остаются на том же уровне, а расположенные в дереве *c*, перемещаются на один уровень вниз. При выполнении поворота влево все узлы дерева *a* перемещаются на один уровень дальше от корневого узла, узлы дерева *b* остаются на том же уровне, а узлы дерева *c* перемещаются на один уровень вверх. Несложно догадаться, что, управляя некоторым общим алгоритмом балансировки, с помощью последовательности этих двух поворотов можно было бы восстановить баланс в дереве бинарного поиска.

Часто эти два вида поворота объединяются попарно и используются в формах так называемых *спаренных двухсторонних поворотов (zig-zag)* и *спаренных односторонних поворотов (zig-zig)*. Существуют две операции спаренного двустороннего поворота и две операции спаренного одностороннего поворота. Операция спаренного двустороннего поворота состоит либо из поворота вправо, за которым следует поворот влево, либо из поворота влево, за которым следует поворот вправо, причем конечным результатом обеих операций является повышение ранга узла на два уровня. И напротив, операции спаренного одностороннего поворота состоят из двух выполняемых последовательно поворотов вправо или влево. Цель

выполнения всех этих спаренных операций состоит в повышении ранга узла на два уровня.

На рис. 8.4 показана операция спаренного двустороннего поворота, которая начинается с поворота влево вокруг узла P. В результате ранг узла R повышается, а ранг узла P понижается. На следующем шаге выполняется вращение вправо вокруг узла G, в результате которого ранг узла R снова повышается, а ранг узла G понижается. Общим результатом операции спаренного двустороннего поворота будет локальная балансировка дерева.

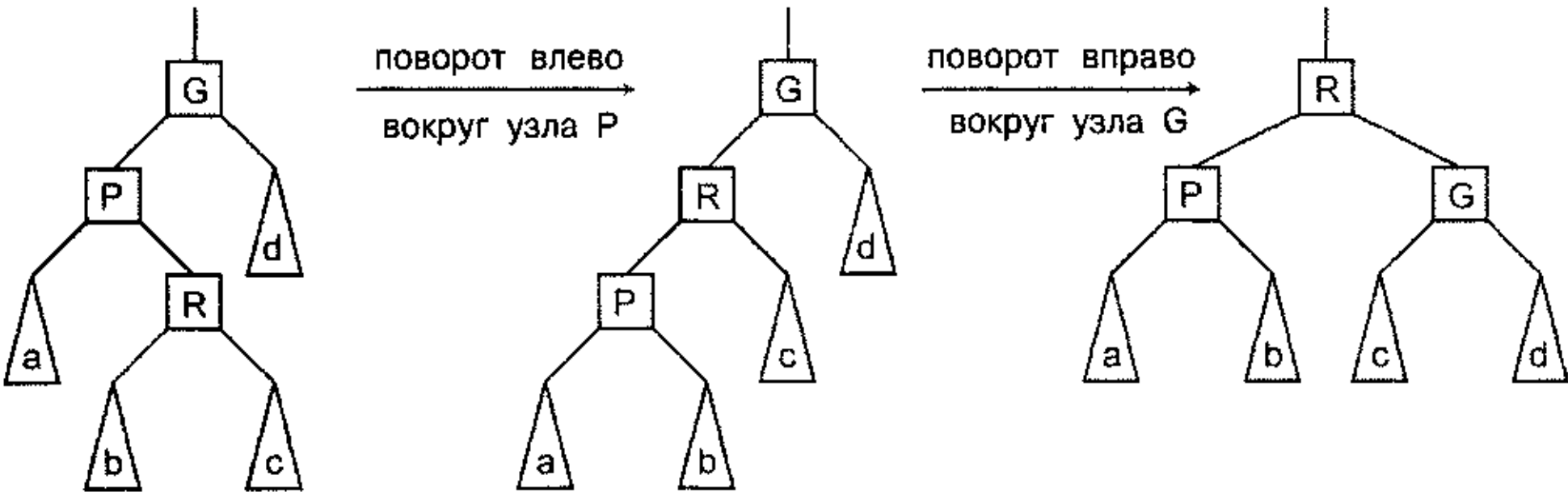


Рисунок 8.4. Операция спаренного двустороннего поворота

На рис. 8.5 изображены обе операции спаренного одностороннего поворота, поскольку они дополняют друг друга. Обратите внимание, что операция спаренного одностороннего поворота всегда начинается с поворота вокруг верхнего узла.

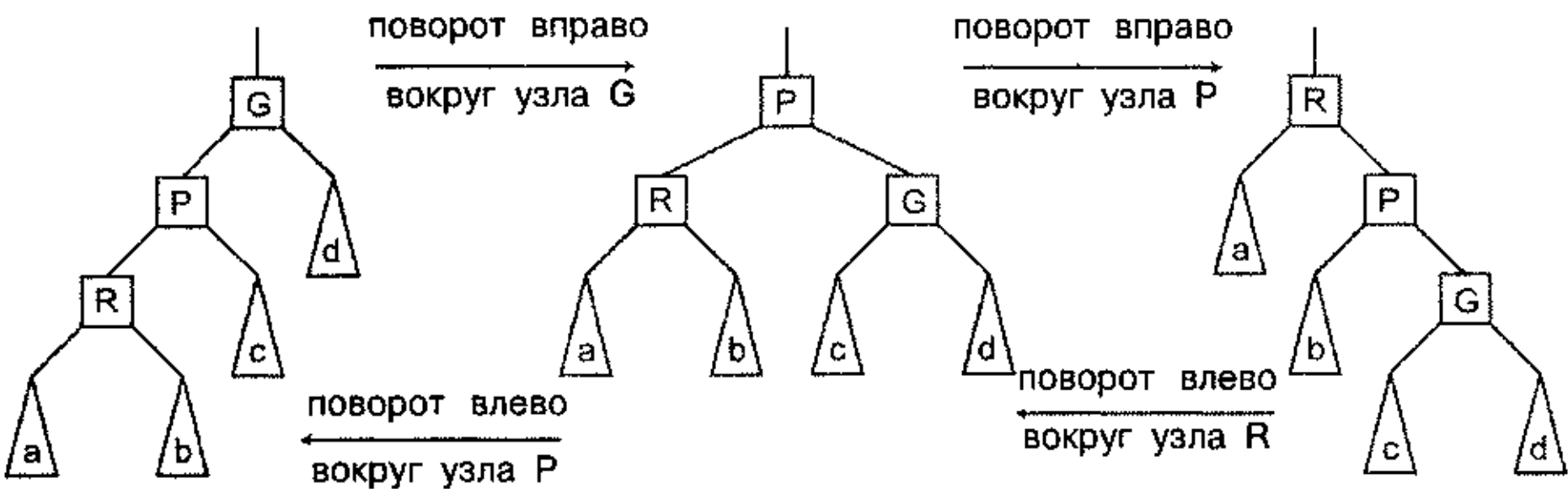


Рисунок 8.5. Операция спаренного одностороннего поворота

Скошенные деревья

Как бы то ни было, ознакомившись с этими операциями простых и спаренных двухсторонних и односторонних поворотов, мы можем их использовать в структуре данных, называемой скошенным деревом. Скошенное дерево (*splay tree*) – это дерево бинарного поиска, сконструированное таким образом, что любое обращение к узлу приводит к его скосу в сторону корневого узла. Скос заключается в применении операций спаренного двустороннего или одностороннего поворота до

тех пор, пока скашиваемый узел не окажется в позиции корневого узла дерева или на один уровень ниже него. В последнем случае его ранг можно повысить до корневого узла, выполнив одиночный поворот. Концепция скошенного дерева была изобретена Д. Д. Слеатором (D. D. Sleator) и Р. Е. Тарьяном (R. E. Tarjan) в 1985 году [22].

Вначале рассмотрим операцию поиска, т.е. нахождение конкретного узла. Мы начнем с применения стандартного алгоритма поиска в дереве бинарного поиска. Обнаружив искомый узел, мы выполняем его снос к корневому узлу.

Иначе говоря, мы применяем операции спаренного двустороннего либо одностороннего поворота, перемещая узел вверх по дереву до тех пор, пока он не достигнет позиции корневого узла. Если в результате этих операций узел оказывается на втором уровне, мы больше не можем применять операции спаренного поворота, и поэтому для перемещения в позицию корневого узла применяем поворот влево или вправо.

Если поиск был безрезультатным, в ходе него мы должны натолкнуться на нулевой узел. В этом случае мы выполняем снос узла, который был бы родительским узлом, если бы искомый узел существовал. Естественно, при этом следовало бы сообщить о невозможности как-либо найти элемент.

Вставку также легко описать: необходимо применить обычный алгоритм вставки в дерево бинарного поиска, а затем выполнить снос добавленного узла.

Чтобы выполнить удаление, мы выполняем обычное удаление из дерева бинарного поиска, а затем выполняем снос родительского узла того узла, который был удален.

Обобщая, можно сказать, что скошенное дерево предоставляет нам самоизменяющуюся структуру – структуру, характеризующуюся тенденцией хранить узлы, к которым часто происходит обращение, вблизи верхушки дерева, в то время как узлы, к которым обращение происходит редко, перемещаются по направлению к листьям. В общем случае время обращения к часто посещаемым узлам будет меньше, а время обращения к редко посещаемым узлам – больше среднего. Важно отметить, что скошенное дерево не обладает никакими явными функциями балансировки, но практика свидетельствует, что снос способствует достаточно успешному поддержанию дерева в сбалансированном состоянии. В среднем время поиска в скошенном дереве пропорционально $O(\log(n))$.

Реализация класса скошенного дерева

Класс **TtdSplayTree** представляет собой простой производный класс класса **TtdBinarySearchTree**, в котором перекрыты методы **Delete**, **Find** и **Insert** и объявлены новые внутренние методы сноса и повышения ранга узла. Код интерфейса этого класса приведен в листинге 8.18.

Листинг 8.18. Интерфейс класса TtdSplayTree

```

type
  TtdSplayTree = class(TtdBinarySearchTree)
    private
    protected
      function stPromote(aNode : PtdBinTreeNode) : PtdBinTreeNode;
      procedure stSplay(aNode : PtdBinTreeNode);
    public
      procedure Delete(aItem : pointer); override;
      function Find(aKeyItem : pointer) : pointer; override;
      procedure Insert(aItem : pointer); override;
  end;

```

Перекрытый метод **Find** (см. листинг 8.19) реализует обычную операцию поиска в дереве бинарного поиска и, если узел найден, выполняет его снос к корневому узлу.

Листинг 8.19. Метод TtdSplayTree.Find

```

function TtdSplayTree.Find(aKeyItem : pointer) : pointer;
var
  Node : PtdBinTreeNode;
  ChildType : TtdChildType;
begin
  if bstFindItem(aKeyItem, Node, ChildType) then begin
    Result := Node^.btData;
    stSplay(Node);
  end
  else
    Result := nil;
  end;

```

Перекрытый метод **Insert** (см. листинг 8.20) реализует обычную операцию вставки в дерево бинарного поиска и выполняет снос нового узла к корневому узлу.

Листинг 8.20. Метод TtdSplayTree.Insert

```

procedure TtdSplayTree.Insert(aItem : pointer);
var
  ChildType : TtdChildType;
begin
  stSplay(bstInsertPrim(aItem, ChildType));
end;

```

Перекрытый метод **Delete** (см. листинг 8.21) реализует обычную операцию удаления из дерева бинарного поиска и выполняет снос родительского узла удаленного узла к корневому узлу.

Листинг 8.21. Метод TtdSplayTree.Delete

```

procedure TtdSplayTree.Delete(aItem : pointer);
var
  Node : PtdBinTreeNode;
  Dad : PtdBinTreeNode;
begin
  Node := bstFindNodeToDelete(aItem);
  Dad := Node^.btParent;
  FBinTree.Delete(Node);
  dec(FCount);
  if (Count <> 0) then
    stSplay(Dad);
end;

```

Эти три перекрытых метода достаточно просты для понимания, поскольку реальная обработка передается методу **stSplay**. Код реализации этого метода приведен в листинге 8.22.

Листинг 8.22. Метод TtdSplayTree.stSplay

```

procedure TtdSplayTree.stSplay(aNode : PtdBinTreeNode);
var
  Dad : PtdBinTreeNode;
  Grandad : PtdBinTreeNode;
  RootNode : PtdBinTreeNode;
begin
  { поскольку мы должны выполнять снос до тех пор, пока не будет достигнут
    корневой узел, сделать корневой узел локальной переменной — это несколько
    ускорит процесс }
  RootNode := FBinTree.Root;
  { если мы находимся в позиции корневого узла, никакой снос больше выполнять не
    требуется }
  if (aNode = RootNode) then
    Exit;
  { получить родительский и прародительский узлы }
  Dad := aNode^.btParent;
  if (Dad = RootNode) then
    Grandad := nil
  else
    Grandad := Dad^.btParent;
  { выполнять операции спаренного двустороннего и одностороннего поворота до
    тех пор, пока это возможно }
  while (Grandad <> nil) do begin
    { определить вид двойного повышения ранга, которое необходимо выполнить }
    if ((Grandad^.btChild[ctLeft] = Dad) and
      (Dad^.btChild[ctLeft] = aNode)) or
      ((Grandad^.btChild[ctRight] = Dad) and
      (Dad^.btChild[ctRight] = aNode)) then begin

```

```
    { выполнить повышение ранга посредством спаренного одностороннего поворота }
    stPromote(Dad);
    stPromote(aNode);
end
else begin
    { выполнить повышение ранга посредством спаренного двустороннего поворота }
    stPromote(stPromote(aNode));
end;
{ после того, как ранг повышен, необходимо получить новый родительски и
  прародительский узел }
RootNode := FBinTree.Root;
if (aNode = RootNode) then begin
    Dad := nil;
    Grandad := nil;
end
else begin
    Dad := aNode^.btParent;
    if (Dad = RootNode) then
        Grandad := nil
    else
        Grandad := Dad^.btParent;
    end;
end;
{ достижение этой точки свидетельствует, что узел находится либо в позиции
  корневого узла, либо на один уровень ниже него; выполнить последнее
  повышение ранга, если это необходимо }
if (Dad <> nil) then
    stPromote(aNode);
end;
```

Хотя эта подпрограмма выглядит сложной, она всего лишь повышает ранг переданного в нее узла до ранга корневого узла. Это делается с помощью ряда повышений ранга посредством спаренных односторонних или двусторонних поворотов: если узел, его родительский и прародительский узлы расположены на одной линии, выполняется повышение ранга за счет спаренного одностороннего поворота. В противном случае применяется повышение ранга за счет спаренного двустороннего поворота. Это процесс выполняется в цикле до тех пор, пока либо ранг узла не будет повышен до корневого, либо родительский узел данного узла не станет корневым. В последнем случае необходимо выполнить еще одно повышение ранга.

Код реорганизации при помощи повышения ранга представлен в методе **stPromote**, который показан в листинге 8.17.

Красно-черные деревья

Рассмотрев простые и спаренные двусторонние и односторонние повороты и ознакомившись с реорганизацией деревьев бинарного поиска за счет использования скошенных деревьев, пора приступить к исследованию соответствующего алгоритма балансировки.

Что должен делать алгоритм балансировки? В идеале он должен обеспечивать, чтобы длина пути от любого из листьев до корневого узла была одинаковой с точностью до единицы. На практике удовлетворить это строгое требование несколько затруднительно (AVL-деревья соответствуют этому определению, и их алгоритм балансировки удовлетворяет данному правилу). Поэтому мы определим какой-то алгоритм, который обеспечивает удовлетворение "менее строгого" требования, но не до такой степени "менее строгого", чтобы мы вернулись к тому, с чего начали.

В 1978 году Гюйба (Guibas) и Седжвик (Sedgewick) изобрели концепцию красно-черного дерева, удовлетворяющего такому умеренно нестрогому требованию. Красно-черные деревья (RB-деревья) – это структуры данных, используемые для реализации карт преобразования данных в библиотеке стандартных шаблонов C++ (C++ Standard Template Library). Красно-черный алгоритм предоставляет быстрый и эффективный метод балансировки дерева бинарного поиска, требующий для каждого узла не слишком много дополнительного объема памяти для хранения информации, необходимой для балансировки (в действительности для этого достаточно единственного дополнительного разряда).

Так что же собой представляют красно-черные деревья? Прежде всего, это дерево бинарного поиска, обладающее обычным простым алгоритмом поиска. Однако в красно-черном дереве каждый узел содержит определенную дополнительную информацию: каждый из них помечается как находящийся в одном из двух состояний. Эти два состояния называются *красным (red)* и *черным (black)*.

Понятно, что этот подход применяется не просто для раскрашивания узлов, и в действительности необходимо выполнить еще три правила:

1. Считается, что нулевые дочерние связи на периферии дерева указывают на другие узлы (естественно, несуществующие). Эти невидимые нулевые узлы называются *внешними узлами* и всегда окрашены в черный цвет.
2. Условие для черных узлов: все пути от корневого узла до каждого из внешних узлов содержат одинаковое количество черных узлов.
3. Условие для красных узлов: каждый красный узел, не являющийся корневым, имеет черный родительский узел.

Учитывая, что до сих пор при создании деревьев мы вполне спокойно игнорировали эти нулевые связи, правило 1 кажется несколько усложненным. Тем не менее, его выполнение требуется, чтобы легче было выполнить правило 2. Следовательно, дерево с единственным узлом содержит также два внешних узла,

являющиеся двумя нулевыми связями, исходящими из единственного реального узла (который называется внутренним). Второе правило – правило балансировки. Оно пытается поддерживать примерно одинаковую длину всех путей от корневого узла до каждого из внешних узлов. Эти пути будут различаться только количеством расположенных вдоль них красных узлов.

Набор простых красно-черных деревьев показан на рис. 8.6, при этом красные узлы изображены серыми квадратами (возможности одноцветной печати довольно-таки ограничены!), а внешние узлы – маленькими черными квадратами. Первое дерево (рисунок а) представляет пустое дерево – оно состоит всего из одного внешнего узла, который является черным – и, следовательно, по определению является красно-черным деревом. На примере второго и третьего деревьев (b и c) видно, что независимо от окрашивания корневого узла в красный или черный цвет, мы получаем красно-черное дерево. Эти деревья явно удовлетворяют всем трем правилам.

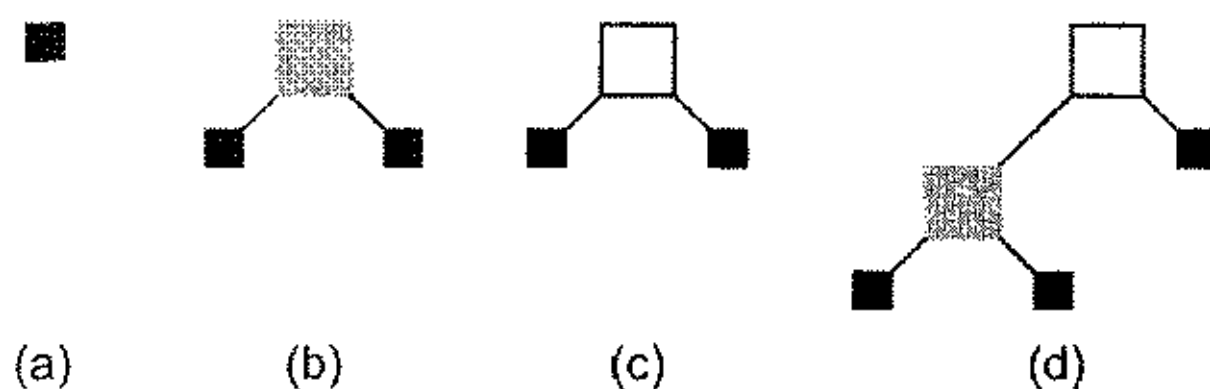


Рисунок 8.6. Набор простых красно-черных деревьев

Прежде чем продолжить, попытайтесь построить красно-черное дерево, содержащее два узла, корневой и левый дочерний, и три внешних узла (d). Выяснится, что в любом случае корневой узел должен быть окрашен в черный цвет, а его левый дочерний узел – в красный. Только такое окрашивание узлов позволяет удовлетворить все три правила.

Взглянем на это под другим углом. Посмотрите на рис. 8.7. Внутренние узлы этого дерева еще не окрашены. Можно ли их окрасить так, чтобы дерево удовлетворяло правилам 2 и 3? Никакого реального решения не существует. Невозможно окрасить внутренние узлы так, чтобы одновременно удовлетворить условия и для черных, и для красных узлов. Дерево, изображенное на рис. 8.7, не может быть красно-черным ни при каких условиях – и это хорошо, поскольку оно представляет начальную стадию вырождения дерева. Итак, важно усвоить следующий принцип: не все деревья могут быть окрашены в красный и черный цвета.

Фактически можно показать, что высота красно-черного дерева, содержащего n внутренних узлов, пропорциональна $\log n$. Иначе говоря, в самом худшем случае для поиска в красно-черном дереве потребуется время, которое пропорционально $O(\log(n))$. Именно к этому мы стре-

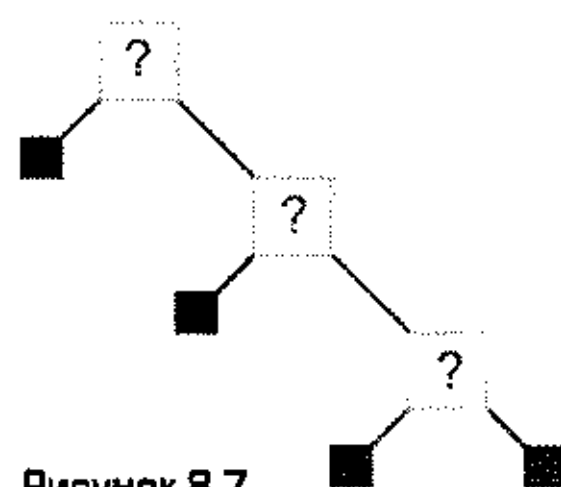


Рисунок 8.7. Дерево, которое не может быть окрашено в красный и черный цвета

мимся при использовании дерева бинарного поиска. Деревья, время поиска в которых пропорционально $O(n)$, являются вырожденными.

Вставка в красно-черное дерево

Теперь, когда мы ознакомились с правилами, определяющими структуру красно-черного дерева, возникает вопрос, как их использовать для вставки нового узла в красно-черное дерево? Начнем со знакомой операции, и выполним поиск узла. Если он будет найден, мы сигнализируем об ошибке (в красно-черном дереве дубликаты не допускаются, точно так же, как это имело место в стандартном дереве бинарного поиска). В противном случае необходимо обратиться к узлу, который можно использовать в качестве родительского узла нового узла, и определяющего, каким дочерним узлом должен быть новый узел. Теперь необходимо заменить внешний узел (вспомните, что это общее имя несуществующего узла на конце нулевой связи) новым узлом. Новый узел автоматически будет вставлен с двумя внешними узлами, которые в соответствии с правилом 1 окрашены в черный цвет. Но в какой цвет должен быть окрашен новый узел?

Начнем с того, что окрасим его в красный цвет. Как это сказывается на соблюдении правил, определенных для красно-черных деревьев? Во-первых, условие для черных узлов по-прежнему выполняется: мы заменяем черный внешний узел красным узлом и двумя черными внешними узлами. Путь от каждого из двух новых внешних узлов до корневого узла по-прежнему содержит столько же черных узлов, сколько и путь от замещенного внешнего узла до корневого узла. А как насчет условия, определенного для красных узлов? Продолжает ли оно выполняться? Возможно, да, а, возможно, и нет. Если новый узел является корневым, и, следовательно, не имеет родительского узла, созданное дерево остается красно-черным (в действительности, при желании новый узел можно было бы перекрасить в черный цвет, и при этом дерево осталось бы красно-черным). Если же новый узел не является корневым, он будет иметь родительский узел. Если этот родительский узел черный, правило 3, определенное для красных узлов, остается применимым, и дерево по-прежнему является красно-черным. Если родительский узел нового узла является корневым, то, чтобы дерево осталось красно-черным, достаточно при необходимости перекрасить родительский узел в черный цвет. (Фактически, в красно-черном дереве, если оба дочерних узла корневого узла являются черными, корневым узел может быть как красным, так и черным – это никак не сказывается на соблюдении правил.)

Если родительский узел нового узла не является корневым и окрашен в красный цвет, мы получаем два следующие друг за другом красные узла. При этом правило, определенное для красных узлов, нарушается, и для воссоздания красно-черного дерева эту проблему придется решить.

В этой ситуации возможны несколько вариантов. Чтобы было проще понять происходящее, вначале присвоим имена ряду узлов. После этого можно будет опи-

сать некоторые преобразования, которые потребуется выполнить, чтобы вернуть дерево в красно-черное состояние.

Назовем новый узел s (от son – сын), его родительский узел d (от dad – отец), родительский узел родительского узла g (granddad – дед), а родственник с родительским узлом – u (uncle – дядя). Непосредственно после добавления узла s возникает следующая ситуация: узлы s и d являются красными (что является нарушением правила 2), узел g должен быть черным (согласно правилу 2), а узел u может быть либо красным, либо черным.

Вначале предположим, что узел u является черным. Для достижения поставленной цели достаточно выполнить либо одиночный поворот, либо спаренный двусторонний поворот, а затем перекрасить некоторые узлы. В первом случае, который на рис. 8.8 представлен первым преобразованием, мы выполняем поворот узла d вправо на место узла g , чтобы g стал дочерним узлом узла d . Затем мы перекрашиваем узел d в черный цвет, а g – в красный. Во втором случае (нижнее преобразование на рис. 8.8) мы выполняем спаренный двусторонний поворот, чтобы поместить узел s на место g , а затем перекрашиваем узел s в черный цвет, а g – в красный. Обратите внимание, что абсолютно не важно, является ли узел u внешним или внутренним; достаточно, чтобы он был черным.

Естественно, возможны еще два случая, представляющие собой зеркальное отражение рассмотренных, однако мы не будем их рассматривать. На рисунке 8.8 легко видеть, что теперь условие, определенное для красных узлов, удовлетворено, и что операции поворота и перекрашивания не нарушают условие, определенное для черных узлов.

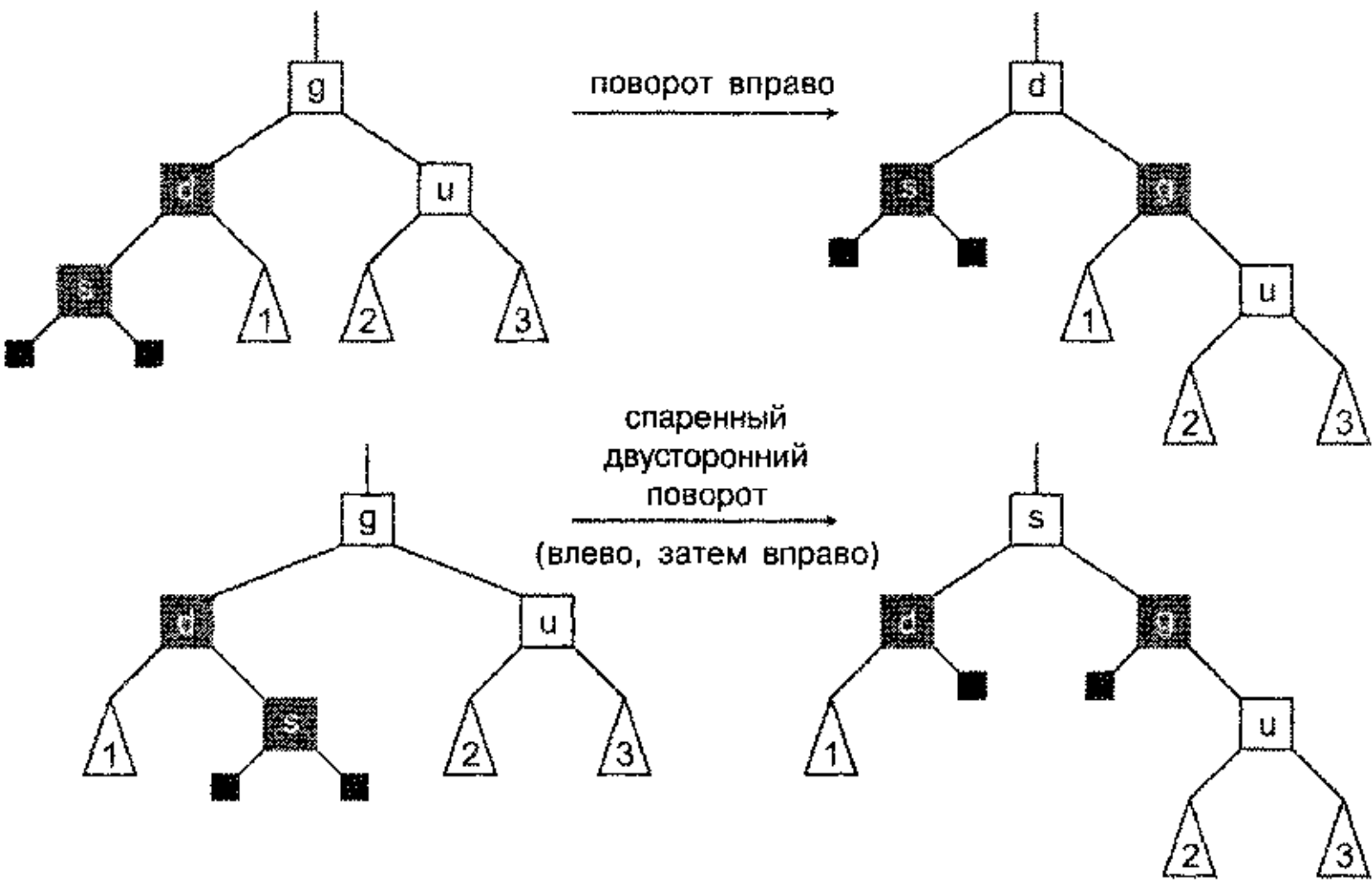


Рисунок 8.8. Балансировка после вставки: два простых случая

Этот случай был простым. Теперь рассмотрим более сложный. Предположим, что узел *u*, дядя нового узла, также окрашен в красный цвет. Первый шаг прост: мы перекрашиваем узлы *d* и *u* в черный цвет, а *g* в красный. Условие для черных узлов по-прежнему выполняется, но, похоже, мы ухудшили общую ситуацию, поскольку условие, определенное для красных узлов, перестало выполняться. Вместо того чтобы признать, что узел *s* нарушает условие, определенное для красных узлов, мы предположили, каким мог бы быть узел *g*. В конце концов, родительский узел узла *g* мог бы быть и красным. Иначе говоря, в действительности эта операция перекрашивания не решает никаких проблем. Мы просто отложили решение проблемы на неопределенный срок. Но действительно ли ситуация ухудшилась? Посмотрите, что мы сделали: мы переместили проблемный узел вверх по дереву. Перемещение вверх ограничено в пространстве, поскольку со временем мы натолкнемся на корневой узел.

Итак, перенесем свое внимание двумя уровнями выше, примем, что узел *g* является новым узлом *s*, и посмотрим, нарушили ли мы какие-либо правила. Иначе говоря, снова применим рассмотренный алгоритм, но на этот раз начнем рассмотрение с узла *g*. Два возможных случая показаны на рис. 8.9 (естественно, могут существовать и два случая, являющиеся зеркальными отражениями представленных, но они не показаны). В обоих результирующих деревьях узел *g* помечен тремя восклицательными знаками, указывающими, что он может нарушать одно из двух правил, и что необходимо продолжать процесс, снова повторяя действия алгоритма.

Не прибегая к подробным математическим выкладкам, отметим, что подобно случаю применения простого бинарного дерева, алгоритм вставки в красно-черное дерево является алгоритмом типа $O(\log(n))$, хотя в этом случае постоянный коэффициент имеет большее значение, поскольку приходится учитывать возможные повороты и повышение ранга узлов.

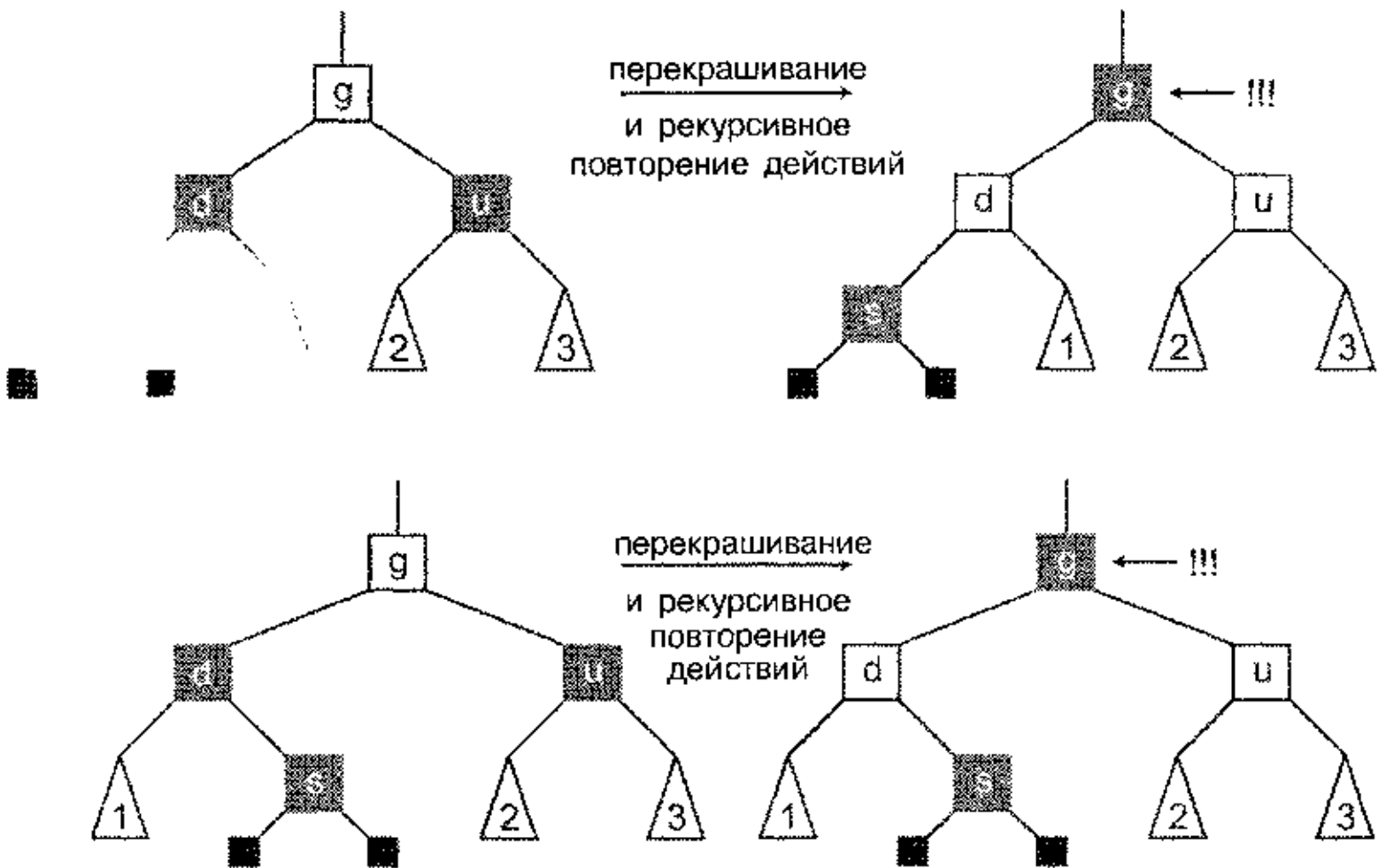


Рисунок 8.9. Балансировка после вставки: два рекурсивных случая

Код реализации этого алгоритма вставки и балансировки приведен в листинге 8.23. Метод содержит внутренний цикл, выход из которого выполняется, когда баланс дерева восстановлен. В начале цикла предполагается, что балансировка дерева должна быть выполнена в данном цикле, и что перемещение по дереву вверх должно выполняться только в том случае, если мы уверены, что снова будем выполнять цикл. В остальном приведенный код служит достаточно точным представлением алгоритма вставки в красно-черное дерево. Единственный неприятный момент – необходимость поддержания информации о том, являются ли определенные узлы левыми или правыми дочерними узлами своих родительских узлов.

Листинг 8.23. Вставка в красно-черное дерево

```
procedure TtdRedBlackTree.Insert(aItem : pointer);  
var  
    Node : PtdBinTreeNode;  
    Dad : PtdBinTreeNode;  
    Grandad : PtdBinTreeNode;  
    Uncle : PtdBinTreeNode;  
    OurType : TtdChildType;  
    DadsType : TtdChildType;  
    IsBalanced : boolean;  
begin  
    {вставить новый элемент, вернуться к вставленному узлу и его связям с  
    родительским узлом}  
    Node := bstInsertPrim(aItem, OurType);  
    {окрасить его в красный цвет}  
    Node^.btColor := rbRed;  
    {продолжать применение в цикле алгоритмов балансировки при вставке в красно-  
    черное дерево до тех пор, пока дерево не окажется сбалансированным}  
    repeat  
        {предположим, что дерево сбалансировано}  
        IsBalanced := true;  
        {если узел является корневым, задача выполнена и дерево сбалансировано,  
        поэтому будем считать, что мы находимся не в корневом узле}  
        if (Node<>FBinTree.Root) then begin  
            {поскольку мы находимся не в корневом узле, необходимо получить родитель-  
            ский узел данного узла}  
            Dad := Node^.btParent;  
            {если родительский узел черный, задача выполнена и дерево сбалансировано,  
            поэтому будем считать, что родительский узел красный}  
            if (Dad^.btColor = rbRed) then begin  
                {если родительский узел является корневым, достаточно перекрасить его в  
                черный цвет, и задача будет выполнена}  
                if (Dad = FBinTree.Root) then  
                    Dad^.btColor := rbBlack  
                {в противном случае родительский узел, в свою очередь, имеет родительс-  
                кий узел}
```

else begin

{получить прародительский узел (он должен быть черным) и перекрасить его в красный цвет}

Grandad := Dad^.btParent;

Grandad^.btColor := rbRed;

{получить узел, соответствующий понятию дяди}

if (Grandad^.btChild[ctLeft] = Dad) **then begin**

DadsType := ctLeft;

Uncle := Grandad^.btChild[ctRight];

end

else begin

DadsType := ctRight;

Uncle := Grandad^.btChild[ctLeft];

end;

{если дядя тоже имеет красный цвет (обратите внимание, что он может быть нулевым!), окрасить родительский узел в черный цвет, дядю в черный цвет и повторить процесс, начиная с прародительского узла}

if IsRed(Uncle) **then begin**

Dad^.btColor := rbBlack;

Uncle^.btColor := rbBlack;

Node := Grandad;

IsBalanced := false;

end

{в противном случае дядя окрашен в черный цвет}

else begin

{если текущий узел имеет такие же отношения со своим родительским узлом, какие его родительский узел имеет с прародительским (т.е. они оба являются либо левыми, либо правыми дочерними узлами), нужно окрасить родительский узел в черный цвет и повысить его ранг. Задача выполнена}

OurType := GetChildType(Node);

if (OurType = DadsType) **then begin**

Dad^.btColor := rbBlack;

rbtPromote(Dad);

end

{в противном случае необходимо окрасить узел в черный цвет и повысить его ранг посредством применения спаренного двустороннего поворота; задача выполнена}

else begin

Node^.btColor := rbBlack;

rbtPromote(rbtPromote(Node));

end;

end;

end;

end;

end;

until IsBalanced;

end;

Необходимо принимать во внимание один небольшой нюанс: следует проверять цвета узлов. Некоторые из узлов, которые мы будем проверять, будут внешними, т.е. нулевыми. Для повышения читабельности кода я написал небольшую подпрограмму **IsRed**, которая выполняет проверку на наличие нулевого узла (возвращая значение **false**), прежде чем выполнять проверку поля цвета узла.

Листинг 8.24. Интеллектуальная подпрограмма IsRed

```
function IsRed(aNode : PtdBinTreeNode) : boolean;  
begin  
  if (aNode = nil) then  
    Result := false  
  else  
    Result := aNode^.btColor = rbRed;  
end;
```

Удаление из красно-черного дерева

По сравнению со вставкой, удаление из красно-черного дерева сопряжено с множеством особых случаев и его может быть трудно отследить.

Как обычно, при использовании деревьев бинарного поиска, начнем с поиска узла, который требуется удалить. Как и ранее, возможны три начальных случая: узел не имеет дочерних узлов (или, применяя терминологию, принятую в красно-черных деревьях, оба его дочерних узла являются внешними); узел имеет один реальный дочерний узел и один внешний дочерний узел; и, наконец, узел имеет два реальных дочерних узла. Удаление узла выполняется так же, как это делалось в стандартном неокрашенном дереве бинарного поиска.

Теперь рассмотрим эти три случая с точки зрения красно-черных деревьев. Первый случай – узел с двумя внешними дочерними узлами (т.е. с нулевыми связями). В соответствии с правилом 1, эти два дочерних узла считаются черными. Однако узел, который нужно удалить, может быть красным или черным. Предположим, что он красный. Удаляя его, мы заменяем дочернюю связь родительского узла нулевым указателем – иначе говоря, внешним черным узлом. Однако мы не изменили количество черных узлов от нового внешнего узла до корневого узла, по сравнению с существовавшими до этого двумя путями. Следовательно, правило 2 по-прежнему выполняется. Очевидно, что правило 3 также не нарушается (мы удаляем красный узел, поэтому никакие проблемы в отношении соблюдения этого правила не возникают). Таким образом, после удаления бинарное дерево остается красно-черным. Эта возможность представлена первым преобразованием на рис. 8.10.

А как насчет второй возможности (когда удаляемый узел окрашен в черный цвет)? Что ж, в этом случае правило 2, сформулированное для черных узлов, неизбежно нарушается. Количество черных узлов в пути до корневого узла уменьшается на 1. Возникающая в результате такого преобразования проблема проил-

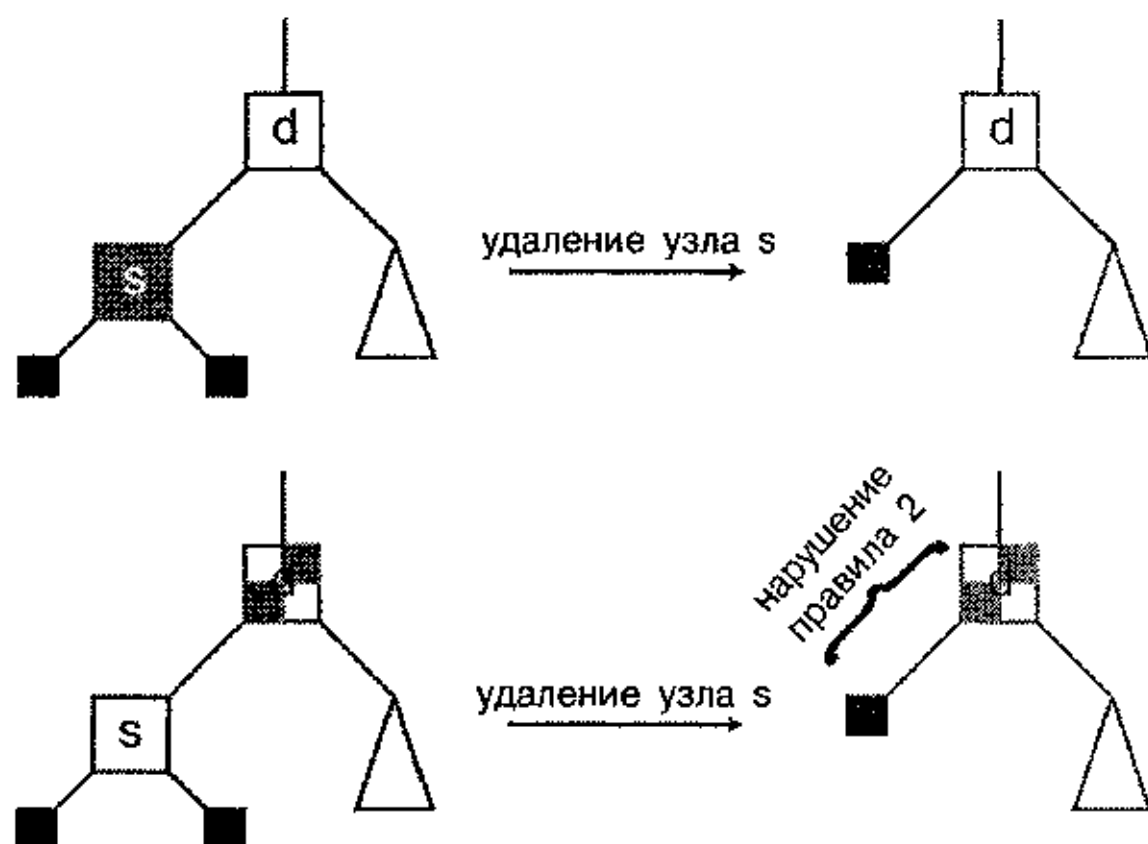


Рисунок 8.10.

Удаление узла,
имеющего два
внешних дочерних узла

люстрирована на нижней части рисунка 8.10. Мысленно заложим в этом месте закладку и рассмотрим другие случаи.

Второй случай удаления – удаление узла, который имеет один реальный дочерний узел и один внешний дочерний узел. Предположим, что удаляемый узел является красным. Его единственный реальный дочерний узел будет черным. Можно удалить узел и заменить его единственным дочерним узлом. Это не приведет к нарушению правила 2, – в конечном счете, мы удаляем красный узел, – а правило 3 в данном случае не затрагивается, следовательно, дерево остается красно-черным. Этот случай представлен первым преобразованием на рис. 8.11.

Теперь предположим, что удаляемый узел является черным. Единственный дочерний узел может быть красным или черным. Предположим, что он красный. Правило 2 неизбежно нарушается, поскольку мы удаляем черный узел, а правило 3 может быть нарушено, так как новый родительский узел красного дочернего узла может также оказаться красным. Однако этот случай достаточно прост: нужно просто перекрасить красный дочерний узел в черный цвет при перемещении его вверх для замещения удаленного узла. В результате этого простого действия мы снова добиваемся выполнения правила 2, а правило 3 в данном случае не затрагивается. Дерево снова становится красно-черным. Тот случай представлен вторым преобразованием, показанным на рис. 8.11.

Однако случай, когда единственный дочерний узел является черным, сложнее (третье преобразование на рис. 8.11). Что ж, вспомним о существовании этой проблемы и рассмотрим третий, он же и последний, случай удаления.

В действительности заключительный случай удаления из дерева бинарного поиска не отличается от двух уже рассмотренных, поскольку, если помните, мы меняем местами узел, который нужно было бы удалить, с наибольшим узлом из левого дочернего дерева, а затем удаляем этот второй узел вместо первого. Этот второй узел будет соответствовать либо первому, либо второму случаю удаления, поэтому две проблемы, решение которых мы отложили, придется решать раньше, чем мы полагали.

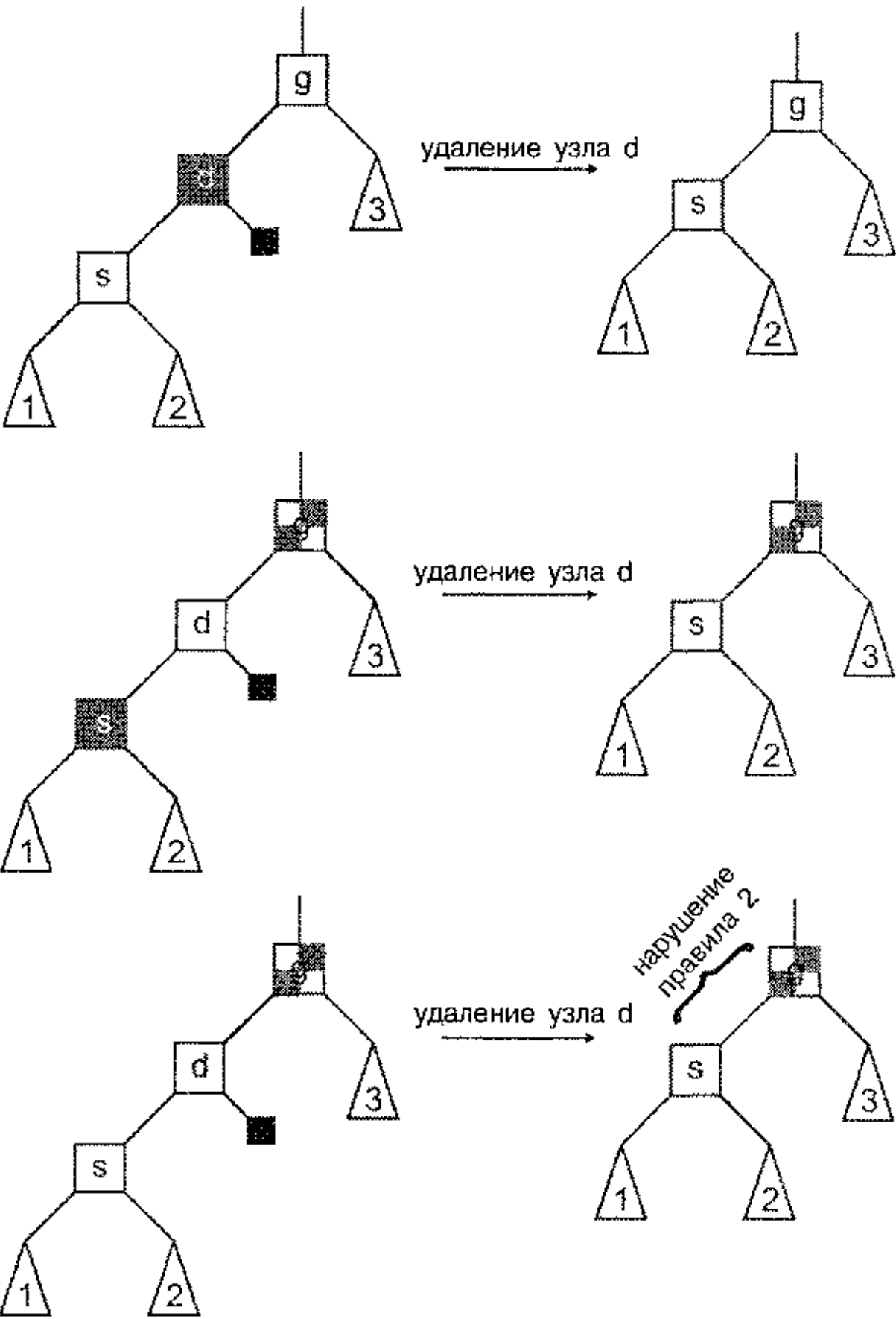


Рисунок 8.11.
Удаление узла,
который имеет один
внутренний и один
внешний дочерний узел

Кратко напомним, в чем они состоят. Удаляемый узел имеет, по меньшей мере, один внешний узел. Если удаляемый узел красный, то его второй дочерний узел должен быть черным (конечно, он может быть внешним узлом, поскольку внешние узлы автоматически окрашиваются в черный цвет). Можно удалить узел, заменить его этим вторым дочерним узлом, и в результате дерево останется красно-черным. Если удаляемый узел является черным и имеет один красный внутренний дочерний узел, то можно удалить узел и заменить его дочерним узлом, окрасив его в черный цвет.

Однако если удаляемый узел черный и имеет, по меньшей мере, один внешний дочерний узел, а другой дочерний узел либо черный, либо внешний, то мы сталкиваемся с двумя ранее описанными проблемами. Повышение ранга дочернего узла в результате удаления приводит к нарушению правила 2 (назовем этот узел *нарушающим узлом*). Эти случаи представлены последними преобразованиями, изображенными на рисунках 8.10 и 8.11.

Попытаемся свести оба случая к одному. Мы должны принимать во внимание родительский и братский узлы нарушающего узла и два дочерних узла братского

узла (узлы-племянники). Обратите внимание, что можно принять *наличие* у братского узла двух дочерних узлов (т.е. считать, что братский узел не является внешним). Почему? Рассмотрим исходное дерево. Оно было красно-черным. Следовательно, все пути, проходящие через удаленный и родительский узлы, имели то же количество черных узлов, что и пути, проходящие через братский и родительский узлы. Поскольку мы предполагаем, что родительский узел черный, а удаленный узел и заменивший его дочерний узел также были черными, то и все пути, проходящие через братский узел, должны содержать, по меньшей мере, два черных узла. Отсюда следует, что, как минимум, братский узел является черным и имеет два черных дочерних узла.

Как бы там ни было, рассмотрим братский узел. Последующие рассуждения упростятся, если принять, что братский узел является черным. Если это не так, нужно перекрасить родительский узел в красный цвет, а братский – в черный, после чего повернуть родительский узел и повысить ранг братского узла. Результирующее дерево будет красно-черным, если не обращать внимания на исходный нарушающий узел, но в нем братский узел гарантированно является черным. Таким образом, в дальнейшем будем считать, что братский узел окрашен в черный цвет. (Обратите внимание, что если бы братский узел был красным, то его дочерние узлы должны были быть черными и, более того, чтобы правило 2 изначально выполнялось, они должны были бы иметь собственные дочерние узлы. Следовательно, это преобразование сохраняет существование братского узла с дочерними узлами и красно-черное состояние дерева.)

Вначале необходимо рассмотреть случай, когда нарушающий узел имеет черный родительский узел и два черных узла-племянника. Перекрашивание братского узла в красный цвет приводит к перемещению проблемной области вверх к родительскому узлу, и можно просто повторить весь алгоритм применительно к этому узлу, как к нарушающему правилу. Эта возможность показана на рис. 8.12.

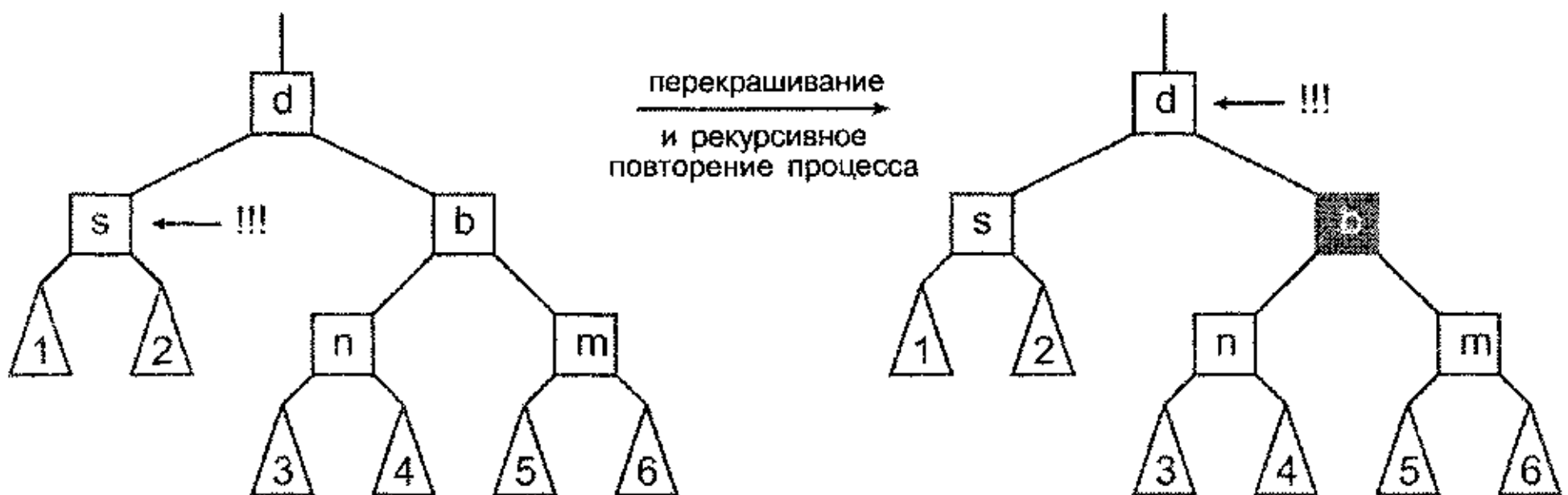


Рисунок 8.12. Балансировка после удаления: первый случай

Второй возможный случай – существование красного родительского узла и двух черных узлов-племянников. Этот случай даже проще предыдущего: нужно перекрасить родительский узел в черный цвет, а братский – в красный. Путь, проходящий через нарушающий узел, снова имеет требуемое количество черных узлов (тем самым удовлетворяя правило 2). Это же можно сказать и по поводу пути, проходящего через братский узел (правило 2 снова выполняется). Только что окрашенный в красный цвет узел имеет черный родительский узел, следовательно, правило 3 не нарушается. Стало быть, мы снова получаем красно-черное дерево. Этот случай показан на рис. 8.13.

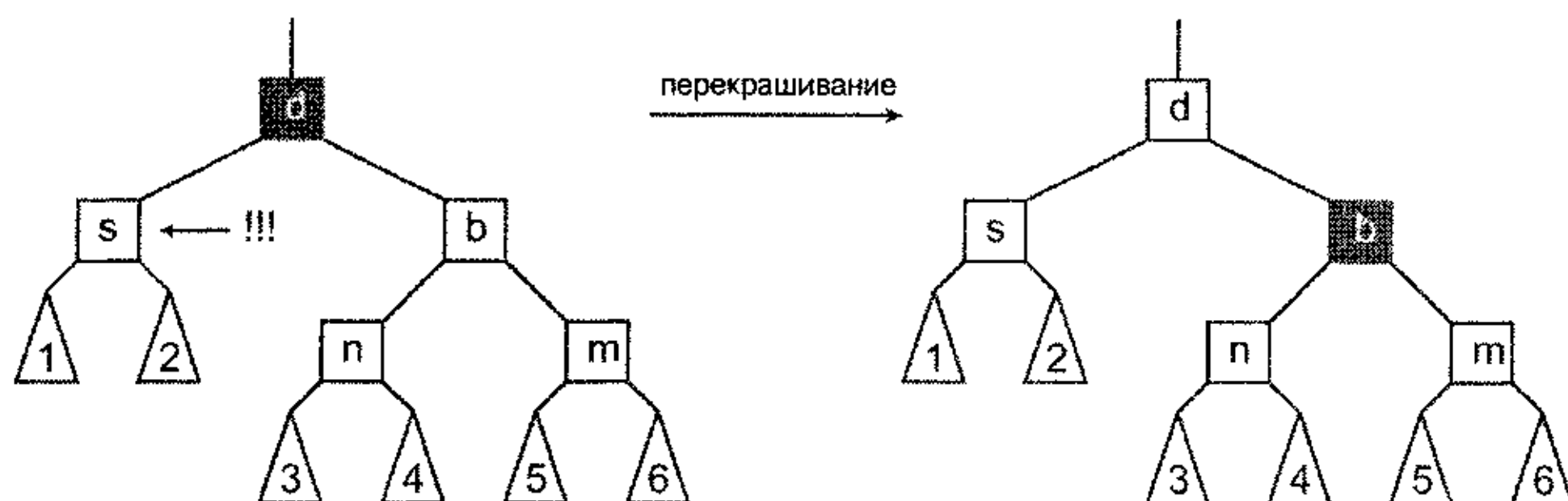


Рисунок 8.13. Балансировка после удаления: второй случай

Теперь предположим, что противоположный по отношению к нарушающему узлу узел-племянник является красным. (Иначе говоря, если нарушающий узел – это левый дочерний узел родительского узла, речь идет о правом узле-племяннике, а если нарушающий узел – правый дочерний узел, речь идет о левом узле-племяннике.) Перекрасим этот узел-племянник в черный цвет. Окрасим братский узел в цвет родительского узла (первоначальный цвет родительского узла не имеет значения), а родительский узел – в черный цвет. Затем повернем родительский узел и повысим ранг братского узла. Тщательно проанализируем эту ситуацию, глядя на рис. 8.14. Вначале проверим выполнение правила 3: очевидно, что мы не ввели никаких новых красных узлов, следовательно, можно быть уверенным, что это правило выполняется. Теперь рассмотрим выполнение правила 2. Все пути, проходящие через нарушающий узел, содержат дополнительный черный узел, что ведет к устранению проблемы, возникшей в результате удаления исходного узла. Все пути, проходящие через дочерние деревья 5 и 6, также содержат то же количество черных узлов, что и ранее. Следовательно, во всех случаях правило 2 выполняется, и результирующее дерево снова является красно-черным.

Теперь рассмотрим последний случай. Предположим, что противоположный узел-племянник окрашен в черный цвет, но второй узел этой же степени родства является красным. На этот раз нужно выполнить спаренный двусторонний поворот. Вначале мы окрашиваем узел-племянник в цвет родительского узла (как и в предыдущем случае, первоначальный цвет родительского узла значения не имеет),

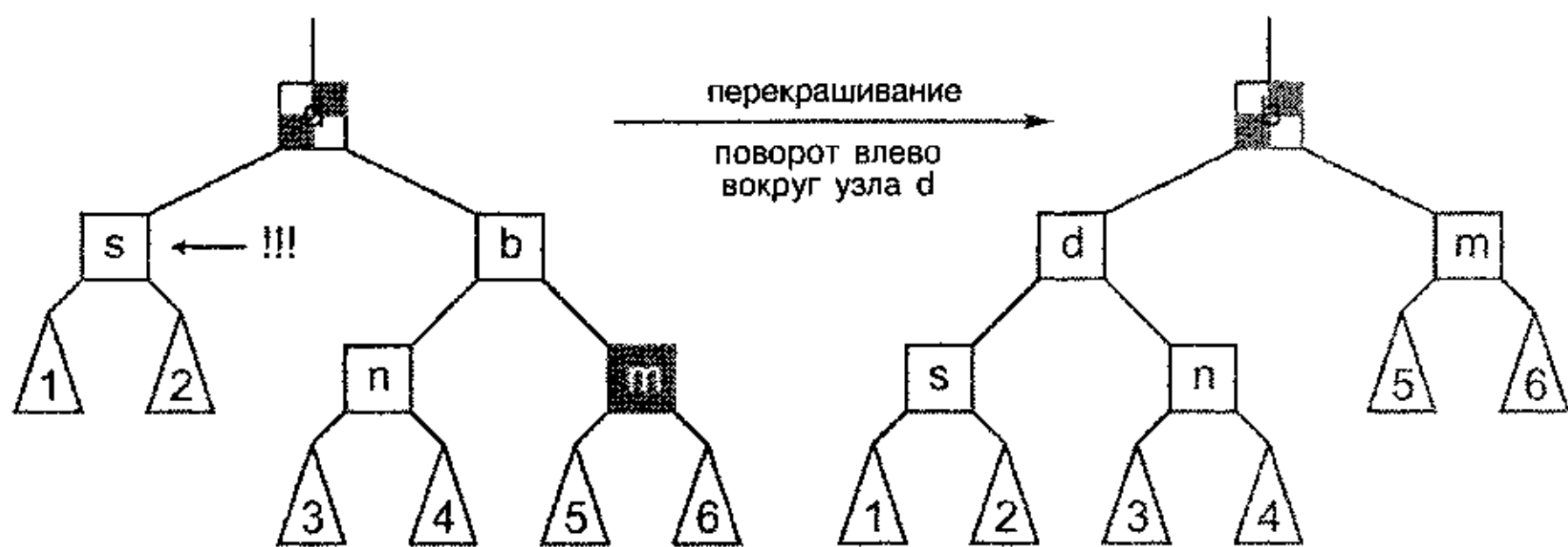


Рисунок 8.14. Балансировка после удаления: третий случай

а затем перекрашиваем родительский узел в черный цвет. Далее мы поворачиваем братский узел, чтобы повысить ранг узла-племянника, а затем поворачиваем родительский узел, чтобы снова повысить ранг узла-племянника. Это преобразование показано на рис. 8.15. В любом случае это не ведет к непреднамеренному нарушению правила 3: мы не ввели никаких новых красных узлов. Теперь что касается правила 2 – все пути, проходящие через нарушающий узел, содержат один дополнительный черный узел, следовательно, ранее описанная проблема устранена. Все пути, проходящие через дочернее дерево 3, по-прежнему содержат одинаковое количество черных узлов. Аналогично, во всех путях, проходящих через дочерние деревья 4, 5 и 6, не был вставлен или удален какой-либо дополнительный черный узел, следовательно, правило 3 по-прежнему выполняется. Дерево снова оказывается красно-черным.

Если нарушающий узел удастся переместить до позиции корневого узла, создается предельная ситуация. В этом случае нарушающий узел не имеет родительского узла и, следовательно, не может иметь братский узел. При этом нарушающий узел больше не представляет проблемы.

Конечно, все рассмотренные случаи имеют аналоги, представляющие их зеркальные отражения, но при этом анализ каждого из случаев удаления остается тем же. При написании кода подпрограммы удаления нужно будет убедиться, что мы правильно отразили как левые, так и правые варианты расположения узлов.

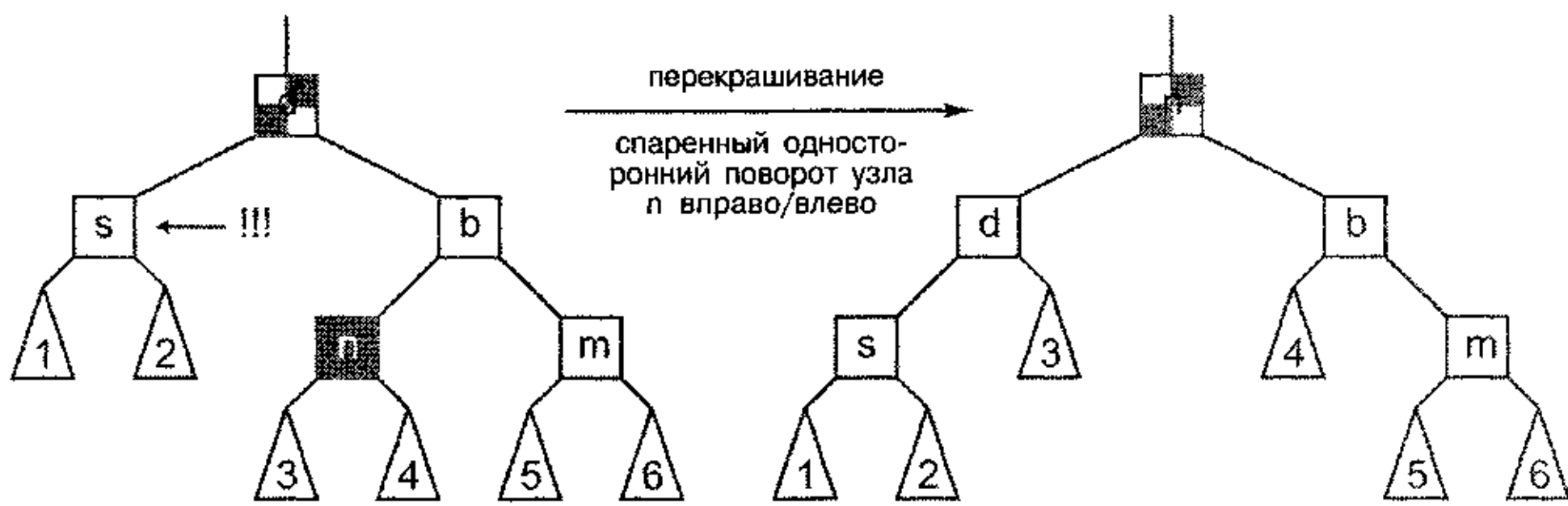


Рисунок 8.15. Балансировка после удаления: заключительный случай

Итак, мы рассмотрели все возможности. При этом использовались два рекурсивных шага или, точнее, два шага, которые требовали дальнейших усилий по балансировке. Первый – когда братский узел был красным, и его нужно было сделать черным. Второй – когда родительский, братский и узлы-племянники были черными. Существовали еще три случая: родительский узел был красным, а братский узел и узлы-племянники были черными; братский узел был черным, а дальний узел-племянник красным (цвет родительского узла и ближайшего узла-племянника "не имели значения"); и, наконец, случай, когда братский узел был черным, дальний узел-племянник черным, а ближайший узел-племянник красным. Если вы еще раз взглянете на рисунки 8.12, 8.13, 8.14 и 8.15, то убедитесь, что мы рассмотрели все варианты.

Опуская математические выкладки, отметим, что алгоритм удаления из красно-черного дерева является алгоритмом типа $O(\log(n))$, хотя постоянный коэффициент времени больше, чем в случае простого бинарного дерева.

Операция удаления узла из красно-черного дерева реализуется с помощью кода, представленного в листинге 8.25.

Листинг 8.25. Удаление из красно-черного дерева

```

procedure TtdRedBlackTree.Delete(aItem : pointer);
var
  Node : PtdBinTreeNode;
  Dad : PtdBinTreeNode;
  Child : PtdBinTreeNode;
  Brother : PtdBinTreeNode;
  FarNephew : PtdBinTreeNode;
  NearNephew : PtdBinTreeNode;
  IsBalanced : boolean;
  ChildType : TtdChildType;
begin
  {выполнить поиск узла, который нужно удалить; этот узел будет иметь
   единственный дочерний узел}
  Node := bstFindNodeToDelete(aItem);
  {если узел красный или является корневым, его можно безнаказанно удалить}
  if (Node^.btColor = rbRed) or (Node = FBinTree.Root) then begin
    FBinTree.Delete(Node);
    dec(FCount);
    Exit;
  end;
  {если единственный дочерний узел является красным, перекрасить его в черный
   цвет и удалить узел}
  if (Node^.btChild[ctLeft] = nil) then
    Child := Node^.btChild[ctRight]
  else
    Child := Node^.btChild[ctLeft];

```

```

if IsRed(Child) then begin
  Child^.btColor := rbBlack;
  FBinTree.Delete(Node);
  dec(FCount);
  Exit;
end;
{на этом этапе узел, который нужно удалить, – узел Node; он является черным и
известно, что дочерний узел Child, который его заменит, является черным (а
также может быть нулевым!) и что существует родительский узел узла Node
(который вскоре станет родительским узлом узла Child); братский узел узла
Node также существует в соответствии с правилом, сформулированным для
черных узлов}
{если узел Child является нулевым, необходимо несколько упростить выполнение
цикла и определить родительский и братский узлы и определить, является ли
узел Node левым дочерним узлом}
if (Child = nil) then begin
  Dad := Node^.btParent;
  if (Node = Dad^.btChild[ctLeft]) then begin
    ChildType := ctLeft;
    Brother := Dad^.btChild[ctRight];
  end
  else begin
    ChildType := ctRight;
    Brother := Dad^.btChild[ctLeft];
  end;
end
else begin
  {следующие три строки предназначены просто для введения в заблуждение
  компилятора и предотвращения вывода ряда ложных предупреждений}
  Dad := nil;
  Brother := nil;
  ChildType := ctLeft;
end;
{удалить узел – он больше не нужен}
FBinTree.Delete(Node);
dec(FCount);
Node := Child;
{циклически применять алгоритмы балансировки при удалении из красно-черного
дерева до тех пор, пока дерево не окажется сбалансированным}
repeat
  {предположим, что дерево сбалансировано}
  IsBalanced := true;
  {если узел является корневым, балансировка выполнена, поэтому предположим,
  что это не так}
  if (Node <> FBinTree.Root) then begin
    {получить родительский и братский узлы}
    if (Node <> nil) then begin
      Dad := Node^.btParent;

```



```
if (Node = Dad^.btChild[ctLeft]) then begin
    ChildType := ctLeft;
    Brother := Dad^.btChild[ctRight];
end
else begin
    ChildType := ctRight;
    Brother := Dad^.btChild[ctLeft];
end;
end;
{ нам требуется наличие черного братского узла, поэтому если в настоящий
  момент братский узел окрашен в красный цвет, окрасить родительский узел в
  красный цвет, братский узел в черный цвет и повысить ранг братского узла;
  затем снова повторить цикл}
if (Brother^.btColor = rbRed) then begin
    Dad^.btColor := rbRed;
    Brother^.btColor := rbBlack;
    rbtPromote(Brother);
    IsBalanced := false;
end
{ в противном случае братский узел является черным}
else begin
    { получить узлы-племянники, помеченные как дальний и ближний}
    if (ChildType = ctLeft) then begin
        FarNephew := Brother^.btChild[ctRight];
        NearNephew := Brother^.btChild[ctLeft];
    end
    else begin
        FarNephew := Brother^.btChild[ctLeft];
        NearNephew := Brother^.btChild[ctRight];
    end;
    { если дальний узел-племянник является красным (обратите внимание, что
      он может быть нулевым), окрасить его в черный цвет, братский узел в
      цвет родительского узла, а родительский узел в красный цвет, а затем
      повысить ранг братского узла; задача выполнена}
    if IsRed(FarNephew) then begin
        FarNephew^.btColor := rbBlack;
        Brother^.btColor := Dad^.btColor;
        Dad^.btColor := rbBlack;
        rbtPromote(Brother);
    end
    { в противном случае дальний узел-племянник является черным}
    else begin
        { если ближний узел-племянник является красным (обратите внимание, что
          он может быть нулевым), окрасить его в цвет родительского узла, роди-
          тельский узел в черный цвет и повысить ранг узла-племянника посред-
          ством спаренного двустороннего поворота; в этом случае задача
          выполнена}
```

```

if IsRed(NearNephew) then begin
    NearNephew^.btColor := Dad^.btColor;
    Dad^.btColor := rbBlack;
    rbtPromote(rbtPromote(NearNephew));
end
{ в противном случае ближний узел-племянник является также черным}
else begin
    { если родительский узел красный, окрасить его в черный цвет, а
    братский узел в красный, в результате чего задача будет выполнена}
    if (Dad^.btColor = rbRed) then begin
        Dad^.btColor := rbBlack;
        Brother^.btColor := rbRed;
    end
    { в противном случае родительский узел красный: окрасить братский
    узел в красный цвет и начать балансировку с родительского узла}
    else begin
        Brother^.btColor := rbRed;
        Node := Dad;
        IsBalanced := false;
    end;
end;
end;
end;
end;
until IsBalanced;
end;

```

За исключением перекрытых методов **Insert** и **Delete**, класс **TtdRedBlackTree** не представляет особого интереса. Код интерфейса и дополнительного внутреннего метода, выполняющего повышение ранга узла, приведен в листинге 8.26.

Листинг 8.26. Класс **TtdRedBlack** и метод повышения ранга узла

```

type
    TtdRedBlackTree = class(TtdBinarySearchTree)
    private
    protected
        function rbtPromote(aNode : PtdBinTreeNode) : PtdBinTreeNode;
    public
        procedure Delete(aItem : pointer); override;
        procedure Insert(aItem : pointer); override;
    end;
function TtdRedBlackTree.rbtPromote(aNode : PtdBinTreeNode)
    : PtdBinTreeNode;

var
    Parent : PtdBinTreeNode;
begin
    { пометить родительский узел узла, ранг которого повышается}
    Parent := aNode^.btParent;

```

{ в обоих случаях существует 6 связей, которые необходимо разорвать и перестроить: связь узла с его дочерним узлом и противоположная связь; связь узла с его родительским узлом и противоположная связь; и связь родительского узла с его родительским узлом и противоположная связь; обратите внимание, что дочерний узел данного узла может быть нулевым }

{ повысить ранг левого дочернего узла, т.е. выполнить поворот родительского узла вправо }

```
if (Parent^.btChild[ctLeft] = aNode) then begin
  Parent^.btChild[ctLeft] := aNode^.btChild[ctRight];
  if (Parent^.btChild[ctLeft] <> nil) then
    Parent^.btChild[ctLeft]^btParent := Parent;
    aNode^.btParent := Parent^.btParent;
  if (aNode^.btParent^.btChild[ctLeft] = Parent) then
    aNode^.btParent^.btChild[ctLeft] := aNode
  else
    aNode^.btParent^.btChild[ctRight] := aNode;
    aNode^.btChild[ctRight] := Parent;
    Parent^.btParent := aNode;
```

end

{ повысить ранг правого дочернего узла, т.е. выполнить поворот родительского узла влево }

else begin

```
  Parent^.btChild[ctRight] := aNode^.btChild[ctLeft];
  if (Parent^.btChild[ctRight] <> nil) then
    Parent^.btChild[ctRight]^btParent := Parent;
    aNode^.btParent := Parent^.btParent;
  if (aNode^.btParent^.btChild[ctLeft] = Parent) then
    aNode^.btParent^.btChild[ctLeft] := aNode
  else
    aNode^.btParent^.btChild[ctRight] := aNode;
    aNode^.btChild[ctLeft] := Parent;
    Parent^.btParent := aNode;
```

end;

{ вернуть узел, ранг которого был повышен }

Result := aNode;

end;

Исходный код класса **TtdRedBlackTree** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDBinTre.pas**.

Резюме

В этой главе мы рассмотрели бинарные деревья – важную структуру данных, которая может использоваться во многих прикладных приложениях. Мы рассмотрели стандартное бинарное дерево, а затем перешли к исследованию его сортированной разновидности – дереву бинарного поиска.

В ходе рассмотрения дерева бинарного поиска мы ознакомились с проблемой, которая может возникнуть во время вставки и удаления – проблемой вырождения дерева, – именно в это связи мы исследовали способы ее устранения. Первое решение, скошенное дерево, предоставляет хорошую возможность, несмотря на то, что при этом эффективность вставки и удаления лишь в среднем, а не всегда, описывается соотношением $O(\log(n))$. Однако эта разновидность дерева представляет собой приемлемый компромисс между стандартным деревом бинарного поиска и таким действительно сбалансированным деревом, как красно-черное дерево.

Воспользовавшись красно-черным деревом, мы, наконец, получили полное дерево бинарного поиска, имеющее встроенные алгоритмы балансировки как для вставки, так и для удаления.



Очереди по приоритету и пирамидальная сортировка

В главе 3 мы рассмотрели несколько очень простых структур данных. Одной из них была очередь. В эту структуру можно было добавлять элементы, а затем извлекать их в порядке поступления. При этом сохранение даты и времени создания записи позволяло не обращать внимания на реальную длину элемента в очереди. Вместо этого мы просто организовали элементы по порядку их поступления в связный список или массив, а затем удаляли их в порядке поступления. При этом использовались две базовые операции: "добавление элемента в очередь" (называемая еще постановкой в очередь) и "удаление самого старого элемента очереди" (или вывод из очереди).

Все это замечательно, и очередь сама по себе является важной структурой данных. Однако ей присуще ограничение, заключающееся в том, что элементы обрабатываются в порядке их поступления. Предположим, что элементы нужно обрабатывать в совершенно ином порядке. Иначе говоря, требуется очередь, для которой по-прежнему определена операция "добавления элемента", но второй операцией является не "удаление самого старого элемента", а "удаление самого большого элемента" или "удаление самого малого элемента". В этом случае упрощенный критерий упорядочения "по возрасту" желательно заменить каким-то совершенно иным критерием. Например, предположим, что элементами очереди являются задачи, которые необходимо выполнить, и требуется извлечь задачу, обладающую наивысшим приоритетом.

Очередь по приоритету

Фактически, упомянутый пример обуславливает название новой структуры данных, называемой очередью по приоритету. Для *очереди по приоритету* (*priority queue*) определены две базовых операции: добавление элемента (как и ранее) и извлечение элемента с наивысшим приоритетом. (Естественно, при этом предполагается, что с каждым элементом связано значение приоритета, которое легко проверить.) У читателей может возникнуть вопрос, что в данном контексте понимается под "приоритетом"? Что ж, приоритетом может быть все что угодно. В классическом понимании это численное значение, указывающее на приоритет элемента в каком-либо процессе. Примерами могут служить очереди на печать в операционных системах, очереди заданий или потоки в многопоточной среде. Если принять в качестве примера очередь печати, каждому заданию печати присваивается приоритет – значение, указывающее важность данного задания печати. Задания на печать с высоким приоритетом должны обрабатываться раньше заданий с низким приоритетом. В этом случае операционная система должна была бы завершить выполнение конкретного задания печати, обратиться к очереди печати и извлечь задание печати с наивысшим приоритетом. По мере выполнения работы в операционной системе, другие задания печати будут добавляться в очередь печати с различными приоритетами, а очередь печати обеспечит такую их организацию, чтобы при необходимости можно было определить печатное задание с наивысшим приоритетом.

Однако следует отметить, что используемое в качестве "приоритета" значение не обязательно должно быть классическим номером приоритета. Оно может иметь любой тип или значение – главное, чтобы значения были связаны отношением упорядочения, и чтобы очередь могла определить элемент с наибольшим значением. (*Отношение упорядочения* набора объектов – это правило, которое позволяет упорядочить объекты так, чтобы объект x был "меньше" объекта y . Если x меньше y , то y не может быть меньше x . Кроме того, если x меньше y , а y меньше z , то x меньше z . Обычное упорядочение целых чисел, когда 2 меньше 3 и т.д., представляет собой пример отношения упорядочения.)

Например, значением приоритета могло бы быть имя (иначе говоря, строка), а упорядочением мог бы быть стандартный алфавитный порядок. Таким образом, вместо операции извлечения элемента с наибольшим приоритетом можно было бы использовать извлечение элемента, располагающегося раньше в алфавитном порядке (т.е. все А располагаются перед всеми В и т.д.).

Итак, очередь по приоритету должна обеспечивать (1) хранение произвольного количества элементов, (2) добавление в очередь элемента с определенным приоритетом и (3) выявление и удаление элемента с наивысшим приоритетом.

Первая простая реализация

При разработке очереди по приоритету первый атрибут (возможность хранения произвольного количества элементов) наталкивает на мысль об использовании ка-

кой-либо расширяемой структуры данных типа связного списка или расширяемого массива, такого как **TList**. Мы будем использовать (по крайней мере, пока) **TList**.

Следующий атрибут (возможность добавления элемента в очередь) легко реализовать в случае применения **TList**: достаточно вызвать метод **Add** структуры **TList**. Мы будем исходить из предположения, что добавляемыми в очередь по приоритету элементами будут каким-то образом описанные объекты, свойством которых является их приоритет. В результате мы получаем достаточно простой элемент, не отвлекающий наше внимание от функциональных возможностей очереди по приоритету.

Реализация третьего атрибута (возможности отыскания наивысшего приоритета и возвращения связанного с ним объекта с удалением его из обрабатываемой очереди по приоритету) несколько сложнее, но все же сравнительно проста. По существу мы выполняем итерационный просмотр элементов структуры **TList**, сравнивая приоритет каждого элемента с наибольшим обнаруженным приоритетом. Если приоритет данного элемента больше наибольшего обнаруженного до этого момента приоритета, мы помечаем индекс этого элемента как нового элемента с наибольшим приоритетом и переходим к следующему элементу. Этот поиск является простым последовательным поиском. После проверки всех элементов в структуре **TList** мы знаем, какой из них является наибольшим (его индекс был запомнен), и просто удаляем его из **TList**.

Пример кода простой очереди по приоритету приведен в листинге 9.1. В нем используется функция сравнения, которая передается очереди по приоритету при ее создании, и которая сравнивает приоритеты элементов. Таким образом, самой очереди по приоритету не нужно уметь сравнивать приоритеты (и, следовательно, знать, являются ли они числами, строками или чем-либо еще): очередь просто вызывает функцию сравнения, передавая ей два элемента, приоритеты которых требуется сравнить. Обратите также внимание, что очереди не нужно знать, что представляют собой элементы. Она просто хранит их. Поэтому можно просто объявить использование указателей в очереди и при необходимости выполнять приведение типов.

Листинг 9.1. Простая очередь по приоритету, построенная на основе структуры **TList**

```
type
TtdSimplePriQueue1 = class
    private
        FCompare : TtdCompareFunc;
        FList : TList;
    protected
        function pqGetCount : integer;
    public
        constructor Create(aCompare : TtdCompareFunc);
        destructor Destroy; override;
```

```

    function Dequeue : pointer;
    procedure Enqueue(aItem : pointer);
    property Count : integer read pqGetCount;
end;

constructor TtdSimplePriQueue1.Create(aCompare : TdCompareFunc);
begin
    inherited Create;
    FCompare := aCompare;
    FList := TList.Create;
end;

destructor TtdSimplePriQueue1.Destroy;
begin
    FList.Free;
    inherited Destroy;
end;

function TtdSimplePriQueue1.Dequeue : pointer;
var
    Inx : integer;
    PQCount : integer;
    MaxInx : integer;
    MaxItem : pointer;
begin
    PQCount := Count;
    if (PQCount = 0) then
        Result := nil
    else if (PQCount = 1) then begin
        Result := FList.List^[0];
        FList.Clear;
    end
    else begin
        MaxItem := FList.List^[0];
        MaxInx := 0;
        for Inx := 1 to pred(PQCount) do
            if (FCompare(FList.List^[Inx], MaxItem) > 0) then begin
                MaxItem := FList.List^[Inx];
                MaxInx := Inx;
            end;
        end;
        Result := MaxItem;
        FList.List^[MaxInx] := FList.Last;
        FList.Count := FList.Count - 1;
    end;
end;

procedure TtdSimplePriQueue1.Enqueue(aItem : pointer);
begin
    FList.Add(aItem);
end;

```

```
function TtdSimplePriQueue1.pqGetCount : integer;  
begin  
    Result := FList.Count;  
end;
```

Из листинга 9.1 видно, что в действительности этот класс является достаточно простым, и даже добавление в него отсутствовавшей ранее проверки на наличие ошибок не делает его громоздким. Единственный фрагмент кода, который представляет интерес – код удаления элемента: мы не вызываем метод **Delete** структуры данных **TList** (операция типа $O(n)$), а просто заменяем элемент, который нужно удалить, последним элементом и уменьшаем на единицу значение счетчика элементов (операция типа $O(1)$).

Исходный код класса **TtdSimplePriQueue1** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDPriQue.pas**.

После того, как мы убедились в простоте разработки создания этой очереди по приоритету, рассмотрим ее эффективность. Во-первых, добавление элемента в очередь по приоритету будет требовать постоянного времени. Иначе говоря, эта операция является операцией типа $O(1)$. Независимо от того, содержит ли очередь ноль или тысячи элементов, добавление нового элемента будет занимать приблизительно одно и то же время: мы всего лишь дописываем его в конец списка.

Теперь рассмотрим противоположную операцию: удаление элемента. В этом случае для отыскания элемента с наивысшим приоритетом потребуется выполнить считывание всех элементов в структуре **TList**. Этот поиск является последовательным и, как было показано в главе 4, эта операция является операцией типа $O(n)$. Требуемое для этого время пропорционально количеству элементов в очереди.

Таким образом, мы разработали и создали структуру данных, реализующую очередь по приоритету, в которой добавление элемента является операцией типа $O(1)$, а удаление – операцией типа $O(n)$. При наличии небольшого количества элементов эта структура оказывается вполне приемлемой и достаточно эффективной.

Вторая простая реализация

Однако при наличии большого количества элементов или при добавлении и удалении из очереди большого количества элементов она оказывается не столь эффективной, как хотелось бы. Уверен, что читатели сразу подумали об одном возможном способе повышения эффективности: поддержании структуры **TList** в порядке приоритетов. Иначе говоря, о поддержании ее в отсортированном виде в ходе всех добавлений. По существу, это усовершенствование означает перенос реальной задачи поддержания очереди из операции удаления элемента в операцию вставки элемента. При добавлении элемента необходимо найти для него правильную позицию внутри структуры **TList** после всех элементов с более низким приоритетом и перед всеми элементами с более высоким приоритетом. В случае вы-

полнения этой дополнительной задачи на этапе добавления все элементы структуры **TList** будут размещены в порядке своих приоритетов и, следовательно, при удалении элемента потребуется всего лишь удалить последний элемент структуры. Фактически, при этом удаление превращается в операцию типа $O(1)$ (нам точно известно, где расположен элемент с наивысшим приоритетом – он находится в конце очереди, поэтому удаление не зависит от количества элементов).

Вычисление времени, которое требуется для вставки в этот отсортированный список **TList**, несколько сложнее. Этот процесс проще всего представить сортировкой простыми вставками (которая была описана в главе 5). Мы увеличиваем размер **TList** на один элемент, а затем, подобно четкам, по одному перемещаем элементы на свободное место, начиная с конца структуры **TList**. Процесс прекращается по достижении элемента, приоритет которого ниже приоритета элемента, который мы пытаемся вставить. В результате в структуре **TList** образуется "пробел", в который можно поместить новый элемент. В структуре **TList**, содержащей n элементов, в среднем придется переместить $n/2$ элементов. Следовательно, вставка является операцией типа $O(n)$ (т.е. требуемое для ее выполнения время снова пропорционально количеству элементов в очереди), хотя это усовершенствование позволяет несколько уменьшить время выполнения операции по сравнению с предыдущей реализацией. Пример кода выполнения этих двух операций в описанной структуре данных приведен в листинге 9.2.

Листинг 9.2. Очередь по приоритету, в которой используется отсортированная структура данных **TList**

```

type
  TtdSimplePriQueue2 = class
    private
      FCompare : TtdCompareFunc;
      FList : TList;
    protected
      function pqGetCount : integer;
    public
      constructor Create(aCompare : TtdCompareFunc);
      destructor Destroy; override;
      function Dequeue : pointer;
      procedure Enqueue(aItem : pointer);
      property Count : integer read pqGetCount;
  end;

constructor TtdSimplePriQueue2.Create(aCompare : TtdCompareFunc);
begin
  inherited Create;
  FCompare := aCompare;
  FList := TList.Create;
end;
```

```
destructor TtdSimplePriQueue2.Destroy;  
begin  
  FList.Free;  
  inherited Destroy;  
end;  
  
function TtdSimplePriQueue2.Dequeue : pointer;  
begin  
  Result := FList.Last;  
  FList.Count := FList.Count - 1;  
end;  
  
procedure TtdSimplePriQueue2.Enqueue(aItem : pointer);  
var  
  Inx : integer;  
begin  
  {увеличить количество элементов в списке}  
  FList.Count := FList.Count + 1;  
  {определить место помещения нового элемента}  
  Inx := FList.Count - 2;  
  while (Inx >= 0) and (FCompare(FList.List^[Inx], aItem) > 0) do begin  
    FList.List^[Inx+1] := FList.List^[Inx];  
    dec(Inx);  
  end;  
  {поместить элемент в эту позицию}  
  FList.List^[Inx+1] := aItem;  
end;  
  
function TtdSimplePriQueue2.pqGetCount : integer;  
begin  
  Result := FList.Count;  
end;
```

Исходный код класса **TtdSimplePriQueue2** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDPriQue.pas**.

В ходе разработки и создания этой усовершенствованной очереди по приоритету мы перешли от быстрой вставки/медленного удаления к медленной вставке/быстрому удалению. Нельзя ли воспользоваться более эффективным алгоритмом?

Еще одна возможность предполагает полный отказ от использования структуры **TList** и переход к другой структуре данных: дереву двоичного поиска, описанному в главе 8, или списку с пропусками, описанному в главе 6. При использовании обеих этих структур данных и вставка и удаление являются операциями типа $O(\log(n))$. Иначе говоря, время, требуемое как для вставки, так и для удаления элемента, пропорционально логарифму числа элементов в структуре. Однако применение обеих этих структур данных сопряжено с некоторыми сложностями. В отношении списка с пропусками это связано с его вероятностной структурой, а в отношении дерева двоичного поиска – потому, что в ходе вставки и удаления

необходимо заботиться о балансировке результирующего дерева. Существует ли какая-то более простая структура данных?

Сортирующее дерево

Классическая структура данных, используемая для создания очереди по приоритету, известна под названием сортирующего дерева (или "кучи"). *Сортирующее дерево (heap)*, на которое еще ссылаются как на частично упорядоченное полное двоичное дерево, — это двоичное дерево с определенными специальными свойствами и несколькими специальными операциями. (Не путайте эту "кучу" с "кучей", используемой в среде Delphi, — областью памяти, в которой выполняется все распределение памяти.)

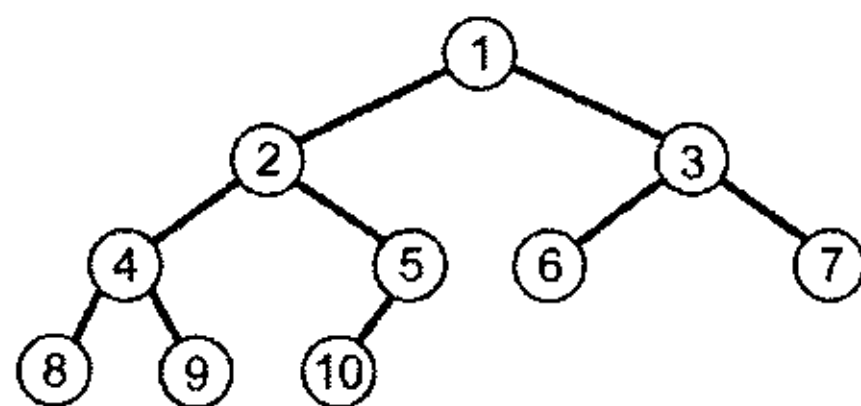


Рисунок 9.1. Сортирующее дерево

В дереве двоичного поиска узлы организованы так, что каждый узел больше своего левого дочернего узла и меньше своего правого дочернего узла. Такое упорядочение называется строгим. В сортирующем дереве используется менее строгое упорядочение, называемое пирамидальным свойством. *Пирамидальное свойство* означает всего лишь, что любой узел в дереве должен быть больше обоих его дочерних узлов. Обратите внимание, что пирамидальное свойство ничего не говорит о порядке дочерних узлов данного узла. Например, оно не утверждает, что левый дочерний узел должен быть меньше правого дочернего узла.

Сортирующее дерево обладает еще одним атрибутом: двоичное дерево должно быть *полным*. Двоичное дерево называется полным, когда все его уровни, за исключением, быть может, последнего, заполнены. В последнем уровне все узлы размещаются максимально сдвинутыми влево. Полное дерево является максимально сбалансированным. Полное двоичное дерево показано на рис. 9.1.

Так как же эта структура может помочь в наших поисках идеальной структуры очереди по приоритету? Что ж, операции вставки и удаления при использовании сортирующего дерева являются операциями типа $O(\log(n))$, но они выполняются значительно быстрее, чем эти же операции в дереве двоичного поиска, независимо от того, является ли оно сбалансированным. Это тот случай, когда O -нотация оказывается неприемлемой — она не позволяет количественно определить, какая из двух операций с одним и тем же значением O большого действительно выполняется *быстрее*.

Вставка в сортирующее дерево

Рассмотрим алгоритмы вставки и удаления. Вначале ознакомимся со вставкой. Чтобы вставить элемент в сортирующее дерево, мы добавляем его в конец этого дерева, в единственную позицию, которая соответствует требованию полноты (на рис. 5 этой позицией была бы позиция правого дочернего узла пятого узла).

Этот атрибут сортирующего дерева сохраняется. При этом может быть нарушен второй атрибут – пирамидальность. Новый узел может быть большего своего родительского узла, поэтому потребуется исправить дерево и восстановить свойство пирамидальности.

Если этот новый дочерний узел больше своего родительского узла, мы меняем его местами с родительским узлом. В своей новой позиции новый узел может быть все же больше своего нового родительского узла, и поэтому их нужно снова поменять местами. Мы продолжаем такое перемещение по сортирующему дереву до тех пор, пока не будет достигнута точка, в которой новый узел не больше родительского узла или пока не достигнем корневого узла дерева. Выполнение упомянутого алгоритма обеспечивает, чтобы все узлы были больше обоих своих дочерних узлов, и, таким образом, свойство пирамидальности восстанавливается. Этот алгоритм называется алгоритмом *пузырькового подъема* (*bubble up*), поскольку новый узел подобно пузырьку воздуха "всплывает" вверх, пока не попадает в требуемую позицию (либо в позиции корневого узла, либо под узлом, который больше него).

По существу, свойство пирамидальности гарантирует размещение наибольшего элемента в позиции корневого узла. Это достаточно легко доказать: если бы наибольший элемент размещался не в позиции корневого узла, он имел бы родительский узел. Поскольку он является наибольшим элементом, мы были бы вынуждены заключить, что он больше своего родительского узла, – а это является нарушением свойства пирамидальности. Следовательно, первоначальное предположение, что наибольший узел размещается не в позиции корневого узла, неверно.

Удаление из сортирующего дерева

Теперь, поскольку мы только что показали, что требуемый элемент расположен в позиции корневого узла, можно приступить к удалению наибольшего узла. Удаление корневого узла и передача этого элемента вызывающей процедуре – не самая лучшая идея. В результате мы получили бы два отдельных дочерних дерева – что было бы полным нарушением атрибута полноты сортирующего дерева. Вместо этого мы заменяем корневой узел последним узлом сортирующего дерева и уменьшаем его размер, тем самым обеспечивая сохранение полноты. Но при этом снова возможно нарушение свойства пирамидальности. Весьма вероятно, что новый корневой узел будет меньше одного или обоих своих дочерних узлов. Поэтому нужно снова исправить сортирующее дерево, чтобы восстановить его свойство пирамидальности. Для этого мы находим больший из двух дочерних узлов и меняем его местами с данным узлом. Как и ранее, эта позиция может нарушать свойство пирамидальности, поэтому мы проверяем, является ли данный узел меньше одного (или обоих) дочерних узлов и повторяем процесс. Со временем выяснится, что узел погрузился (или "просочился") на уровень, где он больше обоих своих дочерних уз-

лов или является листом, не имеющим дочерних узлов. В любом случае свойство пирамидальности восстанавливается. Этот алгоритм называется алгоритмом *просачивания вниз* (*trickle down*).

Если реализовать кучу, используя реальное двоичное дерево, подобное описанному в главе 8, выяснится, что при этом расходуется довольно большой объем памяти. Для каждого узла необходимо поддерживать по три указателя: по одному для каждого дочернего узла, чтобы можно было реализовать алгоритм просачивания в нижние уровни дерева, и один для родительского узла, чтобы можно было реализовать алгоритм пузырькового подъема. При каждом обмене узлов местами придется обновлять бесчисленное количество указателей для множества узлов. Обычно в этом случае применяют прием, когда узлы остаются на своих местах, а вместо этого меняют местами элементы внутри узлов.

Однако существует более простой способ. Полное двоичное дерево легко представить массивом. Снова взгляните на рис. 9.1. Выполните просмотр дерева, используя обход по уровням. Обратите внимание, что в полном дереве обход по уровням не затрагивает никаких пробелов, в которых имеется позиция для узла, но какой-либо узел отсутствует (естественно, до тех пор, пока не будут посещены все узлы и не будет достигнут конец дерева). Узлы легко отобразить элементами массива, чтобы последовательное посещение элементов массива было эквивалентно посещению узлов посредством обхода по уровням. При этом элемент 1 массива был бы корневым узлом сортирующего дерева, элемент 2 – левым дочерним узлом корневого узла, элемент 3 – правым дочерним узлом корневого узла и т.д. Фактически, именно так пронумерованы узлы на рис. 9.1.

Теперь обратите внимание на нумерацию дочерних узлов каждого узла. Дочерними узлами корневого узла 1 являются, соответственно, узлы 2 и 3. Дочерними узлами узла 4 являются узлы 8 и 9, а узла 6 – узлы 12 и 13. Заметили ли вы какую-нибудь закономерность? Дочерними узлами узла n являются узлы $2n$ и $2n + 1$, а родительским узлом узла n является узел $n/2$. Теперь уже не обязательно, чтобы узел содержал указатели на родительский и дочерние узлы. Вместо этого можно воспользоваться простым арифметическим отношением. Таким образом, мы изобрели метод реализации сортирующего дерева при помощи массива, и решив более простую задачу, можно было бы снова отдать предпочтение структуре **TList**.

Проблема заключается в следующем: рассмотренная нами реализация сортирующего дерева в виде массива требует, чтобы отсчет элементов массива начинался единицы, а не с нуля, как имеет место в структуре **TList**. Этого достаточно легко добиться. Достаточно изменить арифметическую формулу вычисления индекса родительского и дочерних узлов. Дочерние узлы узла n должны располагаться в позициях $2n + 1$ и $2n + 2$, а родительский узел этого узла – в позиции $(n - 1)/2$.

Реализация очереди по приоритету при помощи сортирующего дерева

Код интерфейса результирующей очереди по приоритету, в которой используется сортирующее дерево и которая реализована при помощи структуры **TList**, приведен в листинге 9.3.

Листинг 9.3. Интерфейс класса **TtdPriorityQueue**

```

type
  TtdPriorityQueue = class
    private
      FCompare : TtdCompareFunc;
      FDispose : TtdDisposeProc;
      FList : TList;
      FName : TtdNameString;
    protected
      function pqGetCount : integer;
      procedure pqError(aErrorCode : integer;
                       const aMethodName : TtdNameString);
      procedure pqBubbleUp(aFromInx : integer);
      procedure pqTrickleDown;
      procedure pqTrickleDownStd;
    public
      constructor Create(aCompare : TtdCompareFunc;
                       aDispose : TtdDisposeProc );
      destructor Destroy; override;
      procedure Clear;
      function Dequeue : pointer;
      procedure Enqueue(aItem : pointer);
      function Examine : pointer;
      function IsEmpty : boolean;
      property Count : integer read pqGetCount;
      property Name : TtdNameString read FName write FName;
  end;
```

Реализация и конструктора **Create**, и деструктора **Destroy** достаточно проста: первый должен создавать экземпляр **TList**, а второй должен всего лишь освободить внутренний объект **TList**. Подобно стандартной очереди, конструктор **Create** нуждается в процедуре удаления элемента, позволяющей при необходимости освободить элементы. Но, в отличие от стандартной очереди, теперь нам требуется процедура сравнения, позволяющая определить больший из двух элементов.

Листинг 9.4. Конструктор и деструктор очереди по приоритету

```

constructor TtdPriorityQueue.Create(aCompare : TtdCompareFunc;
                                   aDispose : TtdDisposeProc);

begin
  inherited Create;
  if not Assigned(aCompare) then
    pqError(tdePriQueueNoCompare, 'Create');
```

```

FCompare := aCompare;
FDispose := aDispose;
FList := TList.Create;
end;

destructor TtdPriorityQueue.Destroy;
begin
  Clear;
  FList.Free;
  inherited Destroy;
end;

```

Код реализации алгоритма вставки и процедуры, выполняющей реальную операцию пузырькового подъема, показан в листинге 9.5. Операция вставки реализована так, чтобы гарантировать размещение наибольшего элемента в корневом узле. Этот тип очереди по приоритету обычно называют *пирамидальной сортировкой выбором максимального элемента (max-heap)*. Если изменить процедуру сравнения так, чтобы она возвращала отрицательное число, если первый элемент больше второго, в корневом узле очереди по приоритету будет располагаться наименьший элемент. Такая сортировка называется *пирамидальной сортировкой выбором наименьшего элемента (min-heap)*.

Листинг 9.5. Вставка в TtdPriorityQueue: постановка в очередь

```

procedure TtdPriorityQueue.pqBubbleUp(aFromInx : integer);
var
  ParentInx : integer;
  Item : pointer;
begin
  Item := FList.List^[aFromInx];
  {если анализируемый элемент больше своего родительского элемента, необходимо поменять его местами с родительским элементом и продолжить процесс из новой позиции элемента}
  {Примечание: родительский узел узла, имеющего индекс n, располагается в позиции (n-1)/2}
  ParentInx := (aFromInx - 1) div 2;
  {если данный элемент имеет родительский узел и больше родительского элемента...}
  while (aFromInx > 0) and (FCompare(Item, FList.List^[ParentInx]) > 0) do
    begin
      {необходимо переместить родительский элемент вниз по дереву}
      FList.List^[aFromInx] := FList.List^[ParentInx];
      aFromInx := ParentInx;
      ParentInx := (aFromInx - 1) div 2;
    end;
    {сохранить элемент в правильной позиции}
    FList.List^[aFromInx] := Item;
  end;
end;

```

```

procedure TtdPriorityQueue.Enqueue(aItem : pointer);
begin
  {добавить элемент в конец списка и выполнить его пузырьковый подъем на
   максимумально возможный уровень}
  FList.Add(aItem);
  pqBubbleUp(pred(FList.Count));
end;

```

В листинге 9.6 приведен фрагмент кода, реализующий последнюю часть очереди по приоритету: алгоритм удаления и процедуру, которая выполняет операцию просачивания вниз.

Листинг 9.6. Удаление из TtdPriorityQueue: исключение из очереди

```

procedure TtdPriorityQueue.pqTrickleDownStd;
var
  FromInx : integer;
  ChildInx : integer;
  MaxInx : integer;
  Item : pointer;
begin
  FromInx := 0;
  Item := FList.List^[0];
  MaxInx := FList.Count - 1;
  {если анализируемый элемент меньше одного из его дочерних элементов, нужно
   поменять его местами с большим дочерним элементом и продолжить процесс из
   новой позиции}
  {Примечание: дочерние узлы родительского узла n располагаются в позициях
   2n+1 и 2n+2}
  ChildInx := (FromInx * 2) + 1;
  {если существует по меньшей мере левый дочерний узел...}
  while (ChildInx <= MaxInx) do begin
    {если существует также и правый дочерний узел, необходимо вычислить индекс
     большего дочернего узла}
    if (succ(ChildInx) <= MaxInx) and
      (FCompare(FList.List^[ChildInx],
        FList.List^[succ(ChildInx)]) < 0) then
      inc(ChildInx);
    {если данный элемент больше или равен большему дочернему элементу, задача
     выполнена}
    if (FCompare(Item, FList.List^[ChildInx]) >= 0) then
      Break;
    {в противном случае больший дочерний элемент нужно переместить вверх по
     дереву, а сам элемент - вниз по дереву, а затем повторить процесс}
    FList.List^[FromInx] := FList.List^[ChildInx];
    FromInx := ChildInx;
    ChildInx := (FromInx * 2) + 1;
  end;
  {сохранить элемент в правильной позиции}
  FList.List^[FromInx] := Item;
end;

```

```

function TtdPriorityQueue.Dequeue : pointer;
begin
  {проверить наличие элемента для его исключения из очереди}
  if (FList.Count = 0) then
    pqError(tdeQueueIsEmpty, 'Dequeue');
  {вернуть элемент, расположенный в корневом узле}
  Result := FList.List^[0];
  {если очередь содержала только один элемент, теперь она пуста}
  if (FList.Count = 1) then
    FList.Count := 0
  {если очередь содержала два элемента, достаточно заменить корневой узел
    единственным оставшимся дочерним узлом; очевидно, что при этом свойство
    пирамидальности сохраняется}
  else if (FList.Count = 2) then begin
    FList.List^[0] := FList.List^[1];
    FList.Count := 1;
  end
  {в противном случае больший дочерний элемент нужно переместить верх по
    дереву, а сам элемент - вниз по дереву, а затем повторить процесс}
  else begin
    {заменить корневой узел дочерним узлом, расположенным в нижней правой
    позиции, уменьшить размер списка, и, наконец, выполнить просачивание
    корневого элемента вниз на максимальную глубину}
    FList.List^[0] := FList.Last;
    FList.Count := FList.Count - 1;
    pqTrickleDownStd;
  end;
end;

```

Обратите внимание, что на каждом этапе выполнения алгоритма просачивания в процессе перемещения элементов вниз по куче выполняется не более двух сравнений: сравнение двух дочерних элементов с целью определения большего из них и сравнение большего дочернего элемента с родительским элементом для выяснения того, нужно ли их менять местами. По сравнению с операцией пузырькового подъема, когда при подъеме в рамках сортирующего дерева на каждом уровне выполняется только одно сравнение, этот алгоритм выглядит несколько излишне трудоемким. Нельзя ли каким-то образом улучшить ситуацию?

Роберт Флойд (Robert Floyd) обратил внимание, что первый шаг операции исключения из очереди требует удаления элемента с наивысшим приоритетом и замены его одним из наименьших элементов сортирующего дерева. Этот элемент не обязательно должен быть *наименьшим*, но в процессе применения алгоритма просачивания он наверняка будет перемещен на один из нижних уровней дерева. Иначе говоря, большинство операций сравнения родительского элемента с его большим дочерним элементом, выполняемое в ходе процесса просачивания, вероятно, лишено особого смысла, поскольку результат сравнения заведомо известен: родительский элемент будет меньше своего большего дочернего элемента. Поэтому

Флойд предложил следующее: при выполнении процесса просачивания полностью отказаться от сравнений родительского элемента с большими дочерними элементами и всегда менять местами родительский элемент и его больший дочерний элемент. Конечно, со временем мы достигнем нижнего уровня сортирующего дерева, и элемент может оказаться в неправильной позиции (другими словами, он может оказаться больше своего родительского элемента). Это не имеет значения, поскольку в этом случае мы просто воспользуемся операцией пузырькового подъема. Поскольку элемент, к которому было применено просачивание, был одним из наименьших в сортирующем дереве, весьма вероятно, что его придется поднимать не слишком высоко, если вообще придется.

Описанная оптимизация приводит к уменьшению количества сравнений, выполняемых во время операции исключения из очереди, примерно в два раза. Если сравнения требуют значительных затрат времени (например, при сравнении строк), эта оптимизация себя оправдывает. Ее применение оправдано также и в нашей реализации очереди по приоритету, в которой мы используем функцию сравнения, а не простое сравнение целых чисел.

Листинг 9.7: Оптимизированная операция просачивания

```
procedure TtdPriorityQueue.pqTrickleDown;  
var  
    FromInx : integer;  
    ChildInx : integer;  
    MaxInx : integer;  
    Item : pointer;  
begin  
    FromInx := 0;  
    Item := FList.List^[0];  
    MaxInx := pred(FList.Count);  
    {выполнять обмен местами анализируемого элемента и его большего дочернего  
    элемента до тех пор, пока у него не окажется ни одного дочернего элемента}  
    {Примечание: дочерние элементы родительского узла n располагаются в позициях  
    2n+1 и 2n+2}  
    ChildInx := (FromInx * 2) + 1;  
    {до тех пор, пока существует по меньшей мере левый дочерний элемент...}  
    while (ChildInx <= MaxInx) do begin  
        {если при этом существует также правый дочерний элемент, необходимо  
        вычислять индекс большего дочернего элемента}  
        if (succ(ChildInx) <= MaxInx) and  
            (FCompare(FList.List^[ChildInx],  
                FList.List^[succ(ChildInx)]) < 0) then  
            inc(ChildInx);  
        {переместить больший дочерний элемент вверх, а данный элемент вниз по  
        дереву и повторить процесс}  
        FList.List^[FromInx] := FList.List^[ChildInx];  
        FromInx := ChildInx;  
        ChildInx := (FromInx * 2) + 1;  
    end;
```

```

{сохранить элемент в той позиции, в которой процесс был прерван}
FList.List^[FromInx] := Item;
{теперь необходимо выполнить пузырьковый подъем этого элемента
 вверх по дереву}
pqBubbleUp(FromInx);
end;

```

Исходный код класса **TtdPriorityQueue** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDPriQue.pas**.

Пирамидальная сортировка

После того, как мы реализовали очередь по приоритету в виде сортирующего дерева, можно утверждать, что такое дерево можно использовать как алгоритм сортировки: одновременно добавлять в сортирующее дерево ряд элементов, а затем выбирать их по одному в требуемом порядке. (Обратите внимание, что в случае применения описанного метода элементы выбираются в обратном порядке. Т.е. вначале выбирается наибольший элемент. Однако если использовать обратный метод сравнения, элементы можно извлекать в порядке их возрастания.)

Не удивительно, что алгоритм сортировки с помощью сортирующего дерева называется *пирамидальной сортировкой (heapsort)*. Если припоминаете, в главе 5 рассмотрение этого метода сортировки было отложено до приобретения необходимых теоретических сведений.

Только что названный алгоритм состоит в следующем: предположим, что у нас имеется очередь по приоритету, реализованная в виде сортирующего дерева с выбором минимального элемента. Мы добавляем в него все элементы, а затем удаляем их по одному. Если бы вначале в структуре **TList** хранились неотсортированные элементы, применение этого алгоритма означало бы, что все элементы копировались бы из одной структуры **TList** в другую, а затем обратно. Намного более предпочтительным было бы применение сортировки по месту, при которой не нужно было бы копировать элементы из одного массива в другой. Иначе говоря, нельзя ли преобразовать существующий массив в сортирующее дерево, применив к нему свойство пирамидальности?

Алгоритм Флойда

Роберт Флойд разработал такой достаточно интересный алгоритм, при котором время генерирования сортирующего дерева подчиняется отношению $O(n)$, что значительно эффективнее алгоритма типа $O(n \log(n))$ добавления элементов по одному в отдельное сортирующее дерево.

Алгоритм Флойда работает следующим образом. Процесс начинается с родительского узла самого правого дочернего узла (т.е. узла, расположенного в крайней правой позиции последнего уровня сортирующего дерева). Применим к этому

родительскому узлу алгоритм просачивания. Выберем узел, расположенный на этом же уровне слева от родительского узла (конечно, он тоже будет родительским). Снова применим алгоритм просачивания. Продолжим перемещение влево, применяя алгоритм просачивания, пока не останется узлов для обработки. Перейдем к крайнему справа узлу следующего уровня. Продолжим этот же процесс перемещения справа налево, переходя от уровня к уровню, пока не будет достигнут корневой узел. С этого момента массив упорядочен в виде сортирующего дерева.

Чтобы доказать справедливость отношения $O(n)$, предположим, что сортирующее дерево содержит 31 элемент (это сортирующее дерево будет иметь 5 заполненных уровней). На первом этапе нужно было бы выполнить обработку всех узлов четвертого уровня. Таких узлов восемь и для каждого из них потребовалось бы не более одной операции перемещения на более низкий уровень – всего таких операций требовалось бы не более восьми. На следующем этапе нужно было бы сформировать сортирующие мини-деревья на 3 уровне. Таких сортирующих деревьев четыре и для каждого требовалось бы не более двух операций понижения уровня (всего восемь). На следующем шаге потребовалось бы образовать сортирующие деревья на 2 уровне: существует три узла, которые могли бы требовать обработки, для каждого из которых может потребоваться не более трех операций перемещения на более низкий уровень. Таким образом, для узлов этого уровня может потребоваться выполнение не более шести операций. Для образования сортирующего дерева на последнем шаге требуется максимум четыре операции понижения уровня. Таким образом, всего для формирования сортирующего дерева требовалось бы выполнение не более 26 операций перемещения на более низкий уровень – меньше исходного количества узлов. Если применить эти же рассуждения к сортирующему дереву с $2^n - 1$ узлами, выяснится, что для создания сортирующего дерева требуется не более $2^n - n - 1$ операций перемещения на более низкий уровень. Отсюда следует вывод о справедливости первоначального утверждения, что алгоритм Флойда является операцией типа $O(n)$.

Завершение пирамидальной сортировки

Итак, массив упорядочен в виде сортирующего дерева. Что дальше? Удаление элементов по одному по-прежнему означает, что их нужно поместить куда-либо в отсортированном порядке, предположительно, в какой-нибудь вспомогательный массив. Так ли это? Немного подумаем. Если мы удаляем наибольший элемент, размер сортирующего дерева уменьшается на единицу, а в конце массива остается место для только что удаленного элемента. Фактически, алгоритм удаления элемента из сортирующего дерева требует, чтобы самый нижний, крайний справа узел копировался в позицию корневого узла, прежде чем к нему будет применена операция просачивания. Поэтому нужно всего лишь поменять местами корневой узел и самый нижний крайний справа узел, уменьшить значение счетчика количества элементов сортирующего дерева, а затем применить алгоритм просачивания.

Этот процесс нужно продолжать до тех пор, пока элементы в сортирующем дереве не иссякнут. В результате мы получаем элементы исходного массива, но теперь они оказываются отсортированными.

Полный код подпрограммы пирамидальной сортировки, реализованной так же, как были реализованы все процедуры сортировки в главе 5, приведен листинге 9.8.

Листинг 9.8. Алгоритм пирамидальной сортировки

```

procedure HSTrickleDown(  aList : PPointerList;
                           aFromInx : integer;
                           aCount : integer;
                           aCompare : TtdCompareFunc );

var
  Item : pointer;
  ChildInx : integer;
  ParentInx: integer;
begin
  {вначале необходимо выполнить простую операцию просачивания, постоянно
   заменяя родительский узел его большим дочерним элементом, пока не будет
   достигнут нужный уровень сортирующего дерева}
  Item := aList^[aFromInx];
  ChildInx := (aFromInx * 2) + 1;
  while (ChildInx < aCount) do begin
    if (succ(ChildInx) < aCount) and
      (aCompare(aList^[ChildInx], aList^[succ(ChildInx)]) < 0) then
      inc(ChildInx);
    aList^[aFromInx] := aList^[ChildInx];
    aFromInx := ChildInx;
    ChildInx := (aFromInx * 2) + 1;
  end;
  {теперь из позиции, в которой был прекращен предыдущий процесс, необходимо
   выполнить операцию пузырькового подъема}
  ParentInx := (aFromInx - 1) div 2;
  while (aFromInx > 0) and (aCompare(Item, aList^[ParentInx]) > 0) do begin
    aList^[aFromInx] := aList^[ParentInx];
    aFromInx := ParentInx;
    ParentInx := (aFromInx - 1) div 2;
  end;
  {сохранить элемент в той позиции, где был прекращен процесс пузырькового
   подъема}
  aList^[aFromInx] := Item;
end;

procedure HSTrickleDownStd( aList : PPointerList;
                            aFromInx : integer;
                            aCount : integer;
                            aCompare : TtdCompareFunc );

var
  Item : pointer;
  ChildInx : integer;

```

```

begin
  Item := aList^[aFromInx];
  ChildInx := (aFromInx * 2) + 1;
  while (ChildInx < aCount) do begin
    if (succ(ChildInx) < aCount) and
      (aCompare(aList^[ChildInx], aList^[succ(ChildInx)]) < 0) then
      inc(ChildInx);
    if aCompare(Item, aList^[ChildInx]) >= 0 then
      Break;
    aList^[aFromInx] := aList^[ChildInx];
    aFromInx := ChildInx;
    ChildInx := (aFromInx * 2) + 1;
  end;
  aList^[aFromInx] := Item;
end;

procedure TDHeapSort(  aList : TList;
                       aFirst : integer;
                       aLast : integer;
                       aCompare : TtdCompareFunc );

var
  ItemCount : integer;
  Inx : integer;
  Temp : pointer;
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDHeapSort');
  {преобразовать список за счет применения алгоритма пирамидальной сортировки
   Флойда}
  ItemCount := aLast - aFirst + 1;
  for Inx := pred(ItemCount div 2) downto 0 do
    HSTrickleDownStd(@aList.List^[aFirst], Inx, ItemCount, aCompare);
  {удаление элементов из сортирующего дерева по одному, с помещением их в конец
   массива}
  for Inx := pred(ItemCount) downto 0 do begin
    Temp := aList.List^[aFirst];
    aList.List^[aFirst] := aList.List^[aFirst+Inx];
    aList.List^[aFirst+Inx] := Temp;
    HSTrickleDown(@aList.List^[aFirst], 0, Inx, aCompare);
  end;
end;

```

Обратите внимание, что на первом этапе, при создании сортирующего дерева из массива, мы использовали стандартный алгоритм просачивания (алгоритм Флойда), но на втором этапе (при удалении наибольшего элемента из постоянно уменьшающегося сортирующего дерева) был применен оптимизированный алгоритм просачивания Флойда. На первом этапе мы ничего не знали о распределении элементов в массиве, поэтому имело смысл просто применить стандартный алго-

ритм просачивания – в конце концов, в целом алгоритм Флойда является операцией типа $O(n)$. Однако на втором этапе мы знаем, что меняем местами наибольший элемент и один из наименьших элементов. Поэтому в целесообразно осуществить оптимизацию.

До сих пор не был пояснен один момент. Если в качестве очереди по приоритету используется сортирующее дерево, отсортированное выбором максимального элемента, извлечение элементов будет выполняться в обратном порядке – начиная с наибольшего и заканчивая наименьшим. Однако если сортирующее дерево, отсортированное выбором максимального элемента, используется для пирамидальной сортировки, элементы будут отсортированы в порядке возрастания, а не в обратном порядке. При использовании кучи, отсортированной выбором минимального элемента, элементы будут удаляться в порядке возрастания, но пирамидальная сортировка будет выполняться в порядке убывания.

Важность алгоритма пирамидальной сортировки обусловлена целым рядом причин. Во-первых, время его выполнения определяется отношением $O(n \log(n))$, следовательно, он работает достаточно быстро. Во-вторых, пирамидальная сортировка не имеет худшего случая. Сравним ее с быстрой сортировкой. В общем случае, как правило, быстрая сортировка выполняется быстрее пирамидальной (для выполнения пирамидальной сортировки потребуется выполнение большего количества операций сравнения, чем для быстрой сортировки, а внутренний цикл пирамидальной сортировки длится дольше, чем цикл быстрой сортировки). Но при выполнении быстрой сортировки возможны случаи, когда все ее преимущества сводятся буквально на нет, делая ее чрезвычайно медленной. (В худшем случае время выполнения этого алгоритма может определяться отношением $O(n^2)$, если только не будут приняты определенные меры по оптимизации алгоритма.) Если же сравнить пирамидальную сортировку с сортировкой слиянием, то мы видим, что эта сортировка выполняется на месте и не требует большого дополнительного объема памяти, как имеет место при выполнении сортировки слиянием. В заключение приходится признать, что алгоритм пирамидальной сортировки не очень устойчив.

Исходный код процедуры **TDHeapSort** и вспомогательных процедур можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDSorts.pas**.

Расширение очереди по приоритету

Сделав небольшое отступление для ознакомления с пирамидальной сортировкой, пора вернуться к очередям по приоритету и рассмотреть задачу расширения реализованной нами структуры данных.

Мы разработали структуру данных, позволяющую выполнять две основные операции: постановку в очередь, обеспечивающую добавление элемента в структуру, и исключение из очереди, которая возвращает элемент структуры с наивыс-

шим приоритетом (попутно мы рассмотрели определение приоритета за счет использования внешней функции сравнения). Полученную структуру мы назвали очередью по приоритету.

Однако структуры операционных систем, такие как очереди по приоритету потоков или очереди на печать, позволяют выполнять еще две операции: удалять элемент из очереди и возвращать его, независимо от позиции в очереди (элемент не обязательно должен быть наибольшим), а также изменять приоритет любого элемента в очереди.

При работе с очередью на печать операция удаления позволяет отменять задание на печать документа, печать которого больше не требуется, или удалять печатное задание из одной очереди и включать его в другую (например, если принтер, связанный с первой очередью, занят печатью крупного отчета). При работе с очередью по приоритету потоков можно временно повысить приоритет потока для повышения вероятности возобновления выполнения, когда операционная система в следующий раз решит изменить очередность обработки потоков.

На первый взгляд, реализация этих операций за счет использования сортирующего дерева может показаться затруднительной. Однако рассмотрим проблему подробнее. Классу очереди по приоритету нужно было бы передать ссылку на элемент, расположенный где-то в очереди, чтобы его можно было удалить или изменить его приоритет. Как найти элемент в очереди? Это один из тех случаев, когда "свободная" сортировка сортирующего дерева работает против нас. Единственным возможным методом поиска на этом этапе кажется последовательный поиск, но он выполняется достаточно медленно. После того как элемент найден, мы должны либо удалить его, либо изменить его приоритет, а затем восстановить полноту или пирамидальность сортирующего дерева, либо же оба свойства.

Восстановление свойства пирамидальности

Вторую проблему (восстановление свойства пирамидальности) проще решить, чем первую (отыскание элемента, который нужно удалить или изменить его приоритет). Поэтому вначале рассмотрим именно ее.

Чтобы удалить произвольный элемент из сортирующего дерева, его нужно было бы поменять местами с последним элементом и уменьшить размер сортирующего дерева. На этом этапе появляется элемент, который может нарушить свойство пирамидальности.

Для изменения приоритета произвольного элемента следует просто внести изменение, в результате чего элемент может также нарушить свойство пирамидальности.

В обоих случаях мы получаем элемент, который может находиться в сортирующем дереве в неподходящей позиции. Т.е. для этого конкретного элемента нарушается свойство пирамидальности. Но мы знаем, как следует поступить в ситуации подобного рода: ранее мы уже сталкивались с ней при работе со стандартной

очередью по приоритету. Если приоритет данного элемента выше приоритета его родительского элемента, мы перемещаем элемент в верхнюю часть сортирующего дерева за счет применения алгоритма пузырькового подъема. В противном случае мы сравниваем его с дочерними элементами. Если он меньше одного или обоих дочерних элементов, то при помощи алгоритма просачивания мы опускаем его в нижнюю часть сортирующего дерева. Со временем элемент окажется в позиции, где он будет меньше своего родительского и больше обоих дочерних элементов.

Отыскание произвольного элемента в сортирующем дереве

Теперь осталось решить первоначальную проблему: эффективно найти элемент в сортирующем дереве. Эта проблема кажется неразрешимой – сортирующее дерево не содержит никакой вспомогательной информации, поскольку оно было разработано лишь для обеспечения эффективного поиска наибольшего элемента. Возврат к сбалансированному дереву двоичного поиска (при использовании которого для поиска элемента за время, пропорциональное $O(\log(n))$, можно применить стандартный алгоритм поиска) кажется почти неизбежным.

Однако вместо этого мы создадим так называемое *косвенное сортирующее дерево* (*indirect heap*). При добавлении элемента в очередь по приоритету, управление этим элементом передается очереди. Взамен мы получаем *дескриптор* (*handle*). Дескриптор – это значение, по которому очередь "узнает" о добавлении элемента. Если хотите, дескриптор является косвенной ссылкой на реальный элемент в сортирующем дереве.

Итак, чтобы удалить элемент из очереди по приоритету, мы передаем очереди дескриптор этого элемента. Очередь использует этот дескриптор для выяснения позиции элемента в сортирующем дереве, а затем удаляет его, как было описано ранее.

Для изменения приоритета элемента мы просто изменяем значение приоритета элемента и сообщаем очереди о том, что произошло, передавая ей дескриптор элемента. Затем очередь может восстановить свойство пирамидальности. Операция исключения из очереди работает так же, как и ранее (дескриптор элемента не нужно передавать, поскольку очередь сама определит наибольший элемент). Однако очередь уничтожит дескриптор возвращенного элемента, поскольку он больше не присутствует в очереди. Если элементы являются записями или объектами, дескриптор данного элемента можно хранить внутри самого элемента наряду с приоритетом и другими полями.

В рамках операционной системы дескриптор, который, как правило, представляет собой своего рода замаскированный указатель, обычно имеет тип длинного целого. В рассматриваемой реализации мы используем всего лишь нетипизированный указатель.

Реализация расширенной очереди по приоритету

С точки зрения пользователя очереди по приоритету новый интерфейс лишь немногим сложнее рассмотренного ранее. Код интерфейса класса расширенной очереди по приоритету **TtdPriorityQueueEx** приведен в листинге 9.9.

Листинг 9.9. Интерфейс класса **TtdPriorityQueueEx**

```

type
  TtdPQHandle = pointer;

  TtdPriorityQueueEx = class
    private
      FCompare : TtdCompareFunc;
      FHandles : pointer;
      FList : TList;
      FName : TtdNameString;
    protected
      function pqGetCount : integer;
      procedure pqError(aErrorCode : integer;
                        const aMethodName : TtdNameString);
      procedure pqBubbleUp(aHandle : TtdPQHandle);
      procedure pqTrickleDown(aHandle : TtdPQHandle);
    public
      constructor Create(aCompare : TtdCompareFunc);
      destructor Destroy; override;
      procedure ChangePriority(aHandle : TtdPQHandle);
      procedure Clear;
      function Dequeue : pointer;
      function Enqueue(aItem : pointer) : TtdPQHandle;
      function Examine : pointer;
      function IsEmpty : boolean;
      function Remove(aHandle : TtdPQHandle) : pointer;
      property Count : integer read pqGetCount;
      property Name : TtdNameString read FName write FName;
  end;

```

Как видите, единственное реальное различие между этим классом и классом **TtdPriorityQueue** состоит в наличии методов **Remove** и **ChangePriority** и в том, что метод **Enqueue** возвращает дескриптор.

Так как же реализован этот интерфейс? Внутренне очередь, как обычно, содержит сортирующее дерево, но на этот раз она должна поддерживать определенную дополнительную информацию, чтобы иметь возможность отслеживать позицию каждого элемента в сортирующем дереве. Кроме того, очередь должна идентифицировать каждый элемент дескриптором, чтобы поиск элемента по заданному дескриптору выполнялся быстро и эффективно – теоретически быстрее, чем в дереве двоичного поиска, где время поиска определяется соотношением $O(\log(n))$.

Поэтому мы сделаем следующее: когда пользователь будет ставить элемент в очередь, мы будем добавлять элемент в связный список. Это будет сопряжено с определением и использованием узла и, по меньшей мере, двух указателей: указателя самого этого элемента и указателя следующего элемента, хотя по причинам, которые станут понятны несколько позже, мы будем использовать двухсвязный список и поэтому нам потребуется также и предыдущий указатель. Передаваемый нами обратно дескриптор элемента будет адресом узла. Теперь наступает важный момент. Узел хранит также целочисленное значение – позицию элемента в массиве, посредством которого реализовано сортирующее дерево. Сортирующее дерево не хранит сами элементы, а только их дескрипторы (т.е. узлы связного списка). Каждый раз, когда для выполнения сравнения ему нужно будет обратиться к самому элементу, оно будет разыменовывать дескриптор.

К сожалению, мы не можем использовать описанный в главе 3 класс связного списка, поскольку нам требуется доступ к узлам, а этот класс был разработан с целью сокрытия структуры узлов. Это один из случаев, когда нельзя использовать заранее стандартные классы и требуется выполнить кодирование от начала до конца. В случае применения двухсвязного списка это не так страшно, поскольку эта структура достаточно проста. Мы создадим связный список с явными начальным и конечным узлами. В результате удаление обычного узла превращается в исключительно простую задачу. Удаление узлов будет выполняться с применением обоих методов **Dequeue** и **Remove** класса расширенной очереди по приоритету.

Операции постановки в очередь и пузырькового подъема лишь немногим более сложны. Вначале мы создаем дескриптор, выделяя узел для элемента, и добавляем его в связный список узлов. Поскольку мы добавляем дескрипторы к сортирующему дереву, для доступа к элементам потребуется выполнять разыменование дескрипторов, а при перемещении элемента по сортирующему дереву индекс его новой позиции необходимо сохранять внутри узла. Код реализации методов постановки в очередь и пузырькового подъема приведен в листинге 9.10.

Листинг 9.10. Постановка в очередь и пузырьковый подъем в расширенной очереди по приоритету

```
procedure TtdPriorityQueueEx.pqBubbleUp(aHandle : pointer);
var
  FromInx : integer;
  ParentInx : integer;
  ParentHandle : PpqexNode;
  Handle : PpqexNode absolute aHandle;
begin
  {если анализируемый дескриптор больше дескриптора родительского элемента,
   нужно их нужно поменять местами и продолжить процесс с новой позиции}
  {Примечание: родительский узел дочернего узла, имеющего индекс n, имеет
   индекс (n-1)/2}
  FromInx := Handle^.peInx;
```

```

if (FromInx > 0) then begin
  ParentInx := (FromInx - 1) div 2;
  ParentHandle := PpqexNode(FList.List^[ParentInx]);
  {если элемент имеет родительский элемент и больше него...}
  while (FromInx > 0) and
    (FCompare(Handle^.peItem, ParentHandle^.peItem) > 0) do begin
    {нужно переместить родительский элемент вниз по дереву}
    FList.List^[FromInx] := ParentHandle;
    ParentHandle^.peInx := FromInx;
    FromInx := ParentInx;
    ParentInx := (FromInx - 1) div 2;
    ParentHandle := PpqexNode(FList.List^[ParentInx]);
  end;
end;
{сохранить элемент в правильной позиции}
FList.List^[FromInx] := Handle;
Handle^.peInx := FromInx;
end;

function TtdPriorityQueueEx.Enqueue(aItem : pointer) : TtdPQHandle;
var
  Handle : PpqexNode;
begin
  {создать новый узел для связного списка}
  Handle := AddLinkedListNode(FHandles, aItem);
  {добавить дескриптор в конец очереди}
  FList.Add(Handle);
  Handle^.peInx := pred(FList.Count);
  {теперь нужно выполнить его пузырьковый подъем на максимально возможный
  уровень}
  if (FList.Count > 1) then
    pqBubbleUp(Handle);
  {вернуть дескриптор}
  Result := Handle;
end;

```

Подобно методу **Enqueue**, все эти косвенные ссылки несколько усложняют метод **Dequeue**, но в коде все же можно распознать стандартные операции исключения из очереди и просачивания.

Листинг 9.11. Исключение из очереди и просачивание в расширенной очереди по приоритету

```

procedure TtdPriorityQueueEx.pqTrickleDown(aHandle : TtdPQHandle);
var
  FromInx : integer;
  MaxInx : integer;
  ChildInx : integer;
  ChildHandle : PpqexNode;
  Handle : PpqexNode absolute aHandle;

```

begin

{если анализируемый элемент меньше одного из своих дочерних элементов, его нужно поменять местами с большим дочерним элементом и продолжить процесс из новой позиции}

FromInx := Handle^.peInx;

MaxInx := pred(FList.Count);

{вычислить индекс левого дочернего узла}

ChildInx := succ(FromInx * 2);

{если имеется по меньшей мере правый дочерний элемент, необходимо вычислить индекс большего дочернего элемента...}

while (ChildInx <= MaxInx) **do begin**

{если есть хоть один правый дочерний узел, вычислить индекс наибольшего дочернего узла}

if ((ChildInx+1) <= MaxInx) **and**

(FCompare(PpqexNode(FList.List^[ChildInx])^.peItem,
PpqexNode(FList.List^[ChildInx+1])^.peItem) < 0) **then**

inc(ChildInx);

{если элемент больше или равен большему дочернему элементу, задача выполнена}

ChildHandle := PpqexNode(FList.List^[ChildInx]);

if (FCompare(Handle^.peItem, ChildHandle^.peItem) >= 0) **then**

Break;

{в противном случае больший дочерний элемент нужно переместить вверх по дереву, а сам элемент - вниз}

FList.List^[FromInx] := ChildHandle;

ChildHandle^.peInx := FromInx;

FromInx := ChildInx;

ChildInx := succ(FromInx * 2);

end;

{сохранить элемент в правильной позиции}

FList.List^[FromInx] := Handle;

Handle^.peInx := FromInx;

end;

function TtdPriorityQueueEx.Dequeue : pointer;

var

Handle : PpqexNode;

begin

{проверить наличие элементов, которые нужно исключить из очереди}

if (FList.Count = 0) **then**

pqError(tdeQueueIsEmpty, 'Dequeue');

{вернуть корневой элемент, удалить его из списка дескрипторов}

Handle := FList.List^[0];

Result := Handle^.peItem;

DeleteLinkedListNode(FHandles, Handle);

{если очередь содержала только один элемент, теперь она пуста}

if (FList.Count = 1) **then**

FList.Count := 0

{если она содержала два элемента, нужно просто заменить корневой элемент одним из оставшихся дочерних элементов. Очевидно, что при этом свойство пирамидальности сохраняется}


```

else if (FList.Count = 2) then begin
    Handle := FList.List^[1];
    FList.List^[0] := Handle;
    FList.Count := 1;
    Handle^.peInx := 0;
end
{ в противном случае свойство пирамидальности требует восстановления }
else begin
    { заменить корневой узел дочерним узлом, расположенным в самой нижней,
    крайней справа позиции, и уменьшить размер списка; затем за счет
    применения метода просачивания переместить корневой узел как можно дальше
    вниз по дереву }
    Handle := FList.Last;
    FList.List^[0] := Handle;
    Handle^.peInx := 0;
    FList.Count := FList.Count - 1;
    pqTrickleDown(Handle);
end;
end;

```

После ознакомления с операциями постановки в очередь и исключения из нее можно рассмотреть новые операции: удаление и изменение приоритета. Метод **ChangePriority** крайне прост. Прежде чем метод будет вызван, класс предполагает, что приоритет элемента был изменен. Вначале метод проверяет, имеет ли элемент родительский элемент, и если да, то больше ли элемент с новым приоритетом своего родительского элемента. Если это так, то элемент перемещается вверх за счет применения метода пузырькового подъема. Если операция пузырькового подъема невозможна или не требуется, метод проверяет возможность выполнения операции просачивания.

Листинг 9.12. Восстановление свойства пирамидальности после изменения приоритета

```

procedure TtdPriorityQueueEx.ChangePriority(aHandle : TtdPQHandle);
var
    Handle : PpqexNode absolute aHandle;
    ParentInx : integer;
    ParentHandle : PpqexNode;
begin
    { проверить возможность выполнения операции пузырькового подъема }
    if (Handle^.peInx > 0) then begin
        ParentInx := (Handle^.peInx - 1) div 2;
        ParentHandle := PpqexNode(FList[ParentInx]);
        if (FCompare(Handle^.peItem, ParentHandle^.peItem) > 0) then begin
            pqBubbleUp(Handle);
            Exit;
        end;
    end;
end;

```

```

    { в противном случае выполнить операцию просачивания }
    pqTrickleDown(Handle);
end;

```

Последняя операция реализуется при помощи метода **Remove**. В данном случае мы возвращаем элемент, определенный дескриптором, а затем заменяем его последним элементом сортирующего дерева. Дескриптор удаляется из связного списка. Эта операция упрощается благодаря использованию двусвязного списка. Затем значение счетчика элементов в сортирующем дереве уменьшается на единицу. С этого момента процесс полностью совпадает с процессом изменения приоритета, поэтому мы просто вызываем соответствующий метод.

Листинг 9.13. Удаление элемента, заданного его дескриптором

```

function TtdPriorityQueueEx.Remove(aHandle : TtdPQHandle) : pointer;
var
    Handle : PpqexNode absolute aHandle;
    NewHandle : PpqexNode;
    HeapInx : integer;
begin
    { вернуть элемент, а затем удалить дескриптор }
    Result := Handle^.peItem;
    HeapInx := Handle^.peInx;
    DeleteLinkedListNode(FHandles, Handle);
    { выполнить проверку того, что был удален последний элемент. Если это так,
      нужно просто уменьшить размер сортирующего дерева – при этом свойство
      пирамидальности будет сохранено }
    if (HeapInx = pred(FList.Count)) then
        FList.Count := FList.Count - 1
    else begin
        { заменить элемент сортирующего дерева дочерним элементом, расположенным в
          самой нижней крайней справа позиции, и уменьшить размер списка }
        NewHandle := FList.Last;
        FList.List^[HeapInx] := NewHandle;
        NewHandle^.peInx := HeapInx;
        FList.Count := FList.Count - 1;
        { дальнейшие действия совпадают с выполнением операции изменения
          приоритета }
        ChangePriority(NewHandle);
    end;
end;

```

Полный код этого класса можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDPriQue.pas**.

Резюме

В этой главе мы уделили основное внимание очередям по приоритету – очередям, которые возвращают не самый первый помещенный в них элемент, а элемент с наивысшим приоритетом. Исследовав несколько простых реализаций, мы ознакомились с реализацией, предполагающей использование сортирующего дерева. Мы рассмотрели базовые свойства и операции сортирующего дерева и научились применять их в качестве алгоритма пирамидальной сортировки, так и для удовлетворения первоначального требования, предъявляемого к очереди по приоритету.

И, наконец, мы расширили определение очереди по приоритету для обеспечения выполнения ряда дополнительных операций: удаления произвольного элемента и изменения приоритета данного элемента. Мы выяснили, какие изменения нужно внести в реализацию с целью поддержки этих операций.



Глава 10

Конечные автоматы и регулярные выражения

Существует целый класс проблем, которые могут быть решены с помощью авторучки и бумаги. По-моему, это замечательный аспект программирования: иметь возможность графически представить какой-либо процесс, а затем закодировать его. Я имею в виду алгоритмы, в которых используются конечные автоматы.

Конечные автоматы

В отличие от большинства рассмотренных в этой книге алгоритмов, конечные автоматы – это технологии, призванные облегчать разработку других алгоритмов. Они служат средством достижения конечной цели – реализации алгоритма. Тем не менее, как будет показано, они обладают рядом интересных особенностей. В основном мы будем рассматривать конечные автоматы, которые реализуют алгоритмы *синтаксического анализа* (*parsing algorithm*). Синтаксический анализ означает считывание строки (или текстового файла) и разбиение последовательностей символов на отдельные лексемы. Конечный автомат, который выполняет синтаксический анализ, обычно называют *синтаксическим анализатором* (*parser*).

Использование конечного автомата: синтаксический анализ

Чтобы лучше понять весь процесс, рассмотрим пример. Предположим, что требуется разработать алгоритм, который должен извлекать отдельные слова из строки текста. Извлекаемые слова будут помещаться в список строк. Более того, желательно, чтобы внутри строки текст, заключенный в кавычки, воспринимался как одно слово. Т.е., если имеется строка:

He said, "State machines?"

процедура должна игнорировать знаки препинания и пробелы и возвращать следующее:

```
He
said
"State machines?"
```

Обратите внимание, что пробел и вопросительный знак внутри заключенного в кавычки текста остались без изменений.

Простейший способ реализации этого конкретного алгоритма – использование конечного автомата. *Конечный автомат (state machine)* – это система (обычно цифровая), которая переходит из одного состояния в другое в соответствии с принимаемыми ею входными данными (сигналами). Смена состояний называется *переходом (transition)*. Конечный автомат можно представить специальной блок-схемой. Блок-схема рассматриваемого алгоритма показана на рис. 10.1.

Показанный на рисунке конечный автомат имеет три состояния: А, В и С. Работа блок-схемы начинается с состояния А. В этом состоянии выполняется считывание символа из входной строки. Если этот символ – двойная кавычка, осуществляется переход в состояние В. Если символ является пробелом или знаком препинания, выполняется переход в состояние С. Если это любой другой символ, конечный автомат остается в состоянии А (это показано петлей).

После перехода в состояние В считывание символов продолжается в нем до тех пор, пока не будет считан символ закрывающей двойной кавычки. В этот момент происходит переход обратно в состояние А.

С другой стороны, если был выполнен переход в состояние С, считывание символов продолжается в этом состоянии до тех пор, пока не произойдет одно из двух: либо не будет выполнено считывание символа двойной кавычки, в результате чего произойдет переход в состояние В, либо не будет выполнено считывание символа, который не является ни двойной кавычкой, ни пробелом, ни знаком препинания, в результате чего будет осуществлен переход в состояние А.

Во время перехода может требоваться также выполнение какого-либо действия. Предположим, что мы используем строку для накапливания символов текущего слова. Первоначальный переход в состояние А очистит эту строку. Циклический переход из состояния А в состояние А допишет символ к текущему слову. Переход из

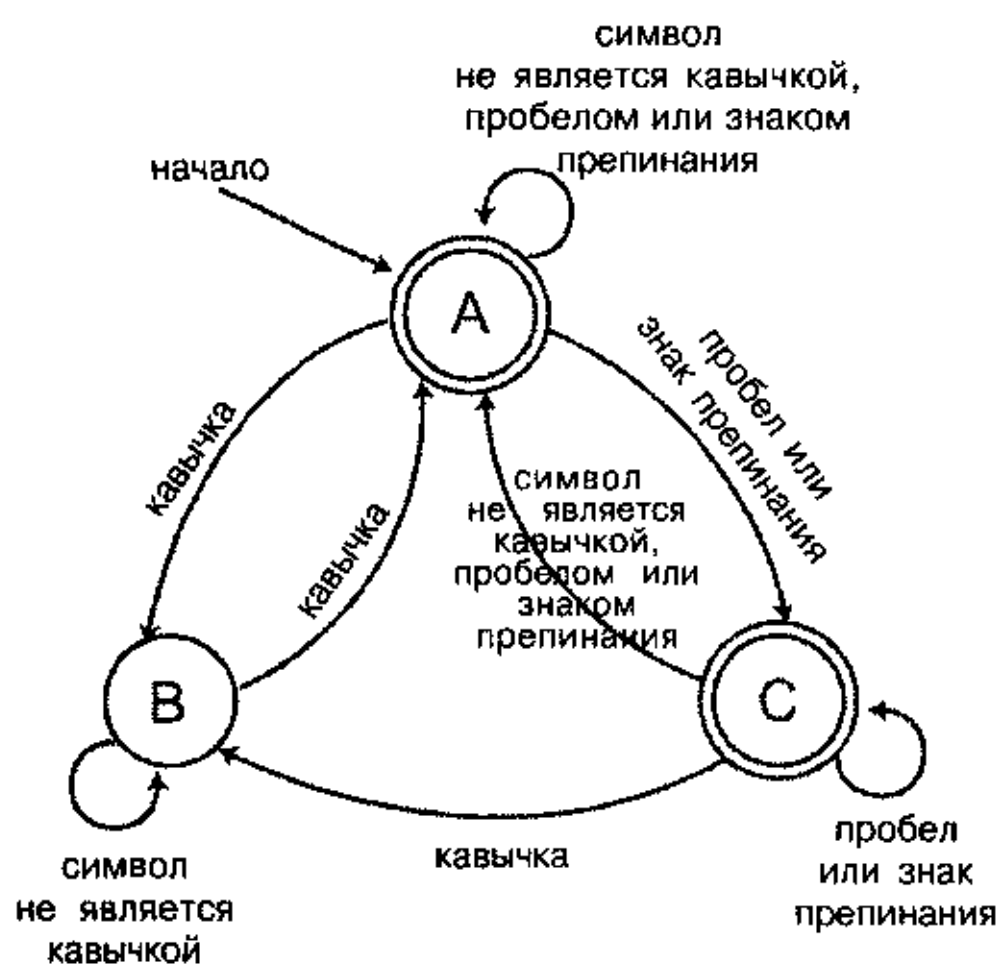


Рисунок 10.1. Конечный автомат извлечения слов из строки

состояния А в состояние В вначале добавит текущее слово (если таковое имеется) к списку строк, а затем установит в качестве текущего слова открывающую двойную кавычку. Циклический переход из состояния В в это же состояние допишет символ к текущему слову. Переход из состояния В обратно в состояние А допишет закрывающую двойную кавычку к текущему слову, добавит его в список строк, а затем очистит текущее слово. При переходе из состояния А в состояние С текущее слово добавляется в список строк, а затем очищается. Переход из состояния С в это же состояние не вызывает никаких действий (именно во время этого перехода происходит действительное отбрасывание пробелов и знаков препинания). При переходе из состояния С в состояние А значение текущего слова устанавливается равным считываемому символу. При переходе из состояния С в состояние В текущее слово устанавливается равным открывающей двойной кавычке.

Проанализировав рисунок 10.1, как это описано в предыдущем абзаце, легко убедиться, что конечный автомат прекрасно реализует рассматриваемый алгоритм.

Переход в состояние А; очистка слова

Считывание 'H'; сохранение состояния А; слово = 'H'

Считывание 'e'; сохранение состояния А; слово = 'He'

Считывание ' '; переход в состояние С; вывод слова 'He', очистка слова

Считывание 's'; переход в состояние А; слово = 's'

Считывание 'a'; сохранение состояния А; слово = 'sa'

Считывание 'i'; сохранение состояния А; слово = 'sai'

Считывание 'd'; сохранение состояния А; слово = 'said'

Считывание ', '; переход в состояние С; вывод слова 'said', очистка слова

Считывание ' '; сохранение состояния С

Считывание '"'; переход в состояние А; слово = '"'

Считывание 'S'; сохранение состояния В; слово = '"S'

... и т.д. ...

Однако, блок-схема конечного автомата, показанная на рис. 10.1, обладает еще одной особенностью, о которой еще ничего не было сказано. Состояния А и С обозначены двойными окружностями, в то время как состояние В – одинарной. По соглашению в диаграммах конечных автоматов двойные окружности используются для обозначения *конечного состояния* (называемого также *состоянием останова* (*halt state*) или *поглощающим состоянием* (*accepting state*)). Когда входная строка полностью считана, конечный автомат оказывается в особом состоянии (применительно к приведенному выше примеру строки заключительное состояние конечного автомата – состояние А). Если заключительное состояние является конечным, говорят что конечный автомат *поглощает* входную строку. Независимо от того, какие символы (или, точнее, *лексемы* (*tokens*)) были найдены во входной строке и какие при этом были осуществлены переходы, конечный автомат "понимает" строку. С другой стороны, если бы конечный автомат прекратил работу в незавершенном состоянии, строка не была бы принята (поглощена) и конечный автомат не понял бы ее.

В данном случае состояние В не является поглощающим состоянием. Что это означает на практике? Если в момент, когда входная строка исчерпана, конечный автомат находится в состоянии В, это означает, что был считан первый символ двойной кавычки, но не второй. Т.е. конечный автомат считывает строку, содержащую текст с непарным символом двойной кавычки. В зависимости от строгости алгоритма, эта ситуация может считаться ошибкой либо просто игнорироваться. В алгоритме, изображенном на рис. 10.1, она считается ошибкой.

Если говорить об ошибках, хотя в данном конкретном примере эта ситуация не отражена, возможно состояние, когда переход к конкретному символу или лексеме невозможен. Это немедленно привело бы к ошибке. В дальнейшем будет показано, как это свойство можно встроить в сам конечный автомат.

Вычертив блок-схему, теперь ее необходимо реализовать. Для простоты понимания мы немного изменим ее, чтобы считывание входной строки управляло конечным автоматом, а не чтобы каждое состояние приводило к считыванию следующего символа из входной строки. Это облегчит понимание процесса выхода из конечного автомата.

Код реализации конечного автомата, показанного на рис. 10.1, приведен в листинге 10.1 (полный исходный код можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStates.pas**). Обратите внимание, что было решено назвать состояния не абстрактно А, В и С, как на рисунке, а с использованием описательных имен *ScanNormal*, *ScanQuoted* и *ScanPunctuation* (соответственно, СчитываниеОбычныхСимволов, СчитываниеКавычек и СчитываниеЗнаковПрепинания).

Листинг 10.1. Извлечение слов из строки

```
procedure TDExtractWords(const S : string; aList : TStrings);
type
  TStates = (ScanNormal, ScanQuoted, ScanPunctuation);
const
  WordDelim = ' !<>[]{}(),./?;:-+=*&';
var
  State : TStates;
  Inx : integer;
  Ch : char;
  CurWord : string;
begin
  {инициализация путем очистки списка строк и начало работы в состоянии
   ScanNormal с пустым словом}
  Assert(aList <> nil, 'TDExtractWords: list is nil');
  aList.Clear;
  State := ScanNormal;
  CurWord := '';
```

```

{считывание всех символов строки}
for Inx := 1 to length(S) do begin
  {get the next character}
  Ch := S[Inx];
  {обработка в зависимости от состояния}
  case State of
    ScanNormal :
      begin
        if (Ch = '"') then begin
          if (CurWord <> '') then
            aList.Add(CurWord);
          CurWord := '';
          State := ScanQuoted;
        end
        else if (TDPosCh(Ch, WordDelim) <> 0) then begin
          if (CurWord <> '') then begin
            aList.Add(CurWord);
            CurWord := '';
          end;
          State := ScanPunctuation;
        end
        else
          CurWord := CurWord + Ch;
        end;
      ScanQuoted :
        begin
          CurWord := CurWord + Ch;
          if (Ch = '"') then begin
            aList.Add(CurWord);
            CurWord := '';
            State := ScanNormal;
          end;
        end;
      ScanPunctuation :
        begin
          if (Ch = '"') then begin
            CurWord := '';
            State := ScanQuoted;
          end
          else if (TDPosCh(Ch, WordDelim) = 0) then begin
            CurWord := Ch;
            State := ScanNormal;
          end
        end;
      end;
    end;
  end;

```

```
{если по достижении конца строки текущим состоянием является ScanQuoted, это  
означает несоответствие символа двойной кавычки}  
if (State = ScanQuoted) then  
    raise EtdStateException.Create(FmtLoadStr(tdeStateMismatchQuote,  
                                                [UnitName, 'TDExtractWords']));  
{если текущее слово не является пустым, добавить его в список}  
if (CurWord <> '') then  
    aList.Add(CurWord);  
end;
```

Код извлекает символ из входной строки, а затем входит в оператор **Case**, который переключает текущее состояние. Для каждого состояния предусмотрены операторы **If**, которые реализуют соответствующие действия и переходы в зависимости от значения текущего символа. В конце кода, если завершение программы происходит в состоянии **ScanQuoted**, генерируется исключение.

Этот код работает неэффективно в 32-разрядной среде Delphi. Код строит текущее слово посимвольно, используя строковую операцию **+**. Для длинных строк этот метод крайне неэффективен, поскольку операция вынуждена периодически перераспределять область памяти, в которой хранится строка, для размещения дополнительных символов. Первоначально строка пуста. Затем в нее добавляется первый символ. Поскольку пустая строка является нулевым указателем, под нее выделяется определенный объем памяти (в лучшем случае 8 байт), и строка изменяется, чтобы указывать на него. Символ добавляется в строку. После того, как в нее будет добавлено еще семь символов, выделенный под строку объем памяти должен быть перераспределен, чтобы в нее можно было поместить еще один символ. Еще одна причина низкой эффективности программы связана с операцией добавления символа. Компилятор генерирует код, обеспечивающий преобразование символа во временную односимвольную строку, а затем объединяет эти строки. Понятно, что преобразование символа в длинную строку требует выделения дополнительного объема памяти.

Оба описанных фактора приводят к снижению быстродействия программы **TDExtractWords**. Чтобы решить указанные проблемы, можно внести в код следующие изменения, хотя они и делают конечную цель менее очевидной, по крайней мере, с точки зрения программиста, отвечающего за сопровождение.

- Вместо того чтобы установить значение переменной **CurWord** равным **''**, необходимо вызвать метод **SetLength**, чтобы заранее распределить память под строку. В зависимости от конкретных требований, следует выбрать приемлемое значение, определяющее длину слова в байтах. (Например, приемлемым значением может быть длина символа **S**. Длина извлекаемого слова не может превышать это значение.)
- Необходимо поддерживать переменную **Curlnx**, определяющую позицию следующего символа. Ее начальным значением должен быть ноль.
- Для каждого добавляемого символа необходимо увеличивать значение **Curlnx** и устанавливать значение **CurWord[Curlnx]** равным символу.

- Когда требуется добавить текущее слово в список строк, необходимо снова вызвать метод `SetLength`, на этот раз передавая ему значение переменной `CurInx`. В результате длина строки будет устанавливаться равной количеству символов в строке. Затем значение `CurInx` необходимо переустановить равным нулю.

Применяя этот алгоритм, мы сознательно пытаемся минимизировать количество операций перераспределения памяти для `CurWord` (нам удалось свести это количество до двух, что можно считать почти идеальным результатом) и предотвращаем автоматическое преобразование компилятором символа в длинную строку.

Как видите, код обеспечивает успешную реализацию конечного автомата. Кроме того, его очень легко расширить. Например, предположим, что должно учитываться также использование одинарных кавычек. Добиться этого достаточно просто: нужно создать новое состояние `D`, работающее таким же образом, как состояние `B`, за исключением того, что при переходе в это состояние и из него должны использоваться одинарные, а не двойные кавычки. Применительно к написанию кода это означает выполнение простого копирования и вставки с целью дублирования функций состояния `B` в состоянии `D`.

Синтаксический анализ файлов с разделяющими запятыми

Часто встречающаяся задача – необходимость выполнить синтаксический анализ файлов с запятыми-разделителями. Файл с запятыми-разделителями представляет собой текстовый файл, описывающий таблицу записей. Каждая строка в файле является отдельной записью, а сами строки делятся на поля записей, разделяемые одно от другого запятыми. (Иногда эту организацию файла называют форматом CSV (comma-separated values – значения, разделяемые запятыми).) При решении этой задачи возникает ряд затруднений (как всегда!). Поле может быть окружено кавычками (в результате значение поля может содержать запятые). Поле может отсутствовать – в этом случае две запятые означают, что поля следуют одно за другим.

Ниже приведен пример строки текста в формате CSV.

```
Julian,Bucknall,,43,"Author, and Columnist"
```

Эта строка содержит пять полей. Первые два поля содержат значения `[Julian]` и `[Bucknall]`, третье поле не имеет значения, значение четвертого поля – `[43]`, а пятого – `[Author, and Columnist]`. (В данном случае строковые значения заключены в квадратные скобки для показа того, что двойные кавычки в исходной строке отбрасываются.)

Будем считать, что конечной целью является создание подпрограммы, которая принимает строку и список строк, разбивает строку на отдельные поля и вставляет поля в список строк. Прежде чем приступить к созданию диаграммы конечного автомата, давайте сформулируем несколько правил в отношении допустимого

формата строки CSV. Во-первых, все символы являются значащими, и единственные отбрасываемые символы – запятые (естественно, после того, как они были использованы для разбиения текста CSV) и двойные кавычки, в которые заключено значение поля. Более того, двойная кавычка имеет значение открывающей двойной кавычки, если она расположена за запятой (или является первым символом строки). В частности, например, это правило означает, что если бы в приведенном примере строки между запятой и открывающей двойной кавычкой имелся один пробел, подпрограмма разбила бы строку на шесть полей, двумя последними из которых были бы ["Author] и [and Columnist"]. Более того, если бы двойная кавычка была идентифицирована в качестве открывающей двойной кавычки, то следующая двойная кавычка закрывала бы значение поля, а следующим символом должна была бы быть запятая (или конец строки). В противном случае имеет место ошибка, и строка усекается.

Теперь можно нарисовать блок-схему конечного автомата. На рис. 10.2 отражены пять состояний. Начальное состояние названо **FieldStart**. Если следующий символ – двойная кавычка, выполняется переход в состояние **ScanQuoted**, в котором выполняется отбор символов до тех пор, пока не встретится следующая двойная кавычка и не будет выполнен переход в состояние **EndQuoted**. Если следующий символ – запятая, можно снова выполнить переход в состояние **FieldStart**. Если это не так, выполняется переход в состояние ошибки, и выполнение программы прекращается. Пребывая в состоянии **FieldStart**, мы также можем получить запятую (поле считается пустым). Или, если мы получаем символ, который не является запятой или двойной кавычкой, осуществляется переход в состояние **ScanField**. В этом состоянии выполняется ввод и накопление символов до тех пор, пока не будет получена запятая.

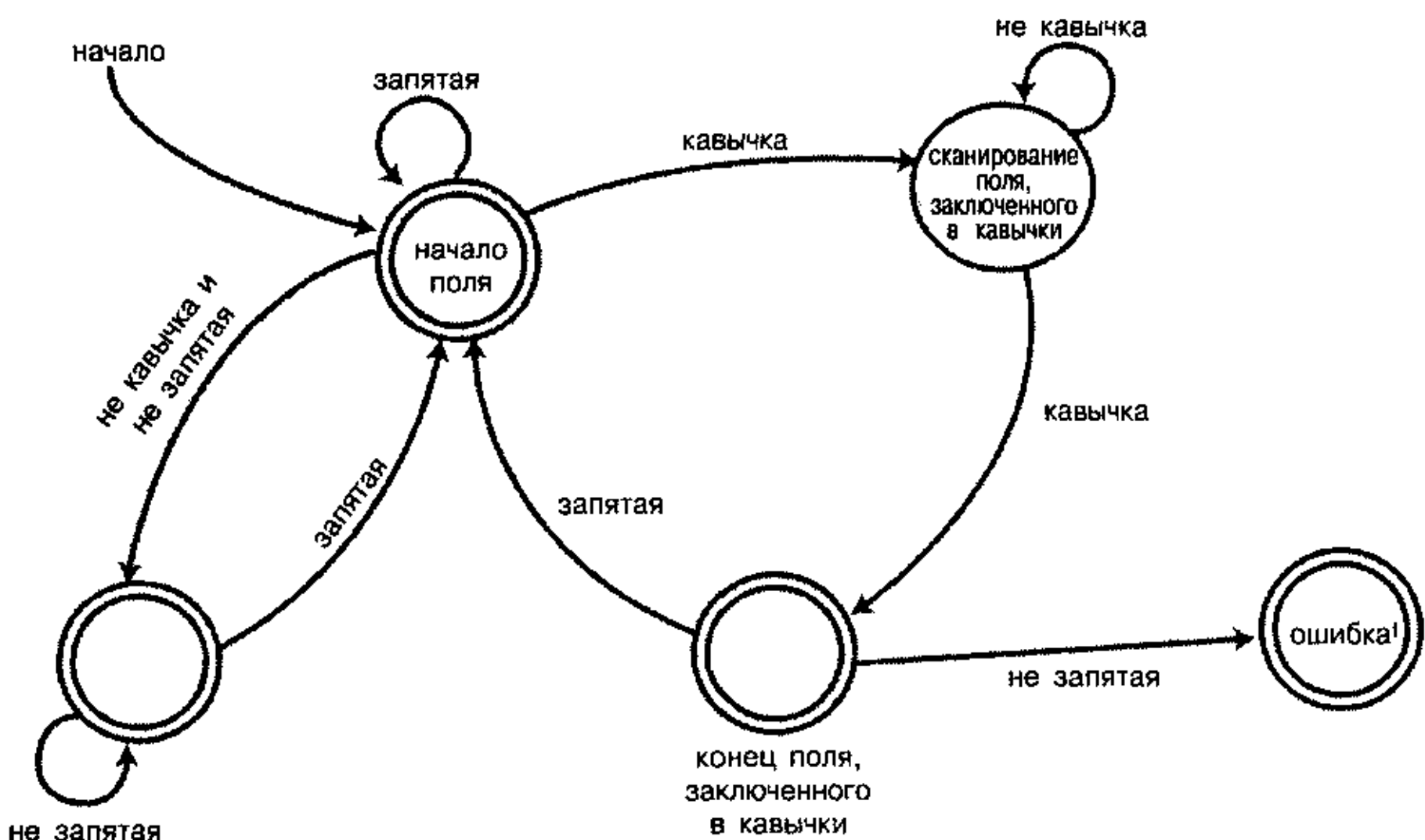


Рисунок 10.2. Конечный автомат синтаксического анализа строки в формате CSV


```

ScanField :
  begin
    if (Ch = ',') then begin
      aList.Add(CurField);
      CurField := '';
      State := FieldStart;
    end
    else
      CurField := CurField + Ch;
    end;
ScanQuoted :
  begin
    if (Ch = '"') then
      State := EndQuoted
    else
      CurField := CurField + Ch;
    end;
EndQuoted :
  begin
    if (Ch = ',') then begin
      aList.Add(CurField);
      CurField := '';
      State := FieldStart;
    end
    else
      State := GotError;
    end;
GotError :
  begin
    raise EtdStateException.Create(FmtLoadStr(tdeStateBadCSV,
      [UnitName, 'TDEextractFields']));
  end;
end;
end;
end;
{нахождение в состоянии ScanQuoted или GotError на момент окончания строки
свидетельствует о наличии проблемы, связанной с закрывающей кавычкой}
if (State = ScanQuoted) or (State = GotError) then
  raise EtdStateException.Create(FmtLoadStr(tdeStateBadCSV,
    [UnitName, 'TDEextractFields']));
{если текущее поле не пусто, добавить его в список}
if (CurField <> '') then
  aList.Add(CurField);
end;

```

Исходный код **TDEextractFields** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStates.pas**.

Детерминированные и недетерминированные конечные автоматы

Теперь, когда мы рассмотрели несколько достаточно сложных конечных автоматов и ближе познакомимся с ними, следует ознакомиться с рядом новых терминов. Первый из них – *автомат* (*automaton* или, в просторечии, *automata*). Это всего лишь еще одно название машины состояний, которое используется исключительно в учебных курсах и учебниках по компьютерным наукам. *Конечный автомат* (он же и *конечная машина состояний*) – это всего лишь машина состояний, количество состояний которой не бесконечно. Оба приведенные ранее примера представляли конечные автоматы: в первом имелось три состояния, во втором – пять.

И еще один новый термин – *детерминированный* (*deterministic*). Взгляните на конечный автомат, представленный на рис. 10.2. Независимо от текущего состояния и от того, каким будет следующий символ, точно известно, в какое состояние должен быть выполнен переход. Все переходы полностью определены. Этот конечный автомат является детерминированным. В процессе его работы не требуется делать какие-либо предположения или осуществлять выбор. Например, если бы двойная кавычка была получена во время нахождения в состоянии **FieldStart**, потребовалось бы выполнить переход в состояние **ScanQuoted**.

Рисунки 10.1 и 10.2 служат примерами *детерминированных конечных машин состояний* (*deterministic finite state machines – DFSM*), или *детерминированных конечных автоматов* (*deterministic finite automata – DFA*). Противоположными им являются конечные автоматы, в ряде состояний которых требуется осуществлять какой-либо выбор. При использовании конечного автомата этого типа приходится решать, нужно ли для данного конкретного символа выполнять переход в состояние X или в состояние Y. Как можно догадаться, реализация конечного автомата такого вида требует несколько более сложного кода. Не удивительно, что эти конечные автоматы называются *недетерминированными конечными машинами состояний* (*non-deterministic finite state machines – NDFSM*), или *недетерминированными конечными автоматами* (*deterministic finite automata – NFA*).

Теперь рассмотрим NFA-автомат. На рис. 10.3 показан NFA-автомат, который может преобразовывать строку, содержащую число в десятичном формате, в двоичное значение. При взгляде на этот рисунок у читателей может возникнуть вопрос, что представляют собой переходы, обозначенные странным символом ϵ . Это – *бесплатные*, или *свободные переходы*, которые можно выполнить без использования текущего символа или лексемы. Так, например, от начала лексемы A к следующей лексеме B можно перейти, используя знак "+", знак "-" или просто выполнив этот переход (бесплатный переход). Эти свободные переходы – отличительная особенность недетерминированных конечных автоматов.

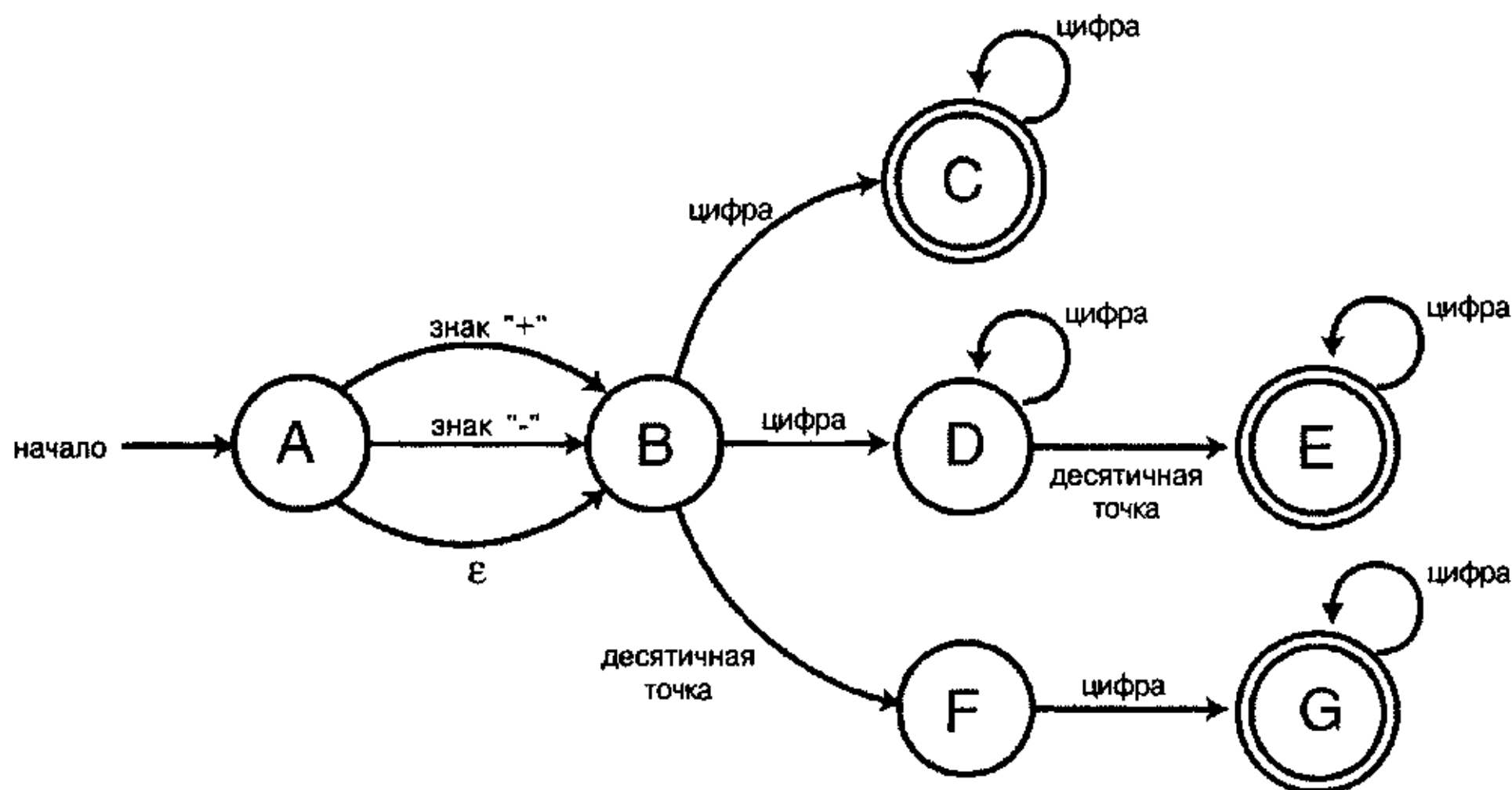


Рисунок 10.3. NFA-автомат для проверки, является ли строка числом

Воспользуемся этим рисунком для проверки таких строк, как "1", "1.23", "+.7", "-12". Как видите, верхняя ветвь служит для обработки целочисленных значений (не содержащих десятичной точки). Средняя ветвь выполняет обработку строк, которые состоят, по меньшей мере, из одной цифры, предшествующей десятичной точке, но которые могут и не иметь цифр, следующих за точкой. Нижняя ветвь предназначена для обработки строк, которые могут не содержать ни одной цифры перед десятичной точкой, но обязательно должны содержать хотя бы одну цифру после нее. Если немного подумать, становится понятно, что этот конечный автомат не сможет воспринимать самостоятельно вводимую десятичную точку.

Однако одна проблема остается нерешенной: хотя конечный автомат воспримет строку "1.2", как он "узнает", что нужно выполнять среднюю ветвь? Более того, может возникнуть более принципиальный вопрос: зачем вообще связываться с NFA-автоматом? Весь алгоритм кажется слишком сложным. Поэтому, почему бы не ограничиться применением DFA-автомата?

В действительности на второй вопрос проще ответить, чем на первый. NFA – естественные конечные автоматы для вычисления регулярных выражений. Разобравшись в использовании NFA-автоматов, мы проходим более половины пути к конечной цели этой главы – к возможности сопоставления строки с регулярным выражением.

Вернемся к первому вопросу: откуда NFA-автомат знает, что для строки "1.2" необходимо выполнять среднюю ветвь алгоритма? Естественно, автомат этого не знает. Существует несколько способов обработки строки с помощью подобного конечного автомата. И простейшим для описания является алгоритм проб и ошибок. В качестве вспомогательного мы используем еще один алгоритм – *алгоритм с отходом* (*backtracking algorithm*).

Обратите внимание, что нас интересует определение только одного пути конечного автомата, воспринимающего строку. Могут существовать и другие, но перечисление их всех интереса для нас не представляет.

Посмотрим, как работает этот алгоритм, проследив, что происходит при попытке ввода строки "12.34".

Работа алгоритма начинается с состояния А. Первой лексемой является "1". Мы не можем выполнить ни переход "+" в состояние В, ни переход "-". Поэтому мы выполняем свободный переход (связь ϵ). В результате автомат оказывается в состоянии В с той же лексемой "1". Теперь у нас имеются две возможности: выполнить переход в состояние С или в состояние D, поглощая при этом лексему. Выберем первую возможность. Прежде чем выполнить переход, отметим, что именно мы собираемся сделать, чтобы в случае неудачи не повторять ошибку. Итак, мы выполняем переход в состояние С, поглощая при этом лексему. Мы получаем вторую лексему, "2". Пока все достаточно просто. Автомат остается в том же состоянии и использует лексему.

Мы получаем следующую лексему ".". Теперь возможные переходы вообще отсутствуют. Мы оказались в тупике. Возможные переходы отсутствуют, но имеется лексема, которую нужно обработать. Именно здесь выступает на сцену алгоритм с отходом. Просмотрев свои заметки, мы замечаем, что в состоянии В был сделан выбор, при котором была предпринята попытка использования лексемы "1". Вероятно, этот выбор был ошибочным, поэтому мы осуществляем отход, чтобы найти правильное решение. Мы сбрасываем конечный автомат обратно в состояние В, а значение входной строки – в значение лексемы "1". Поскольку выбор первой возможности привел к проблеме, мы проверяем вторую возможность: переход в состояние D. Мы выполняем этот переход, поглощая лексему "1". Следующая лексема – "2". Мы используем ее и остаемся в состоянии D. Следующая лексема – ".": она обуславливает переход в состояние E, которое фактически поглощает следующие две цифры. Входная строка исчерпана и NFA-автомат находится в конечном состоянии. Поэтому можно сказать, что NFA-автомат воспринимает строку "12.34".

При преобразовании этого конечного автомата в код потребуются решить несколько проблем.

Во-первых, мы больше не располагаем простым циклом **For** для циклической обработки символов в строке. В случае применения детерминированного автомата каждый считываемый из входной строки символ вызывал переход (даже если это переход в то же самое состояние) и отсутствовала какая-либо возможность отхода или возврата к уже посещенному символу. В случае применения недетерминированного конечного автомата мы заменяем цикл **For** циклом **While** и при необходимости обеспечиваем увеличение переменной индекса строки.

Во-вторых, в некоторых состояниях мы не можем использовать применительно к входному символу простой оператор **Case** или **If**. Нам приходится иметь дело с множеством "вариантов перехода". Некоторые из них будут немедленно отбрасываться, поскольку текущий символ не соответствует условию перехода. Другие будут приняты, причем некоторые из них будут отброшены на более позднем этапе, а какой-то вариант будет использован. А пока просто пронумеруем возможные переходы и поочередно их выполним. Для этого будем использовать целочисленную переменную.

Теперь нужно рассмотреть последний фрагмент кода: реализацию собственно алгоритма с отходом. При каждом выборе допустимого перехода (сравните его с отбрасыванием перехода из-за того, что текущий символ не соответствует условиям перехода) необходимо сохранить информацию о конкретном выполненном переходе. Тогда, при необходимости выполнить отход к тому же состоянию с тем же самым входным символом, можно легко выбрать следующий переход и проверить его. Конечно, выбор вариантов переходов может требоваться в любом состоянии. Поэтому нужно записать их все, чтобы их можно было выполнить в обратном порядке. Отход выполняется в состояние, предшествовавшее последнему сделанному выбору. Иначе говоря, следует воспользоваться структурой типа "последним вошел, первым вышел", т.е. стеком. Применим один из стеков, которые были реализованы в главе 3.

Что же нужно сохранять в стеке? Разумеется, в нем необходимо сохранять состояние, в котором был сделан выбор, номер выполненного перехода (чтобы для проверки можно было выбрать следующий переход) и, наконец, индекс символа, для которого был осуществлен выбор. Используя эти три информационных элемента, можно легко вернуть конечный автомат к предшествующему состоянию, чтобы можно было выбрать следующий, и, возможно, более удачный вариант перехода.

Код реализации NFA-автомата для анализа десятичных чисел приведен в листинге 10.3. Этот конечный автомат будет поглощать строку в момент, когда строка исчерпана, а автомат находится в конечном состоянии. Автомат не примет строку, если строка исчерпана, а состояние отличается от конечного, или если в данном состоянии текущий символ не удовлетворяет условиям перехода. Во второй ситуации должно выполняться также следующее условие: стек отхода должен быть пуст.

Листинг 10.3. Проверка того, что строка является числом, с помощью NFA-автомата

type

```
TnfaState = (StartScanning,      {состояние А на рисунке}  
             ScannedSign,        {состояние В на рисунке}  
             ScanInteger,        {состояние С на рисунке}  
             ScanLeadDigits,     {состояние D на рисунке}  
             ScannedDecPoint,    {состояние E на рисунке}  
             ScanLeadDecPoint,   {состояние F на рисунке}  
             ScanDecimalDigits); {состояние G на рисунке}
```

```

PnfaChoice = ^TnfaChoice;
TnfaChoice = packed record
    chInx : integer;
    chMove : integer;
    chState : TnfaState;
end;
procedure DisposeChoice(aData : pointer); far;
begin
    if (aData <> nil) then
        Dispose(PnfaChoice(aData));
    end;
procedure PushChoice(aStack : TtdStack;
                    aInx : integer;
                    aMove : integer;
                    aState : TnfaState);
var
    Choice : PnfaChoice;
begin
    New(Choice);
    Choice^.chInx := aInx;
    Choice^.chMove := aMove;
    Choice^.chState := aState;
    aStack.Push(Choice);
end;
procedure PopChoice(aStack : TtdStack;
                    var aInx : integer;
                    var aMove : integer;
                    var aState : TnfaState);
var
    Choice : PnfaChoice;
begin
    Choice := PnfaChoice(aStack.Pop);
    aInx := Choice^.chInx;
    aMove := Choice^.chMove;
    aState := Choice^.chState;
    Dispose(Choice);
end;
function IsValidNumberNFA(const S : string) : boolean;
var
    StrInx : integer;
    State : TnfaState;
    Ch : AnsiChar;
    Move : integer;
    ChoiceStack : TtdStack;
begin
    {предположим, что число является недопустимым}
    Result := false;
    {инициализировать стек вариантов}
    ChoiceStack := TtdStack.Create(DisposeChoice);

```



```
try
  {подготовиться к сканированию}
  Move := 0;
  StrInx := 1;
  State := StartScanning;
  {считывание всех символов строки}
  while StrInx <= length(S) do begin
    {извлечь текущий символ}
    Ch := S[StrInx];
    {обработать в зависимости от состояния}
    case State of
      StartScanning :
        begin
          case Move of
            0 : {переход к ScannedSign по ветви +}
              begin
                if (Ch = '+') then begin
                  PushChoice(ChoiceStack, StrInx, Move, State);
                  State := ScannedSign;
                  Move := 0;
                  inc(StrInx);
                end
                else
                  inc(Move);
                end;
            1 : {переход к ScannedSign по ветви -}
              begin
                if (Ch = '-') then begin
                  PushChoice(ChoiceStack, StrInx, Move, State);
                  State := ScannedSign;
                  Move := 0;
                  inc(StrInx);
                end
                else
                  inc(Move);
                end;
            2 : {бесплатный переход к ScannedSign}
              begin
                PushChoice(ChoiceStack, StrInx, Move, State);
                State := ScannedSign;
                Move := 0;
              end;
          else
            {для этого состояния допустимые переходы отсутствуют}
            Move := -1;
          end;
        end;
      end;
```

```

ScannedSign :
begin
  case Move of
    0 : {переход к ScanInteger с использованием цифры}
      begin
        if TDIIsDigit(Ch) then begin
          PushChoice(ChoiceStack, StrInx, Move, State);
          State := ScanInteger;
          Move := 0;
          inc(StrInx);
        end
        else
          inc(Move);
        end;
    1 : {переход к ScanLeadDigits с использованием цифры}
      begin
        if TDIIsDigit(Ch) then begin
          PushChoice(ChoiceStack, StrInx, Move, State);
          State := ScanLeadDigits;
          Move := 0;
          inc(StrInx);
        end
        else
          inc(Move);
        end;
    2 : {переход к ScanLeadDigits с использованием десятичного
        разделителя}
      begin
        if (Ch = DecimalSeparator) then begin
          PushChoice(ChoiceStack, StrInx, Move, State);
          State := ScanLeadDecPoint;
          Move := 0;
          inc(StrInx);
        end
        else
          inc(Move);
        end;
      else
        {для этого состояния допустимые переходы отсутствуют}
        Move := -1;
      end;
    end;
  end;
end;

```

```
ScanInteger :  
  begin  
    case Move of  
      0 : {сохранить данное состояние для текущей цифры}  
        begin  
          if TDIIsDigit(Ch) then  
            inc(StrInx)  
          else  
            inc(Move);  
          end;  
        else  
          {для этого состояния допустимые переходы отсутствуют}  
          Move := -1;  
        end;  
      end;  
    end;  
ScanLeadDigits :  
  begin  
    case Move of  
      0 : {сохранить данное состояние для текущей цифры}  
        begin  
          if TDIIsDigit(Ch) then  
            inc(StrInx)  
          else  
            inc(Move);  
          end;  
      1 : {переход к ScanDecPoint с использованием десятичного  
          разделителя}  
        begin  
          if (Ch = DecimalSeparator) then begin  
            PushChoice(ChoiceStack, StrInx, Move, State);  
            State := ScannedDecPoint;  
            Move := 0;  
            inc(StrInx);  
          end  
          else  
            inc(Move);  
          end;  
        end;  
      else  
        {для этого состояния допустимые переходы отсутствуют}  
        Move := -1;  
      end;  
    end;  
  end;
```

ScannedDecPoint :

begin

case Move of

0 : {сохранить данное состояние для текущей цифры}

begin

if TDisDigit(Ch) **then**

inc(StrInx)

else

inc(Move);

end;

else

{для этого состояния допустимые переходы отсутствуют}

Move := -1;

end;

end;

ScanLeadDecPoint :

begin

case Move of

0 : {переход к ScanDecPoint с использованием цифры}

begin

if TDisDigit(Ch) **then begin**

PushChoice(ChoiceStack, StrInx, Move, State);

State := ScanDecimalDigits;

Move := 0;

inc(StrInx);

end

else

inc(Move);

end;

else

{для этого состояния допустимые переходы отсутствуют}

Move := -1;

end;

end;

ScanDecimalDigits :

begin

case Move of

0 : {сохранить данное состояние для текущей цифры}

begin

if TDisDigit(Ch) **then**

inc(StrInx)

else

inc(Move);

end;

else

{для этого состояния допустимые переходы отсутствуют}

Move := -1;

end;

end;

end;

```
{если для конкретного состояния допустимые переходы отсутствуют,  
выполнить отход за счет отказа от последнего выбора, и выполнить переход  
со следующим номером}  
if (Move = -1) then begin  
  {если стек пуст, возможность выполнения отхода отсутствует}  
  if ChoiceStack.IsEmpty then  
    Exit;  
  {отказаться от последнего выбора, выполнить следующий по порядку переход}  
  PopChoice(ChoiceStack, StrInx, Move, State);  
  inc(Move);  
end;  
end;  
{в этой точке число допустимо, если текущее состояние является конечным}  
if (State = ScanInteger) or  
  (State = ScannedDecPoint) or  
  (State = ScanDecimalDigits) then  
  Result := true;  
finally  
  ChoiceStack.Free;  
end;  
end;
```

Исходный код подпрограммы **IsValidNumberNFA** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStates.pas**.

Из листинга 10.3 видно, что базовая структура кода реализации всех состояний одинакова. Предполагается, что для каждого состояния существует ряд переходов, начиная с 0 (на рис. 10.3 переходы пронумерованы по ходу часовой стрелки). Для каждого состояния поочередно выполняется проверка возможности выполнения каждого из переходов. Если переход можно выполнить, сделанный выбор затапливается в стек, после чего переход выполняется. Если переход невозможен, предпринимается попытка выполнения следующего перехода.

Если нужно осуществить отход, мы выталкиваем верхний выбор из стека и проверяем возможность выполнения следующего перехода. Хранящаяся в стеке информация достаточна для восстановления состояния подпрограммы, существовавшего в момент выбора.

Для сравнения на рис. 10.4 показана блок-схема детерминированного автомата, который выполняет эту же проверку, а код, реализующий его, приведен в листинге 10.4.

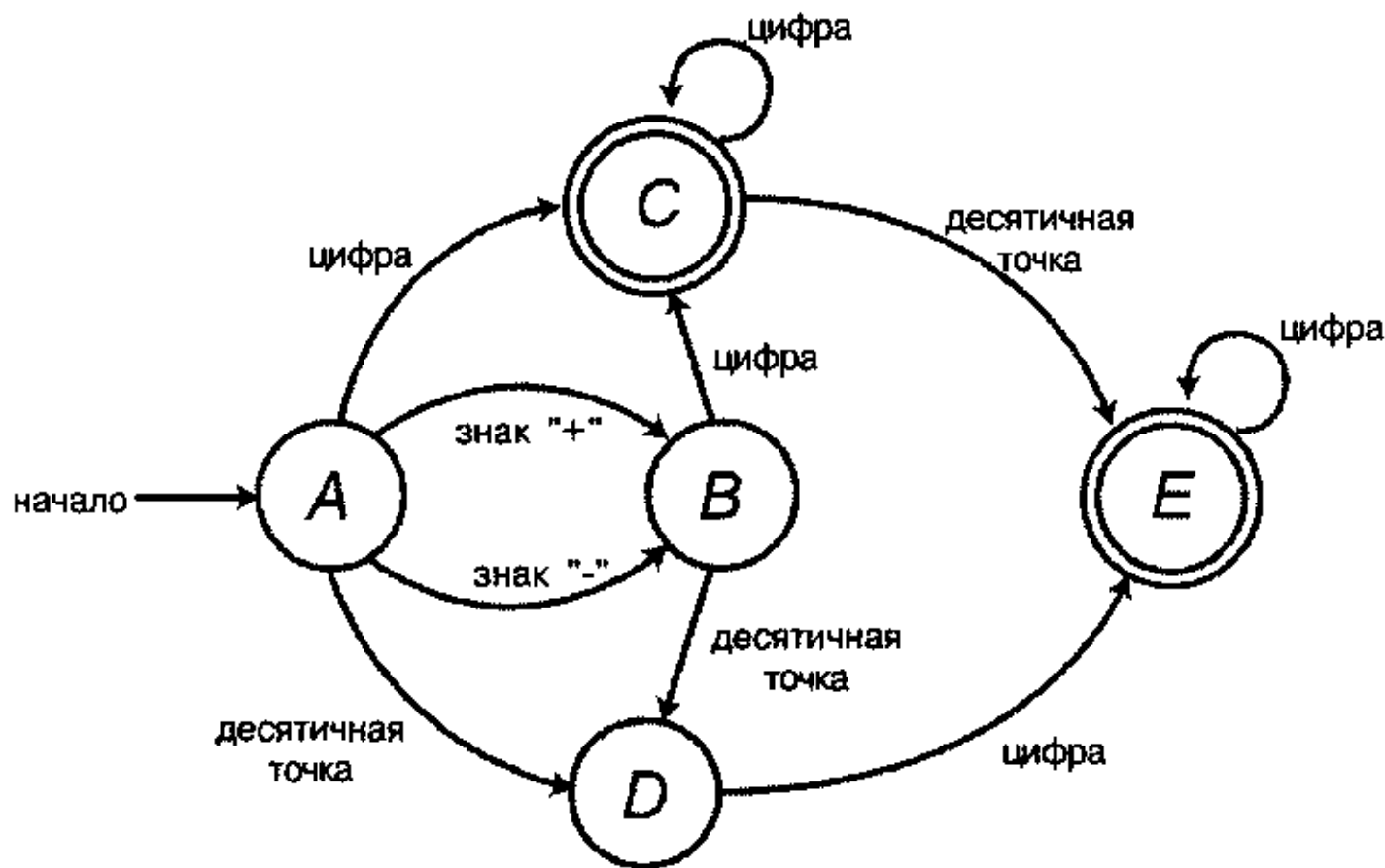


Рисунок 10.4. DFA-автомат для проверки, является ли строка числом

Листинг 10.4: Проверка того, что строка является числом, с помощью DFA-автомата

```

function IsValidNumber(const S : string) : boolean;
type
  TStates = (StartState, GotSign,
             GotInitDigit, GotInitDecPt, ScanDigits);
var
  State : TStates;
  Inx : integer;
  Ch : AnsiChar;
begin
  {предположим, что число является недопустимым}
  Result := false;
  {подготовиться к сканированию}
  State := StartState;
  {считывание всех символов строки}
  for Inx := 1 to length(S) do begin
    {извлечь текущий символ}
    Ch := S[Inx];
    {обработать в зависимости от состояния}
    case State of
      StartState :
        begin
          if (Ch = '+') or (Ch = '-') then
            State := GotSign
          else if (Ch = DecimalSeparator) then
            State := GotInitDecPt
          else if TDIIsdigit(Ch) then
            State := GotInitDigit
          else
            Exit;
        end;
    end;
  
```



```
GotSign :
begin
  if (Ch = DecimalSeparator) then
    State := GotInitDecPt
  else if TDIIsDigit(Ch) then
    State := GotInitDigit
  else
    Exit;
end;
GotInitDigit :
begin
  if (Ch = DecimalSeparator) then
    State := ScanDigits
  else if not TDIIsDigit(Ch) then
    Exit;
end;
GotInitDecPt :
begin
  if TDIIsDigit(Ch) then
    State := ScanDigits
  else
    Exit;
end;
ScanDigits :
begin
  if not TDIIsDigit(Ch) then
    Exit;
end;
end;
{в этой точке число допустимо, если текущее состояние является конечным}
if (State = GotInitDigit) or
  (State = ScanDigits) then
  Result := true;
end;
```

Исходный код подпрограммы **IsValidNumber** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStates**.

Если сравнить коды, приведенные в листингах 10.3 и 10.4, невозможно не заметить, что код NFA-автомата значительно сложнее. Он содержит целый набор вспомогательных подпрограмм, которые необходимо закодировать и поддерживать. Он также более чреват ошибками (необходимо побеспокоиться о поддержке стека, о возврате конечного автомата к предшествующему состоянию, о выборе следующего перехода и т.п.).

В общем случае, если требуется фиксированный, заранее определенный автомат, следует попытаться разработать и использовать детерминированный автомат. Следует попытаться свести реализацию недетерминированных автоматов к автоматическим алгоритмам. Реализация их вручную – чересчур трудоемкая задача.

Конечно, в рассмотренном примере NFA-автомат (и в примере его аналога DFA-автомата) мы всего лишь проверяем, является ли строка текстовым описанием целого числа или числа с плавающей точкой. Обычно желательно также вычислить интересующее число, а это усложняет код реализации переходов. Реализация этой функции при использовании DFA-автомата достаточно проста. Мы устанавливаем значение аккумуляторной (накопительной) переменной равным 0. При декодировании каждой цифры, расположенной перед десятичной точкой, мы умножаем значение аккумуляторной переменной на 10.0 и добавляем к нему значение новой цифры. Для цифр, следующих за десятичной точкой, мы поддерживаем значение счетчика текущего десятичного разряда и увеличиваем его на единицу при считывании каждой цифры. Для каждой такой цифры мы добавляем ее значение, умноженное на 0.1 в степени, соответствующей достигнутой десятичной позиции.

А как насчет NFA-автомата? Что ж, в этом случае решить задачу достаточно трудно. Вся сложность обусловлена необходимостью реализации алгоритма отхода. В любой момент времени внезапно может оказаться, что необходимо вернуться к предыдущему состоянию. В примере преобразования строки в число с плавающей точкой это не очень страшно: при заталкивании выбора в стек достаточно сохранить в нем и текущее значение аккумуляторной переменной (и значения всех необходимых дополнительных переменных). При выполнении отхода в качестве данных для восстановления состояния в момент неудачного выбора мы вытолкнем из стека и значение накопительной переменной.

Регулярные выражения

Теперь снова обратимся к теме, в связи с которой рассматривались NFA-автоматы. Поговорим о регулярных выражениях. Прежде всего, вспомним, что они собой представляют. По существу, *регулярные выражения* (*regular expression*) – это мини-язык простого описания шаблона, предназначенного для поиска текста (или, если говорить более строго, совпадающего с ним текста). В самой простой форме регулярное выражение состоит из слова или набора символов. Однако, используя стандартные метасимволы (или символы операций регулярного выражения), можно выполнять поиск более сложных шаблонов. Стандартными метасимволами являются "." (соответствует любому символу, кроме символа новой строки), "?" (соответствует нулю или более повторений предыдущего подвыражения), "*" (соответствует нулю или более повторений предыдущего подвыражения), "+" (соответствует одному или более повторений предыдущего подвыражения) и "|" (символ операции ИЛИ, которая устанавливает соответствие с левым или с пра-

вым подвыражением). Можно определить также класс символа для установки соответствия с одним из наборов символов. Если первым символом класса символов является "^", это означает отрицание класса. Т.е. символы класса не должны совпадать с остальными символами набора.

Правила представления регулярных выражений, с которыми мы будем работать, показаны на рис. 10.5. Они записаны в стандартной форме BNF (*Bachus-Naur Form – форма Бэкуса-Наура, БНФ*). "::=" означает "определено как", а "|" означает "ИЛИ". Следовательно, первая строка означает следующее: <выражение> является либо <членом>, либо <членом>, за которым следует символ вертикальной черты, а за ним – еще одно <выражение>. Вторая строка означает: <член> – это либо <коэффициент>, либо <коэффициент>, за которым следует <член>, и т.д. Это определение грамматических правил (они называются "грамматическими", поскольку определяют язык. Если обратиться к справочной системе Delphi, в ней можно найти грамматические правила языка Object Pascal. Они определены таким же образом.) может использоваться для генерирования подпрограммы вычисления регулярного выражения. Вскоре мы увидим, как это делается. А пока примите к сведению, что определение грамматических правил может использоваться для быстрой проверки того, что данное регулярное выражение является правильным.

Вероятно, лучше привести несколько примеров регулярных выражений. Это поможет понять их применение.

[a-zA-Z_][a-zA-Z0-9_]*

<выражение>	::=	<член>	
		<член> ' ' <выражение>	- дизъюнкция
<член>	::=	<коэффициент>	
		<коэффициент><член>	- конкатенация
<коэффициент>	::=	<элемент>	
		<элемент> '?'	- ноль или более
		<элемент> '*'	- ноль или более
		<элемент> '+'	- один или более
<элемент>	::=	<символ>	
		'.'	- любой символ
		' (' <выражение> ') '	- круглые скобки
		' [' <класс символов> '] '	- обычный класс
		' [^ ' <класс символов> '] '	- класс с отрицанием
<класс символов>	::=	<диапазон символов>	
		<диапазон символов><класс символов>	
<диапазон символов>	::=	<набор символов>	
		<набор символов> '-' <набор символов>	
<символ>	::=	<любой символ за исключением метасимволов>	
		' \ ' <вообще любой символ>	

Рис.10.5. Грамматические правила составления регулярных выражений, представленные в форме БНФ

Это регулярное выражение соответствует имени идентификатора в языке Pascal. Первое заключенное в квадратные скобки подвыражение – класс символов, из определения которого следует, что первым символом строки, для которой будет устанавливаться соответствие, должна быть буква, прописная или строчная, или символ подчеркивания. Второе заключенное в квадратные скобки подвыражение – еще один класс символов, совпадающий с первым, за исключением того, что в него добавлены цифры. Этот шаблон может повторяться ноль или более раз (что определено символом * в конце регулярного выражения). Таким образом, этому регулярному выражению соответствует буква или символ подчеркивания, за которой следует ноль или более букв, символов подчеркивания или цифр.

$(+|-)?[0-9]+(.[0-9]+)?$

Это регулярное выражение соответствует представлению целого числа или числа с плавающей точкой в языке Pascal. Оно означает необязательный знак, одну или более цифр и необязательную дробную часть. Дробная часть состоит из десятичной точки, за которой следует одна или более цифр. Если дробная часть отсутствует, число является целым. Если она присутствует, число является числом с плавающей точкой.

$\{[^{}}]*\}$

Этот последний пример регулярного выражения соответствует комментарию в языке Pascal, который помещается в фигурные скобки. Выражение означает наличие открывающей фигурной скобки, за которой следует ноль или более символов, ни один из которых не является закрывающей скобкой, а затем следует закрывающая фигурная скобка.

Использование регулярных выражений

Существует три этапа использования регулярного выражения. На первом регулярное выражение разбивается на составляющие его лексемы, на втором они преобразуются форму, пригодную для установки соответствия (компиляция регулярного выражения) и на заключительном этапе скомпилированная форма регулярного выражения используется для собственно установки соответствия со строками. Этот материал изложен в данной главе потому, что скомпилированная форма регулярного выражения реализуется в виде NFA-автомата.

Синтаксический анализ регулярных выражений

Последовательно рассмотрим три упомянутых выше этапа. В первую очередь необходимо решить проблему синтаксического анализа данной строки регулярного выражения. Целью этого процесса является простая проверка того, что строка регулярного выражения соответствует синтаксису, определенному грамматическими правилами.

Так как же, располагая определением грамматических правил и регулярным выражением, можно выполнить считывание символов строки и проверить регулярное выражение в целом на предмет соответствия грамматическим правилам? Проще всего создать для этого *нисходящий синтаксический анализатор* (*top-down parser*), который иногда еще называют *рекурсивным нисходящим синтаксическим анализатором* (*recursive descent parser*). При условии, что грамматические правила четко определены, эта задача достаточно проста.

При выполнении нисходящего синтаксического анализа каждая *продукция* (*production*) в грамматическом правиле становится отдельной подпрограммой. (Продукция – это одно из определений грамматики, т.е. одна из строк, содержащих символ операции "::<=".) Преобразуем первую продукцию грамматики (определяющую <выражение>) в метод **ParseExpr**.

Что же должен делать метод **ParseExpr**? Продукция утверждает, что <выражение> – это либо отдельный <член>, либо <член>, за которым следует символ вертикальной черты, а за ним еще один <член>. Предположим, что существует метод **ParseTerm**, который выполняет синтаксический анализ <члена>. В любом случае, прежде всего, необходимо вызвать эту подпрограмму для выполнения синтаксического анализа <члена>. Если после возврата из нее текущим символом является символ вертикальной черты, необходимо продолжить и рекурсивно вызвать подпрограмму **ParseExpr**, чтобы выполнить синтаксический анализ следующего <выражения>. Это все, что касается подпрограммы **ParseExpr**.

На некоторое время оставим без внимания реализацию метода **ParseTerm** (вскоре станет понятно, почему) и рассмотрим метод **ParseFactor**, выполняющий синтаксический анализ <коэффициента>. Как и в предыдущем случае, код достаточно прост. Вначале необходимо выполнить синтаксический анализ <элемента> путем вызова метода **ParseAtom**, а затем выполнить проверку на наличие одного из трех метасимволов: "*", "+" или "?". (*Метасимвол* – это символ, имеющий специальное значение с точки зрения грамматических правил – например, звездочка, знак плюса, круглые скобки и т.п. Другие символы не имеют никакого специального значения.) Кодирование метода **ParseAtom** достаточно тривиально. Элемент может быть <символом> или точкой; открывающей круглой скобкой, за которой следуют <выражение> и закрывающая круглая скобка; открывающей квадратной скобкой, за которой следуют <класс символов> и закрывающая квадратная скобка; открывающей квадратной скобкой, за которой следуют символ "^", <класс символов> и закрывающая квадратная скобка. Именно эту форму мы и реализуем в коде. Остальные методы, реализующие другие продукции, столь же просты. Обратите внимание, что в этих методах реальную проверку выполняет метод самого нижнего уровня. Например, метод **ParseAtom** будет проверять наличие закрывающей круглой скобки после того, как в результате синтаксического анализа обнаружены открывающая круглая скобка и <выражение>. Метод **ParseChar** удостоверяется, что текущий символ не является метасимволом. И так далее. Код, созданный в соответствии с приведенными рассуждениями, можно найти в листинге 10.5.

Листинг 10.5. Программа синтаксического анализа регулярных выражений

```

type
  TtdRegexParser = class
    private
      FRegexStr : string;
      {$IFDEF Delphi1}
      FRegexStrZ: PAnsiChar;
      {$ENDIF}
      FPosn : PAnsiChar;
    protected
      procedure rpParseAtom;
      procedure rpParseCCChar;
      procedure rpParseChar;
      procedure rpParseCharClass;
      procedure rpParseCharRange;
      procedure rpParseExpr;
      procedure rpParseFactor;
      procedure rpParseTerm;
    public
      constructor Create(const aRegexStr : string);
      destructor Destroy; override;
      function Parse(var aErrorPos : integer) : boolean;
  end;

constructor TtdRegexParser.Create(const aRegexStr : string);
begin
  inherited Create;
  FRegexStr := aRegexStr;
  {$IFDEF Delphi1}
  FRegexStrZ := StrAlloc(succ(length(aRegexStr)));
  StrPCopy(FRegexStrZ, aRegexStr);
  {$ENDIF}
end;

destructor TtdRegexParser.Destroy;
begin
  {$IFDEF Delphi1}
  StrDispose(FRegexStrZ);
  {$ENDIF}
  inherited Destroy;
end;

function TtdRegexParser.Parse(var aErrorPos : integer) : boolean;
begin
  Result := true;
  aErrorPos := 0;
  {$IFDEF Delphi1}
  FPosn := FRegexStrZ;
  {$ELSE}
  FPosn := PAnsiChar(FRegexStr);

```



```

{ $ENDIF}
try
  rpParseExpr;
  if (FPosn^ <> #0) then begin
    Result := false;
    { $IFDEF Delphi1}
    aErrorPos := FPosn - FRegexStrZ + 1;
    { $ELSE}
    aErrorPos := FPosn - PAnsiChar(FRegexStr) + 1;
    { $ENDIF}
  end;
except
  on E:Exception do begin
    Result := false;
    { $IFDEF Delphi1}
    aErrorPos := FPosn - FRegexStrZ + 1;
    { $ELSE}
    aErrorPos := FPosn - PAnsiChar(FRegexStr) + 1;
    { $ENDIF}
  end;
end;
end;

procedure TtdRegexParser.rpParseAtom;
begin
  case FPosn^ of
    '(' : begin
      inc(FPosn);
      writeln('open paren');
      rpParseExpr;
      if (FPosn^ <> ')') then
        raise Exception.Create('Regex error: expecting a closing
          parenthesis');
      inc(FPosn);
      writeln('close paren');
      end;
    '[' : begin
      inc(FPosn);
      if (FPosn^ = '^') then begin
        inc(FPosn);
        writeln('negated char class');
        rpParseCharClass;
      end
      else begin
        writeln('normal char class');
        rpParseCharClass;
      end;
      inc(FPosn);
    end;
  end;
end;

```

```

    '.' : begin
        inc(FPosn);
        writeln('any character');
    end;
else
    rpParseChar;
end; {case}
end;

procedure TtdRegexParser.rpParseCCChar;
begin
    if (FPosn^ = #0) then
        raise Exception.Create(
            'Regex error: expecting a normal character, found null terminator');
    if FPosn^ in [' ', '-'] then
        raise Exception.Create(
            'Regex error: expecting a normal character, found a metacharacter');
    if (FPosn^ = '\') then begin
        inc(FPosn);
        writeln('escaped ccchar ', FPosn^);
        inc(FPosn);
    end
    else begin
        writeln('ccchar ', FPosn^);
        inc(FPosn);
    end;
end;

procedure TtdRegexParser.rpParseChar;
begin
    if (FPosn^ = #0) then
        raise Exception.Create(
            'Regex error: expecting a normal character, found null terminator');
    if FPosn^ in MetaCharacters then
        raise Exception.Create(
            'Regex error: expecting a normal character, found a metacharacter');
    if (FPosn^ = '\') then begin
        inc(FPosn);
        writeln('escaped char ', FPosn^);
        inc(FPosn);
    end
    else begin
        writeln('char ', FPosn^);
        inc(FPosn);
    end;
end;
end;

```

```
procedure TtdRegexParser.rpParseCharClass;
begin
  rpParseCharRange;
  if (FPosn^ <> ']') then
    rpParseCharClass;
end;

procedure TtdRegexParser.rpParseCharRange;
begin
  rpParseCCChar;
  if (FPosn^ = '-') then begin
    inc(FPosn);
    writeln('--range to--');
    rpParseCCChar;
  end;
end;

procedure TtdRegexParser.rpParseExpr;
begin
  rpParseTerm;
  if (FPosn^ = '|') then begin
    inc(FPosn);
    writeln('alternation');
    rpParseExpr;
  end;
end;

procedure TtdRegexParser.rpParseFactor;
begin
  rpParseAtom;
  case FPosn^ of
    '?' : begin
      inc(FPosn);
      writeln('zero or one');
    end;
    '*' : begin
      inc(FPosn);
      writeln('zero or more');
    end;
    '+' : begin
      inc(FPosn);
      writeln('one or more');
    end;
  end; {case}
end;
```

Полный исходный код класса **TtdRegexParser** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDRegex.pas**.

Если вы просмотрите листинг 10.5, то увидите, что эта программа синтаксического анализа всего лишь выводит текущий грамматический элемент на экран монитора и генерирует исключение в ситуации, когда можно констатировать, что регулярное выражение неверно. Естественно, ни одно из этих действий не будет выполняться в реальной рабочей среде. Первое не будет выполняться потому, что нашей целью является компиляция регулярного выражения в код NFA-автомата, а второе – потому, что исключения не следует использовать для проверки, поскольку это слишком неэффективно. Тем не менее, этот код может служить иллюстрацией общей структуры упрощенного нисходящего синтаксического анализатора: вначале выполняется разработка грамматических правил, а затем достаточно простым образом они преобразуются в код.

Нам осталось только рассмотреть реализацию метода **ParseTerm**. По сравнению с уже реализованными методами эта реализация несколько сложнее. Проблема состоит в том, что согласно формулировке продукции, <член> является либо <коэффициентом>, либо <коэффициентом>, за которым следует еще один <член> (т.е. имеет место конкатенация). Не существует никакой операции, типа знака плюса или чего-то подобного, которая бы связывала два элемента. Если бы такая операция существовала, метод **ParseTerm** можно было бы реализовать так же, как были реализованы остальные методы **ParseXXXX**. Однако, поскольку никакого метасимвола выполнения конкатенации не существует, приходится прибегнуть к другому средству.

Рассмотрим проблему более внимательно. Предположим, что мы выполняем синтаксический анализ регулярного выражения "ab". Его нужно было бы проанализировать в качестве <выражения>, что означает анализ в качестве <члена>, затем <коэффициента>, затем <элемента>, а затем <символа>. В результате была бы выполнена обработка фрагмента "a". Затем грамматический разбор был бы продолжен, пока снова не было бы достигнуто определение <члена>, в котором говорится, что за первым <коэффициентом> может следовать еще один <член>. Продолжая анализ продукции, мы идентифицируем фрагмент "b" как <символ>, и на этом выполнение задачи завершается.

Все сказанное звучит достаточно просто. Так в чем же трудность? Выполним эти же действия для выражения "(a)". На этот раз синтаксический анализ продукции выполняется до тех пор, пока не будет достигнуто определение, согласно которому <элемент> может состоять из "(", за которой следует <выражение>, а за ним ")". Таким образом, обработка "(" завершается и снова начинается с синтаксического анализа верхней грамматической конструкции – <выражения>. Снова выполним нисходящий анализ: <выражение>, затем <член>, затем <коэффициент>, затем <элемент> и, наконец, <символ>. В результате выполняется обработка фрагмента "a". Снова возвращаясь к началу, мы встречаем альтернативное определение продукции <член>. Так почему бы на этот раз нам не обратиться к альтернативной ветви и не попытаться выполнить синтаксический анализ конкатенации?

Очевидно, что подобное делать нельзя, поскольку на этот раз текущим символом является ")". В первом примере мы решили выполнить синтаксический анализ конкатенации, поскольку текущим символом был "b", но на сей раз им является ")". Прежде чем решить, выполнять ли синтаксический анализ еще одного сцепленного <члена>, необходимо быстро проанализировать текущий символ. Если его можно считать началом еще одного <элемента>, то мы продолжаем обработку и анализируем его в качестве такового. Если же нет, мы считаем, что что-то другое (а именно вызывающий метод) выполнит с ним какие-либо действия, и что конкатенация отсутствует.

Этот процесс называют *разрывом грамматического правила* (*breaking the grammar*). Мы должны предположить, что если в данном случае конкатенация имеет место, текущий символ будет служить начальным символом <элемента>. Иначе говоря, если текущий символ – ".", "{", "[" или обычный символ, мы должны выполнить синтаксический анализ еще одного <члена>. Если же нет – мы считаем, что конкатенация отсутствует, и осуществляем выход из метода **ParseTerm**. Для определения того, что нужно делать с продукцией <член> (продукцией "более высокого" уровня), мы используем информацию продукции <элемент> (продукции "более низкого" уровня). Излишне повторять, что необходимость в таком подходе возникает только по причине отсутствия метасимвола конкатенации.

Код двух последних методов класса синтаксического анализатора регулярных выражений: метода **ParseTerm** и интерфейсного метода **Parse** показан в листинге 10.6.

Листинг 10.6. Методы **ParseTerm** и **Parse**

```
procedure TtdRegexParser.rpParseTerm;
begin
  rpParseFactor;
  if (FPosn^ = '(') or
     (FPosn^ = '[') or
     (FPosn^ = '.') or
     ((FPosn^ <> #0) and not (FPosn^ in MetaCharacters)) then
    rpParseTerm;
end;

function TtdRegexParser.Parse(var aErrorPos : integer) : boolean;
begin
  Result := true;
  aErrorPos := 0;
  {$IFDEF Delphi1}
  FPosn := FRegexStrZ;
  {$ELSE}
  FPosn := PAnsiChar(FRegexStr);
  {$ENDIF}
```

```

try
  rpParseExpr;
  if (FPosn^ <> #0) then begin
    Result := false;
    {$IFDEF Delphi1}
    aErrorPos := FPosn - FRegexStrZ + 1;
    {$ELSE}
    aErrorPos := FPosn - PAnsiChar(FRegexStr) + 1;
    {$ENDIF}
  end;
except
  on E:Exception do begin
    Result := false;
    {$IFDEF Delphi1}
    aErrorPos := FPosn - FRegexStrZ + 1;
    {$ELSE}
    aErrorPos := FPosn - PAnsiChar(FRegexStr) + 1;
    {$ENDIF}
  end;
end;
end;
end;

```

Итак, мы научились выполнять синтаксический анализ регулярного выражения. Теперь мы можем принять строку и вернуть информацию о том, образует ли она допустимое регулярное выражение.

Компиляция регулярных выражений

Следующий шаг состоит в создании NFA-автомата для регулярного выражения. Решение этой задачи мы начнем с создания блок-схемы конечного автомата выполнения регулярного выражения. Создание блок-схемы конечного автомата для конкретного регулярного выражения – достаточно простая задача. В общем случае правила языка утверждают, что регулярное выражение состоит из различных подвыражений (которые сами являются регулярными выражениями), скомпонованных или объединенных различными способами. Каждое подвыражение имеет единственное начальное состояние и единственное конечное состояние. И подобно тому, как это делается в конструкторе "Лего", эти простые строительные блоки собираются воедино, образуя все регулярное выражение. Блок-схема, приведенная на рис. 10.6, содержит конструкции, имеющие наибольшее значение.

Первый пример – конечный автомат, выполняющий распознавание отдельного символа алфавита. Второй пример столь же прост: он представляет собой конечный автомат, выполняющий распознавание любого символа алфавита (другими словами, это операция "."). Четвертая конструкция служит иллюстрацией того, как выполняется конкатенация (одного выражения, за которым следует второе). При этом мы просто объединяем начальное состояние второго подвыражения с конечным состоянием первого. Следующей показана конструкция, выполняющая

дизъюнкцию. Мы создаем новое начальное состояние и получаем два возможных бесплатных перехода, по одному для каждого из подвыражений. Конечное состояние первого подвыражения объединяется с конечным состоянием второго подвыражения, и это последнее состояние становится конечным состоянием всего выражения. Следующий конечный автомат реализует операцию "?": в данном случае мы создаем новое начальное состояние с двумя ветвями ϵ ; первая выполняет соединение с начальным состоянием подвыражения, а вторая – с его конечным состоянием. Это конечное состояние является конечным состоянием всего выражения. Вероятно, наиболее сложными конструкциями являются конечные автоматы для выполнения операций "+" и "*".

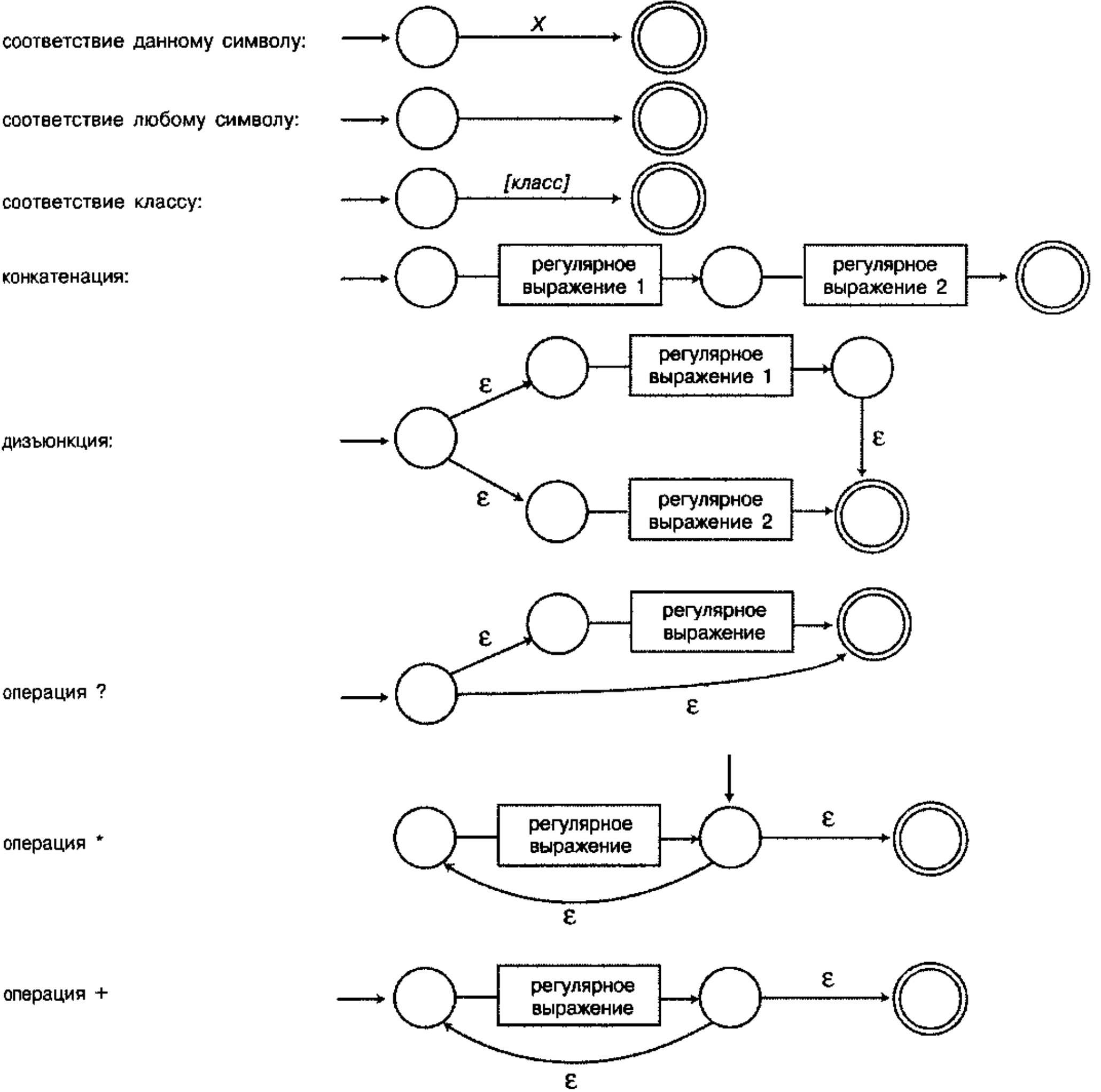


Рисунок 10.6. Конечные NFA-автоматы выполнения операций в регулярных выражениях

Если вы взглянете на рис. 10.6, то наверняка обратите внимание на ряд интересных свойств. В некоторых конструкциях для создания конечных автоматов определены и используются дополнительные состояния, но это делается вполне определенным образом: для каждого состояния существует один или два перехода из него, причем оба являются бесплатными. Это обусловлено веской причиной – в результате кодирование существенно упрощается.

Рассмотрим простой пример: регулярное выражение $(a|b)^*bc$ (повторенный ноль или более раз символ a или b , за которым следуют символы b и c). Используя описанные конструкции, можно шаг за шагом построить конечный NFA-автомат для этого регулярного выражения. Последовательность действий показана на рис. 10.7. Обратите внимание, что на каждом шаге мы получаем конечный NFA-автомат с одним начальным и одним конечным состоянием, причем из каждого нового создаваемого состояния возможно не более двух переходов.

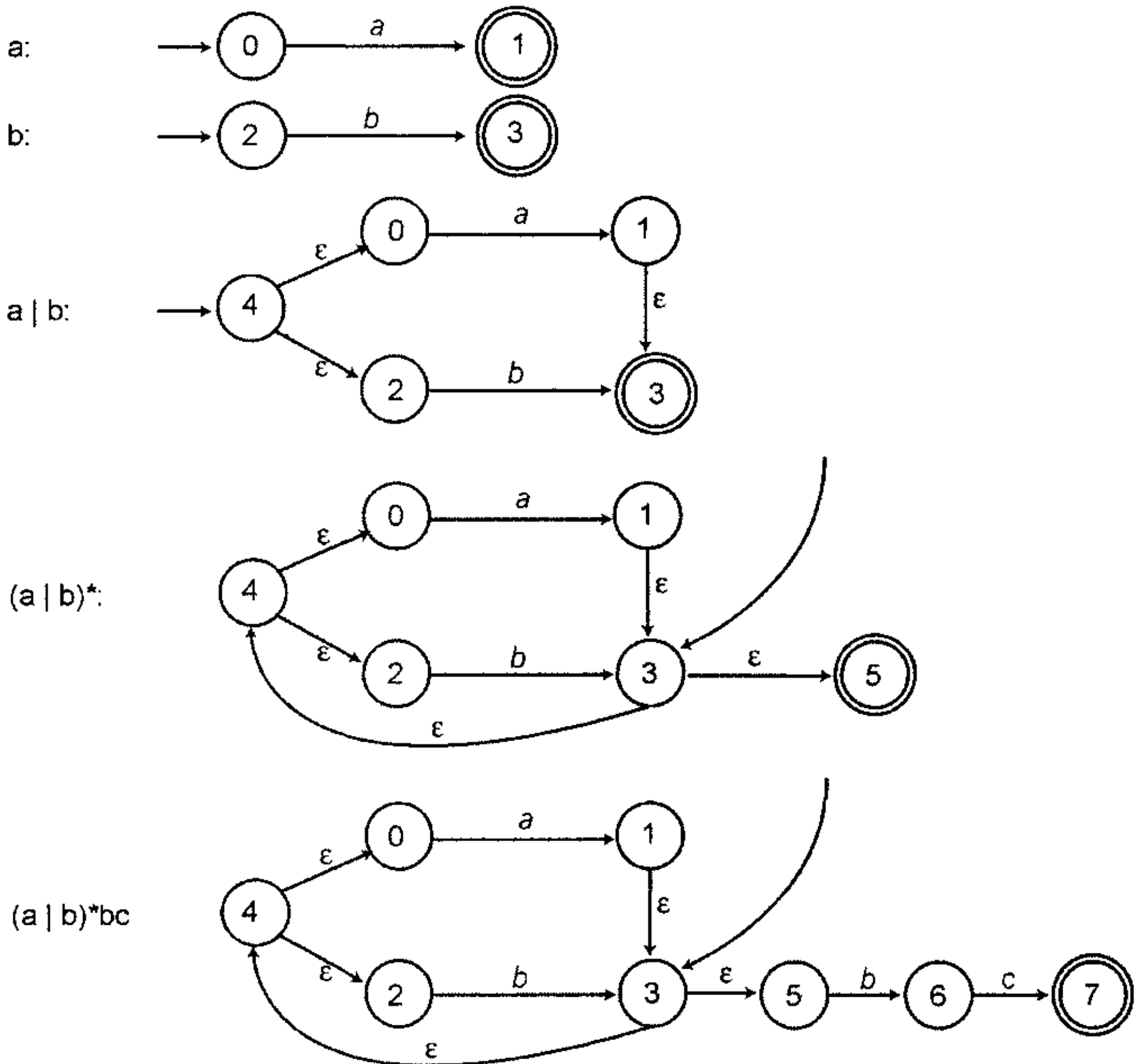


Рисунок 10.7. Пошаговое построение конечного NFA-автомата

Благодаря используемому методу конструирования, можно создать очень простое табличное представление каждого состояния. Каждое состояние будет представлено записью в массиве таких записей (номер состояния является индексом записи в массиве). Запись каждого состояния будет состоять из чего-либо для сравнения и двух номеров состояний для следующего состояния (**NextState1**, **NextState2**). "Что-либо для сравнения" – это шаблон символов, с которым нужно устанавливать соответствие. Им может быть ветвь `ε`, реальный символ, символ операции `"."`, означающий соответствие с любым символом, класс символов (т.е. набор символов, один из которых должен совпадать с входным символом) или класс символов с отрицанием (входной символ не может быть частью набора, с которым устанавливается соответствие). Будучи построенным, этот массив известен под названием *таблицы переходов* (*transition table*). В ней представлены все переходы из одного состояния в другое.

Используя заключительную блок-схему NFA-автомата, показанную на рис. 10.7, можно вручную построить таблицу переходов для регулярного выражения `"(a|b)*bc"`. Результат приведен в таблице 10.1. Мы начинаем с состояния 0 и осуществляем переходы, выполняя сравнение с каждым символом во входной строке, пока не достигнем состояния 7. Реализация алгоритма установки соответствия, использующего подобную таблицу переходов, должна быть очень простой.

Таблица 10.1. Таблица переходов для выражения `(a|b)*bc`

Состояние	0	1	2	3	4	5	6	7
Сравниваемый символ		a		b		b	c	
Следующее состояние 1	1	3	3	5	0	6	7	-1
Следующее состояние 2	-1	-1	-1	4	2	-1	-1	-1

Теперь, когда мы научились графически представлять NFA-автомат для конкретного регулярного выражения и узнали, что этот конечный NFA-автомат может быть представлен простой таблицей переходов, необходимо объединить оба алгоритма в анализаторе регулярных выражений, чтобы он мог выполнять непосредственную компиляцию таблицы состояний. После этого можно будет приступить к рассмотрению заключительной задачи – сопоставлению строк за счет использования таблицы переходов.

Прежде всего, необходимо выбрать способ представления таблицы состояний. Наиболее очевидный выбор – использование класса **TtdRecordList**, описанного в главе 2. Этот класс позволяет при необходимости увеличивать размер внутреннего массива. При этом заранее не нужно определять, сколько состояний может существовать для данного регулярного выражения.

В качестве подсказки будем использовать отдельные конструктивные блоки, показанные на рис. 10.6. Простейшим является выражение, которое распознает отдельный символ. Как видно из первой части рисунка 10.6, нам требуется начальное состояние, в котором будет выполняться распознавание символа, и которое

будет иметь единственную связь с конечным состоянием (каждый из этих элементов будет также требоваться). Создадим простую подпрограмму, которая будет создавать новое состояние (как запись) и дописывать его в таблицу переходов. Код реализации этого простого метода приведен в листинге 10.7. Как видите, он принимает тип соответствия, символ, указатель на класс символов и две связи с другими состояниями. Конечно, не все из этих параметров будут требоваться для каждого создаваемого состояния. Но проще использовать один метод, который может создавать любой тип записи состояния, нежели целый набор таких методов, по одному для каждого возможного типа состояния.

Листинг 10.7. Добавление нового состояния в таблицу состояний

```
function TtdRegexEngine.rcAddState(aMatchType : TtdNFAMatchType;
                                   aChar : AnsiChar;
                                   aCharClass : PtdCharSet;
                                   aNextState1: integer;
                                   aNextState2: integer) : integer;

var
  StateData : TNFAState;
begin
  {определить поля в записи состояния}
  if (aNextState1 = NewFinalState) then
    StateData.sdNextState1 := succ(FTable.Count)
  else
    StateData.sdNextState1 := aNextState1;
  StateData.sdNextState2 := aNextState2;
  StateData.sdMatchType := aMatchType;
  if (aMatchType = mtChar) then
    StateData.sdChar := aChar
  else if (aMatchType = mtClass) or (aMatchType = mtNegClass) then
    StateData.sdClass := aCharClass;
  {добавить новое состояние}
  Result := FTable.Count;
  FTable.Add(@StateData);
end;
```

При взгляде на первую часть рисунка 10.6 кажется, что для этой простой подпрограммы распознавания символа нужно создать два новых состояния. В действительности же можно ограничиться созданием только одного – начального состояния – и принять, что конечным состоянием будет следующее состояние, которое требуется добавить в список. Будем считать его "виртуальным" конечным состоянием. Если бы этот подход удалось применить в каждой из подпрограмм синтаксического анализа, можно было бы избавиться от необходимости создания конечного состояния, эквивалентного начальному состоянию другого подвыражения. Поэтому с этого момента будем считать, что все подпрограммы синтаксического анализа будут возвращать свое начальное состояние, и что конечное состоя-

ние, если оно действительно существует, будет номером индекса следующего состояния, которое необходимо добавить в таблицу переходов.

Из листинга 10.7 видно, что в действительности при передаче номера специального состояния **NewFinalState** в качестве номера следующего состояния мы определяем ссылку на индекс следующего элемента, который должен быть добавлен в таблицу переходов. Конечно, этот элемент еще не существует, но мы предполагаем, что он будет существовать, или что произойдет что-либо еще, позволяющее определить новую ссылку.

Код реализации метода распознавания отдельного символа приведен в листинге 10.8. Снова обратившись к листингу 10.5, обратите внимание на то, как был изменен первоначальный метод синтаксического анализа символа. Во-первых, мы больше не генерируем никаких исключений или сообщений об ошибках. Вместо этого мы возвращаем номер специального состояния **ErrorState**. Мы также отслеживаем код ошибки для каждой происходящей ошибки. Если какие-либо ошибки отсутствуют, новое состояние добавляется в таблицу переходов и возвращается как результат выполнения функции. Естественно, это состояние является начальным состоянием данного выражения. В действительности эта подпрограмма – метод класса машины обработки регулярных выражений.

Листинг 10.8. Синтаксический анализ отдельного символа и добавление его состояния

```
function TtdRegexEngine.rcParseChar : integer;
var
  Ch : AnsiChar;
begin
  {если встречается конец строки, это является ошибкой}
  if (FPosn^ = #0) then begin
    Result := ErrorState;
    FErrorCode := recSuddenEnd;
    Exit;
  end;
  {если текущий символ – один из метасимволов, это ошибка}
  if FPosn^ in MetaCharacters then begin
    Result := ErrorState;
    FErrorCode := recMetaChar;
    Exit;
  end;
  {в противном случае состояние, соответствующее символу, добавляется в
  таблицу состояний}
  {..если он является отмененным символом: вместо него нужно извлечь следующий
  символ}
  if (FPosn^ = '\') then
    inc(FPosn);
  Ch := FPosn^;
  Result := rcAddState(mtChar, Ch, nil, NewFinalState, UnusedState);
  inc(FPosn);
end;
```

Это было достаточно просто, поэтому давайте рассмотрим другой, более сложный метод, который выполняет синтаксический анализ элемента. Первый случай – выражение заключенное в круглые скобки, – во многом подобен рассмотренному ранее: для него не нужно добавлять никакие новые состояния. Второй случай – класс символов или класс символов с отрицанием – определенно нуждается в новом конечном автомате. Синтаксический анализ класса символов выполняется так же, как ранее (при этом он обрабатывается как набор диапазонов, каждый из которых может быть отдельным символом или двумя символами, разделенными дефисом). Однако на этот раз нужно записывать символы в класс. Для этого мы используем набор символов, распределенный в куче. Последним шагом является добавление в таблицу переходов нового состояния, которое распознает данный класс, подобно тому, как это было сделано для подпрограммы распознавания символов. Для заключительного случая, кроме уже рассмотренного конечного автомата для распознавания отдельного символа требуется конечный автомат для обработки символа операции "любой символ", т.е. точки ("."). Реализация этого конечного автомата достаточно проста: необходимо создать новое состояние, которое соответствует любому символу. Полный листинг подпрограммы синтаксического анализа элемента приведен в листинге 10.9. Как и в предыдущем случае, начальное состояние для этих выражений возвращается в качестве результата функции, а конечное состояние является виртуальным конечным состоянием.

Листинг 10.9. Синтаксический анализ <элемента> и вспомогательных компонентов

```
function TtdRegexEngine.rcParseAtom : integer;
var
  MatchType : TtdNFAMatchType;
  CharClass : PtdCharSet;
begin
  case FPosn^ of
    '(' :
      begin
        { обработка открывающей круглой скобки }
        inc(FPosn);
        { синтаксический анализ всего регулярного выражения, заключенного в
          круглые скобки }
        Result := rcParseExpr;
        if (Result = ErrorState) then
          Exit;
        { если текущий символ не является закрывающей круглой скобкой, имеет
          место ошибка }
        if (FPosn^ <> ')') then begin
          FErrorCode := recNoCloseParen;
          Result := ErrorState;
          Exit;
        end;
        { обработка закрывающей круглой скобки }
        inc(FPosn);
      end;
```



```

'[' :
  begin
    {обработка открывающей квадратной скобки}
    inc(FPosn);
    {если первый символ класса - '^', то класс является классом с
    отрицанием, в противном случае это обычный класс}
    if (FPosn^ = '^') then begin
      inc(FPosn);
      MatchType := mtNegClass;
    end
    else begin
      MatchType := mtClass;
    end;
    {выделить набор символов класса и выполнить синтаксический анализ
    класса символов; в результате возврат будет выполнен либо в случае
    ошибки, либо при обнаружении закрывающей квадратной скобки}
    New(CharClass);
    CharClass^ := [];
    if not rcParseCharClass(CharClass) then begin
      Dispose(CharClass);
      Result := ErrorState;
      Exit;
    end;
    {обработка закрывающей квадратной скобки}
    inc(FPosn);
    {добавить новое состояние для класса символов}
    Result := rcAddState(MatchType, #0, CharClass,
      NewFinalState, UnusedState);
  end;
'.' :
  begin
    {обработка метасимвола точки}
    inc(FPosn);
    {добавить новое состояние для лексемы 'любой символ'}
    Result := rcAddState(mtAnyChar, #0, nil,
      NewFinalState, UnusedState);
  end;
else
  {в противном случае - выполнить синтаксический анализ отдельного символа}
  Result := rcParseChar;
end; {case}
end;

```

До сих пор мы создавали состояния без каких-либо ссылок состояний друг на друга. Но если вы обратитесь к блок-схеме конечного NFA-автомата для операции "|", то увидите, что, в конце концов, некоторые состояния приходится объединять друг с другом. Необходимо сохранить начальные состояния для каждого подвыражения и нужно создать новое начальное состояние, которое будет связано

бесплатными связями с каждым из этих двух состояний. Заключительное состояние первого подвыражения должно быть связано с заключительным состоянием второго подвыражения, которое после этого становится конечным состоянием выражения дизъюнкции.

Однако это сопряжено с небольшой проблемой. Заключительное состояние для первого выражения *не существует*. Поэтому его нужно создать, но это следует сделать осторожно, чтобы остальные состояния не стали ошибочно указывать на него.

Естественно, прежде всего, необходимо выполнить синтаксический анализ исходного <члена>. Мы получим начальное состояние (поэтому сохраним его в переменной). При этом известно, что конечное состояние является виртуальным конечным состоянием, следующим непосредственно за концом списка. Если следующим символом является "|", это свидетельствует о выполнении синтаксического анализа за дизъюнктивной конструкцией и о необходимости синтаксического анализа следующего <выражения>. Именно здесь нужно проявить повышенную осторожность. Перво-наперво, мы создаем состояние для конечного состояния этого исходного <члена>. В данный момент, нас не волнует, на какие состояния указывают его связи. Вскоре они будут исправлены. Создание этого конечного состояния означает также, что любые состояния в <члене>, указывающие на виртуальное конечное состояние, фактически будут указывать на состояние, которое мы только что сделали реальным. Теперь нужно создать начальное состояние дизъюнкции. Нам известна одна из связей (исходный <член>), но еще не известна вторая. В конце концов, синтаксический анализ второго <выражения> еще не был выполнен. Теперь мы можем его выполнить. Мы получим начальное состояние, которое используем для исправления второй связи в начальном состоянии дизъюнкции. Новое виртуальное конечное состояние может быть использовано для создания связи, исходящей из конечного состояния исходного <члена>.

В результате выполнения всех этих манипуляций нам пришлось создать два новых состояния (первое является начальным состоянием для дизъюнкции, а второе – конечным состоянием исходного <члена>). При этом мы проявили достаточную осмотрительность, чтобы виртуальное конечное состояние второго <выражения> было виртуальным конечным состоянием всей операции дизъюнкции. Код реализации этого конечного автомата приведен в листинге 10.10 (обратите внимание, что был создан еще один метод, который определяет связи для состояния после его создания).

Листинг 10.10. Синтаксический анализ операции "|"

```
function TtdRegexEngine.rcSetState(aState : integer;  
                                aNextStatel: integer;  
                                aNextState2: integer) : integer;  
var  
    StateData : PNFASState;
```

```

begin
  {извлечь запись состояния и изменить информацию о переходе}
  StateData := PNFAState(FTable[aState]);
  StateData^.sdNextState1 := aNextState1;
  StateData^.sdNextState2 := aNextState2;
  Result := aState;
end;
function TtdRegexEngine.rcParseExpr : integer;
var
  StartStatel : integer; StartState2 : integer;
  EndStatel : integer; OverallStartState : integer;
begin
  {предположим, что имеет место наилучший случай}
  Result := ErrorState;
  {выполнить синтаксический анализ исходного члена}
  StartStatel := rcParseTerm;
  if (StartStatel = ErrorState) then
    Exit;
  {если текущий символ является *не* символом вертикальной черты, дизъюнкция
  отсутствует, поэтому начальное состояние исходного члена необходимо
  вернуть в качестве текущего начального состояния}
  if (FPosn^ <> '|') then
    Result := StartStatel
  {в противном случае необходимо выполнить синтаксический анализ второго
  выражения и объединить их в таблице переходов}
  else begin
    {обработать символ вертикальной черты}
    inc(FPosn);
    {конечное состояние исходного члена еще не существует (хотя член и содержит
    состояние, которое указывает на него), поэтому его нужно создать}
    EndStatel := rcAddState(mtNone, #0, nil, UnusedState, UnusedState);
    {для конструкции ИЛИ требуется новое начальное состояние: оно будет
    указывать на исходный член и на второе выражение, синтаксический анализ
    которого будет выполняться следующим}
    OverallStartState := rcAddState(mtNone, #0, nil,
                                   UnusedState, UnusedState);
    {выполнить синтаксический анализ следующего выражения}
    StartState2 := rcParseExpr;
    if (StartState2 = ErrorState) then
      Exit;
    {изменить состояние, определенное для всего выражения, чтобы вторая связь
    указывала на начало второго выражения}
    Result := rcSetState(OverallStartState, StartStatel, StartState2);
    {определить конечное состояние исходного члена, чтобы оно указывало на
    результирующее конечное состояние, определенное для второго выражения и
    всего выражения в целом}
    rcSetState(EndStatel, FTable.Count, UnusedState);
  end;
end;
end;

```

После ознакомления с этой конкретной конструкцией создание конечных автоматов для операций замыкания ("*", "+" и "?") сложности не представляет. Важно только создавать состояния в правильном порядке. Рассмотрим код, приведенный в листинге 10.11.

Листинг 10.11. Синтаксический анализ операций замыкания

```
function TtdRegexEngine.rcParseFactor : integer;
var
  StartStateAtom : integer;
  EndStateAtom : integer;
begin
  {предположим худшее}
  Result := ErrorState;
  {вначале выполнить синтаксический анализ элемента}
  StartStateAtom := rcParseAtom;
  if (StartStateAtom = ErrorState) then
    Exit;
  {проверить на наличие операции замыкания}
  case FPosn^ of
    '?' : begin
      {обработать символ операции ?}
      inc(FPosn);
      {конечное состояние элемента еще не существует, поэтому его нужно
      создать}
      EndStateAtom := rcAddState(mtNone, #0, nil,
                                UnusedState, UnusedState);
      {создать новое начальное состояние для всего регулярного выражения}
      Result := rcAddState(mtNone, #0, nil,
                          StartStateAtom, EndStateAtom);
      {обеспечить, чтобы новое конечное состояние указывало на следующее
      еще не использованное состояние}
      rcSetState(EndStateAtom, FTable.Count, UnusedState);
    end;
    '*' : begin
      {обработать символ операции *}
      inc(FPosn);
      {конечное состояние элемента еще не существует, поэтому его нужно
      создать; оно будет начальным состоянием всего подвыражения регуляр-
      ного выражения}
      Result := rcAddState(mtNone, #0, nil,
                          NewFinalState, StartStateAtom);
    end;
    '+' : begin
      {обработать символ операции +}
      inc(FPosn);
      {конечное состояние элемента еще не существует, поэтому его нужно
      создать}
      rcAddState(mtNone, #0, nil, NewFinalState, StartStateAtom);
    end;
  end;
```

```
        {начальное состояние всего подвыражения регулярного выражения будет  
        начальным состоянием элемента}  
        Result := StartStateAtom;  
    end;  
else  
    Result := StartStateAtom;  
end; {case}  
end;
```

При выполнении ноля или одного замыкания (операции "?") нужно создать конечное состояние элементарного выражения, к которому применяется операция, и начальное состояние всего конечного автомата. Эти новые состояния связаны между собой, как показано на рис. 10.5.

При выполнении ноля или более замыканий (операции "*") задача еще проще: нужно создать только конечное состояние для элемента. Оно становится начальным состоянием всего выражения. При этом виртуальное конечное состояние является конечным состоянием выражения.

При выполнении одного или более замыканий (операции "+") задача почти столь же проста. Потребуется создать конечное состояние для элемента и связать его с начальным состоянием элемента (которое является также начальным состоянием выражения). При этом виртуальное конечное состояние снова является конечным состоянием выражения.

Теперь осталось написать код только для выполнения операции конкатенации. На рисунке 10.6 эта операция выглядит просто: конечное состояние первого подвыражения становится начальным состоянием второго, и эти подвыражения связаны одно с другим. На практике не все так просто. Конечное состояние первого выражения является виртуальным конечным состоянием, причем не существует никакой гарантии, что оно будет совпадать с начальным состоянием следующего выражения (в этом случае они были бы автоматически связаны). Нет, вместо этого необходимо создать конечное состояние первого выражения и связать его с начальным состоянием второго выражения. Код решения этой последней задачи, включая создание заключительного конечного состояния, приведен в листинге 10.12.

На данный момент мы успешно связали аспекты синтаксического анализа и компиляции, что позволяет принять регулярное выражение и выполнить его синтаксический анализ с целью генерации скомпилированной таблицы переходов. На этапе компиляции программа определит и сохранит начальное состояние полного конечного NFA-автомата для регулярного выражения.

Однако прежде чем приступить к компиляции, необходимо выполнить несколько дополнительных действий для некоторого повышения эффективности. В ряде случаев нам приходилось добавлять некоторые состояния, выход из которых был связан всего с одним бесплатным переходом, причем самым неприятным был случай, когда дополнительное состояние требовалось для выполнения конкатенации.

Листинг 10.12. Синтаксический анализ конкатенации

```

function TtdRegexEngine.rcParseTerm : integer;
var
    StartState2 : integer;
    EndState1 : integer;
begin
    {выполнить синтаксический анализ исходного коэффициента; возвращенный при
     этом номер состояния буде также номером возвращаемого состояния}
    Result := rcParseFactor;
    if (Result = ErrorState) then
        Exit;
    if (FPosn^ = '(') or (FPosn^ = '[') or (FPosn^ = '.') or
        ((FPosn^ <> #0) and not (FPosn^ in MetaCharacters)) then begin
        {конечное состояние исходного коэффициента еще не существует (хотя члены и
         содержит состояние, которое указывает на него), поэтому его нужно создать}
        EndState1 := rcAddState(mtNone, #0, nil, UnusedState, UnusedState);
        {выполнить синтаксический анализ следующего члена}
        StartState2 := rcParseTerm;
        if (StartState2 = ErrorState) then begin
            Result := ErrorState;
            Exit;
        end;
        {объединить первый коэффициент со вторым членом}
        rcSetState(EndState1, StartState2, UnusedState);
    end;
end;

```

Естественно, состояния с единственным переходом для выхода приводят к не-рациональной трате времени. Поэтому необходимо выполнить оптимизацию, исключив их из таблицы переходов. Такие состояния называются *фиктивными*.

Однако вместо того, чтобы их удалять, мы просто их пропустим. Соответствующий алгоритм достаточно прост: необходимо выполнить считывание всех состояний. Для каждого состояния необходимо следовать по ссылке, указанной в его поле **NextState1**. Если она устанавливает связь с одним из фиктивных состояний, связь нужно заменить связью **NextState1** фиктивного состояния. Это же потребует-ся выполнить для связи **NextState2** каждого состояния, если она существует. Код выполнения этой итерационной процедуры приведен в листинге 10.13.

Листинг 10.13. Оптимизация фиктивных состояний

```

procedure TtdRegexEngine.rcLevel1Optimize;
var
    i : integer;
    Walker : PNFAState;
begin
    {оптимизация первого уровня удаляет все состояния, которые содержат только
     один бесплатный переход к другому состоянию}

```

```

{циклически обработать все записи состояний, кроме последней}
for i := 0 to (FTable.Count - 2) do begin
  {получить данное состояние}
  with PNFAState(FTable[i])^ do begin
    {выполнить проход по цепочке, указанной первым следующим состоянием, и
    разорвать связи с состояниями, которые являются простыми одиночными
    бесплатными переходами}
    Walker := PNFAState(FTable[sdNextState1]);
    while (Walker^.sdMatchType = mtNone) and
      (Walker^.sdNextState2 = UnusedState) do begin
      sdNextState1 := Walker^.sdNextState1;
      Walker := PNFAState(FTable[sdNextState1]);
    end;
    {выполнить проход по цепочке, указанной вторым следующим состоянием, и
    разорвать связи с состояниями, которые являются простыми одиночными
    бесплатными переходами}
    if (sdNextState2 <> UnusedState) then begin
      Walker := PNFAState(FTable[sdNextState2]);
      while (Walker^.sdMatchType = mtNone) and
        (Walker^.sdNextState2 = UnusedState) do begin
        sdNextState2 := Walker^.sdNextState1;
        Walker := PNFAState(FTable[sdNextState2]);
      end;
    end;
  end;
end;
end;
end;
end;

```

Сопоставление строк с регулярными выражениями

Пора решить заключительную часть задачи использования регулярных выражений – выполнить сопоставление с ними строк. Вместо того чтобы использовать уже рассмотренный алгоритм обратной трассировки, мы применим другой алгоритм. Используя входную строку, мы выполним обход конечного NFA-автомата (т.е. таблицы переходов), при этом *одновременно* отслеживая все возможные пути через конечный автомат. Со временем символы в строке будут исчерпаны, причем к этой точке будет вести один или более путей, либо возможных путей обработки строки больше не останется.

Однако для реализации этого алгоритма потребуется реализация *очереди с двусторонним доступом (deque)*. Очередь с двусторонним доступом – это двусторонняя очередь, в которой постановку в очередь и исключение из очереди можно выполнять с любого конца. Нам потребуется возможность постановки элементов в конец очереди и их заталкивания в начало и из начала очереди (иначе говоря, исключение элементов из очереди должно выполняться только из ее начала и никогда из ее конца). Элементы, которые нужно будет ставить в очередь, представляют собой целочисленные значения (фактически, номера состояний). Код реализации

этой простой очереди с двусторонним доступом показан в листинге 10.14 (его также можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDInt.Deq.pas**).

Листинг 10.14. Класс очереди целочисленных значений с двусторонним доступом

```

type
  TtdIntDeque = class
    private
      FList : TList;
      FHead : integer;
      FTail : integer;
    protected
      procedure idGrow;
      procedure idError(aErrorCode : integer;
        const aMethodName : TtdNameString);
    public
      constructor Create(aCapacity : integer);
      destructor Destroy; override;
      function IsEmpty : boolean;
      procedure Enqueue(aValue : integer);
      procedure Push(aValue : integer);
      function Pop : integer;
    end;
  constructor TtdIntDeque.Create(aCapacity : integer);
begin
  inherited Create;
  FList := TList.Create;
  FList.Count := aCapacity;
  { для облегчения задачи пользователя очереди с двусторонним доступом
  поместить указатели начала и конца очереди в ее середину – вероятно, это
  более эффективно}
  FHead := aCapacity div 2;
  FTail := FHead;
end;
destructor TtdIntDeque.Destroy;
begin
  FList.Free;
  inherited Destroy;
end
procedure TtdIntDeque.Enqueue(aValue : integer);
begin
  FList.List^[FTail] := pointer(aValue);
  inc(FTail);
  if (FTail = FList.Count) then
    FTail := 0;
  if (FTail = FHead) then
    idGrow;
end;

```

```
procedure TtdIntDeque.idGrow;
var
  OldCount : integer;
  i, j : integer;
begin
  {увеличить размер списка на 50%}
  OldCount := FList.Count;
  FList.Count := (OldCount * 3) div 2;
  {распределить данные по увеличенной области, поддерживая при этом очередь с
   двусторонним доступом}
  if (FHead = 0) then
    FTail := OldCount
  else begin
    j := FList.Count;
    for i := pred(OldCount) downto FHead do begin
      dec(j);
      FList.List^[j] := FList.List^[i]
    end;
    FHead := j;
  end;
end;

function TtdIntDeque.IsEmpty : boolean;
begin
  Result := FHead = FTail;
end;

procedure TtdIntDeque.Push(aValue : integer);
begin
  if (FHead = 0) then
    FHead := FList.Count;
  dec(FHead);
  FList.List^[FHead] := pointer(aValue);
  if (FTail = FHead) then
    idGrow;
end;

function TtdIntDeque.Pop : integer;
begin
  if FHead = FTail then
    idError(tdeDequeIsEmpty, 'Pop');
  Result := integer(FList.List^[FHead]);
  inc(FHead);
  if (FHead = FList.Count) then
    FHead := 0;
end;
```

Алгоритм работает следующим образом. Поставим значение **-1** в очередь с двусторонним доступом. Это специальное значение, которое указывает о необходимости выполнить считывание входной строки по одному элементу. Теперь поставим в очередь с двусторонним доступом номер исходного состояния. Установим целочисленное значение равным **0**. Это значение будет индексом текущего символа в строке, сопоставление с которой выполняется.

После того, как подготовка закончена, мы входим в цикл. На каждом этапе выполнения цикла выполняется одно и то же действие: выталкивание верхнего значения из очереди. Если этим значением является **-1** (как, естественно, это будет вначале), мы увеличиваем индекс текущего символа и извлекаем этот символ из сопоставляемой строки. Снова поставим значение **-1** в очередь, чтобы знать, когда нужно выполнить считывание следующего символа. Если это значение не **-1**, оно должно быть реальным номером состояния. Взглянем на запись состояния в таблице переходов. Если текущий входной символ соответствует шаблону символов этого состояния, значение **NextState1** состояния нужно поставить в очередь. Понятно, что если шаблоном символов состояния был **ε**, символ не соответствовал шаблону. В этом случае в очередь с двусторонним доступом мы заталкиваем значение **NextState1**, а затем значение **NextState2**.

Выполнение цикла прекращается, как только очередь с двусторонним доступом оказывается пустой (ни один путь не соответствует входной строке) или при считывании всех символов из сопоставляемой строки (в этом случае очередь содержит набор состояний, достигнутых на момент достижения конца строки, которые можно выталкивать из очереди до тех пор, пока в зависимости от конкретной ситуации не будет найдено или не найдено одно единственное конечное состояние).

Общий результат применения этого алгоритма состоит в том, что в очередь с двусторонним доступом помещается значение "извлечь следующий символ" (**-1**). "Слева" от него располагается набор состояний, с которым нам по-прежнему необходимо сравнить текущий символ (мы постоянно выталкиваем из очереди эти состояния и помещаем в нее те, которых можно достичь за счет выполнения бесплатного перехода). "Справа" от него находятся состояния, полученные из тех, которые уже соответствуют текущему символу. Переход к ним будет осуществляться сразу после выталкивания значения **-1** из очереди и извлечения следующего символа. Как видите, алгоритм одновременно проверяет все пути обхода конечного NFA-автомата.

Подпрограмма сопоставления приведена в листинге 10.15. Она была создана в качестве метода машины обработки регулярных выражений. Ей передается строка, с которой должно быть выполнено сопоставление, и значение индекса. Значение индекса указывает позицию в строке, начиная с которой предположительно должно начинаться совпадение. Это позволяет использовать регулярное выражение для сопоставления с любой частью строки, а не со всей строкой, как делалось в приведенных простых примерах конечных автоматов. Метод будет возвращать значение **true**, если таблица переходов регулярного выражения соответствует строке, начиная с данной позиции.

Листинг 10.15. Сопоставление подстрок с таблицей переходов

```

function TtdRegexEngine.rcMatchSubString(const S : string;
                                         StartPosn : integer) : boolean;

var
  Ch : AnsiChar;
  State : integer;
  Deque : TtdIntDeque;
  StrInx : integer;
begin
  {предположить, что сопоставление будет неудачным}
  Result := false;
  {создать очередь с двусторонним доступом}
  Deque := TtdIntDeque.Create(64);
  try
    {поставить в очередь специальное значение, означающее начало сканирования}
    Deque.Enqueue(MustScan);
    {поставить в очередь первое состояние}
    Deque.Enqueue(FStartState);
    {подготовить индекс строки}
    StrInx := StartPosn - 1;
    {выполнять цикл до тех пор, пока очередь не будет пуста, или пока строка не закончится}
    while (StrInx <= length(S)) and not Deque.IsEmpty do begin
      {вытолкнуть верхнее состояние из очереди}
      State := Deque.Pop;
      {вначале выполнить обработку состояния "необходимо выполнить сканирование"}
      if (State = MustScan) then begin
        {если очередь пуста, вполне вероятно, что задача выполнена, поскольку не осталось никаких состояний для обработки новых символов}
        if not Deque.IsEmpty then begin
          {если строка не закончилась, нужно извлечь символ и снова поставить в очередь состояние "необходимо выполнить сканирование"}
          inc(StrInx);
          if (StrInx <= length(S)) then begin
            Ch := S[StrInx];
            Deque.Enqueue(MustScan);
          end;
        end;
      end;
      {в противном случае необходимо обработать состояние}
      else with PNFAState(FTable[State])^ do begin
        case sdMatchType of
          mtNone :
            begin
              {для бесплатных переходов необходимо заталкивать в очередь следующие состояния}
              Deque.Push(sdNextState2);
              Deque.Push(sdNextState1);
            end;
        end;
      end;
    end;
  except
    Result := false;
  end;
end;

```

```

mtAnyChar :
  begin
    { для сопоставления с любым символом необходимо поставить в очередь
      следующее состояние }
    Deque.Enqueue(sdNextState1);
  end;
mtChar :
  begin
    { для сопоставления с символом необходимо поставить в очередь
      следующее состояние }
    if (Ch = sdChar) then
      Deque.Enqueue(sdNextState1);
    end;
mtClass :
  begin
    { для сопоставления с символом, входящим в состав класса,
      необходимо поставить в очередь следующее состояние }
    if (Ch in sdClass^) then
      Deque.Enqueue(sdNextState1);
    end;
mtNegClass :
  begin
    { для сопоставления с символом, не входящим в состав класса,
      необходимо поставить в очередь следующее состояние }
    if not (Ch in sdClass^) then
      Deque.Enqueue(sdNextState1);
    end;
mtTerminal :
  begin
    { в случае достижения конечного состояния строка соответствует
      регулярному выражению, если регулярное выражение не содержало
      никакого символа привязки или достигнут конец строки }
    if (not FAnchorEnd) or (StrInx > length(S)) then begin
      Result := true;
      Exit;
    end;
  end;
end;
end;
end;

```

{ достижение этой точки свидетельствует либо о том, что очередь исчерпана, либо о достижении конца строки. В первом случае подстрока не соответствует регулярному выражению, поскольку отсутствуют состояния для сопоставления. Во втором случае необходимо проверить состояния, расположенные слева от очереди, чтобы проверить, не является ли одно из них конечным. Если это так, строка соответствует регулярному выражению, определенному таблицей переходов }


```

while not Deque.IsEmpty do begin
  State := Deque.Pop;
  with PNFAState(FTable[State])^ do begin
    case sdMatchType of
      mtNone :
        begin
          { для бесплатных переходов необходимо заталкивать в очередь
            следующие состояния }
          Deque.Push(sdNextState2);
          Deque.Push(sdNextState1);
        end;
      mtTerminal :
        begin
          { в случае достижения конечного состояния строка соответствует
            регулярному выражению, если регулярное выражение не содержало
            никакого символа привязки или достигнут конец строки }
          if (not FAnchorEnd) or (StrInx > length(S)) then begin
            Result := true;
            Exit;
          end;
        end;
      end; {case}
    end;
  end;
finally
  Deque.Free;
end;
end;

```

Было бы желательно, чтобы подпрограмму сопоставления можно было бы применять не только к любой начальной позиции строки, но, при необходимости, и только ко всей строке.

Поэтому представим два новых символа операций регулярных выражений: символы операций привязки "^" и "\$". Знак вставки "^" означает, что любое соответствие должно иметь место только с начала строки. Знак доллара "\$" означает, что совпадение должно происходить на всем пути до самого конца строки. Так, например, регулярное выражение "^function" означает "совпадение со словом function с начала строки", а "^end.\$" означает, что вся строка должна состоять из символов e, n, d и точки. Она не должна содержать никаких других символов. Символы ^ и \$ могут присутствовать, соответственно, только в начале и конце регулярного выражения. Они не могут находиться ни в какой другой позиции.

Это обуславливает небольшое изменение определенных нами грамматических правил. Изменение не очень велико, но, как мы видели, корректная формулировка грамматических правил существенно упрощает создание кода. Код реализации нового правила и соответствующего метода синтаксического анализа приведен в ли-

стинге 10.16. Естественно, интерфейсный метод **Parse** также изменен, чтобы вызывать именно его, а не первоначальный метод.

Листинг 10.16. Использование операций привязки

```

{<anchorexpr> ::= <expr> |
                    '^' <expr> |
                    <expr> '$' |
                    '^' <expr> '$'}

function TtdRegexEngine.rcParseAnchorExpr : integer;
begin
    {проверить на наличие начального символа '^'}
    if (FPosn^ = '^') then begin
        FAnchorStart := true;
        inc(FPosn);
    end;
    {выполнить синтаксический анализ выражения}
    Result := rcParseExpr;
    {в случае успеха необходимо выполнить проверку на наличие конечного символа '$'}
    if (Result <> ErrorState) then begin
        if (FPosn^ = '$') then begin
            FAnchorEnd := true;
            inc(FPosn);
        end;
    end;
end;

```

Теперь код выполнения сопоставления строк можно изменить для сопоставления как целых строк, так и подстрок. Если регулярное выражение начинается с символа "^", нужно просто попытаться становить соответствие строки, начиная с первого символа. Если нет, необходимо попытаться установить соответствие с каждой из подстрок, образованных из исходной строки. Код метода **MatchString**, в котором принимается это решение, приведен в листинге 10.17.

Листинг 10.17. Метод MatchString

```

function TtdRegexEngine.MatchString(const S : string) : integer;
var
    i : integer;
    ErrorPos : integer;
    ErrorCode : TtdRegexError;
begin
    {если синтаксический анализ строки регулярного выражения еще не был
    выполнен, необходимо его выполнить}
    if (FTable.Count = 0) then begin
        if not Parse(ErrorPos, ErrorCode) then
            rcError(tdeRegexParseError, 'MatchString', ErrorPos);
    end;

```

```
{теперь необходимо выяснить, соответствует ли строка регулярному выражению  
(сопоставление пустых строк не выполняется)}  
Result := 0;  
if (S <> '') then  
  {если указанное регулярное выражение содержит начальный символ привязки,  
   нужно проверить соответствие строки только начиная с первой позиции}  
  if FAnchorStart then begin  
    if rcMatchSubString(S, 1) then  
      Result := 1;  
  end  
  {в противном случае необходимо проверить соответствие строки в каждой из  
   позиций и при первом же успешном сопоставлении выполнить возврат}  
  else begin  
    for i := 1 to length(S) do  
      if rcMatchSubString(S, i) then begin  
        Result := i;  
        Break;  
      end;  
    end;  
  end;  
end;
```

Если вы еще раз внимательно просмотрите листинг 10.15, то увидите, что код сопоставления уже обеспечивает применение конечного символа привязки. Код воспринимает конечное состояние в качестве признака соответствия регулярному выражению, если регулярное выражение не содержало конечного символа привязки, или же в случае достижения конца строки. При невыполнении любого из этих условий, конечное состояние будет игнорироваться..

Полный исходный код класса **TtdRegexEngine** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TdRegex.pas**.

Резюме

В этой главе мы рассмотрели как детерминированные (DFA), так и недетерминированные (NFA) конечные автоматы. При этом мы исследовали несколько простых примеров DFA-автоматов.

Мы установили также, что при кодировании вручную конечные DFA-автоматы проще определить, понять и создать соответствующий код, в то время как конечные NFA-автоматы больше подходят для автоматических процессов. В заключение мы реализовали полную машину обработки регулярных выражений, которая выполняет синтаксический анализ и компиляцию регулярного выражения в конечный NFA-автомат (представленный таблицей переходов). Этот конечный NFA-автомат может использоваться для сопоставления строк.



Сжатие данных

Думая о данных, обычно мы представляем себе ни что иное, как передаваемую этими данными информацию: список клиентов, мелодию на аудио компакт-диске, письмо и тому подобное. Как правило, мы не слишком задумываемся о физическом представлении данных. Заботу об этом – отображении списка клиентов, воспроизведении компакт-диска, печати письма – берет на себя программа, манипулирующая данными.

Представление данных

Рассмотрим двойственность природы данных: с одной стороны, содержимое информации, а с другой – ее физическое представление. В 1950 году Клод Шеннон (Claude Shannon) заложил основы теории информации, в том числе идею о том, что данные могут быть представлены определенным минимальным количеством битов. Эта величина получила название энтропии данных (термин был заимствован из термодинамики). Шеннон установил также, что обычно количество бит в физическом представлении данных превышает значение, определяемое их энтропией.

В качестве простого примера рассмотрим исследование понятия вероятности с помощью монеты. Можно было бы подбросить монету множество раз, построить большую таблицу результатов, а затем выполнить определенный статистический анализ этого большого набора данных с целью формулирования или доказательства какой-то теоремы. Для построения набора данных, результаты подбрасывания монеты можно было бы записывать несколькими различными способами: можно было бы записывать слова "орел" или "решка"; можно было бы записывать буквы "О" или "Р"; или же можно было бы записывать единственный бит (напри-

мер "да" или "нет", в зависимости от того, на какую сторону падает монета). Согласно теории информации, результат каждого подбрасывания монеты можно закодировать единственным битом, поэтому последний приведенный вариант был бы наиболее эффективным с точки зрения объема памяти, необходимого для кодирования результатов. С этой точки зрения первый вариант является наиболее расточительным, поскольку для записи результата единственного подбрасывания монеты требовалось бы четыре или пять символов.

Однако посмотрим на это под другим углом: во всех приведенных примерах записи данных мы сохраняем одни и те же результаты – одну и ту же информацию – используя все меньший и меньший объем памяти. Другими словами, мы выполняем сжатие данных.

Сжатие данных

Сжатие данных (data compression) – это алгоритм эффективного кодирования информации, при котором она занимает меньший объем памяти, нежели ранее. Мы избавляемся от *избыточности (redundancy)*, т.е. удаляем из физического представления данных те биты, которые в действительности не требуются, оставляя только то количество битов, которое необходимо для представления информации в соответствии со значением энтропии. Существует показатель эффективности сжатия данных: *коэффициент сжатия (compression ratio)*. Он вычисляется путем вычитания из единицы частного от деления размера сжатых данных на размер исходных данных и обычно выражается в процентах. Например, если размер сжатых данных равен 1000 бит, а несжатых – 4000 бит, коэффициент сжатия составит 75%, т.е. мы избавились от трех четвертей исходного количества битов.

Конечно, сжатые данные могут быть записаны в форме недоступной для непосредственного считывания и понимания человеком. Люди нуждаются в определенной избыточности представления данных, способствующей их эффективному распознаванию и пониманию. Применительно к эксперименту с подбрасыванием монеты последовательности символов "О" и "Р" обладают большей наглядностью, чем 8-битовые значения байтов. (Возможно, что для большей наглядности пришлось бы разбить последовательности символов "О" и "Р" на группы, скажем, по 10 символов в каждой.) Иначе говоря, возможность выполнения сжатия данных бесполезна, если отсутствует возможность их последующего восстановления. Эту обратную операцию называют *декодированием (decoding)*.

Типы сжатия

Существует два основных типа сжатия данных: *с потерями (lossy)* и *без потерь (lossless)*. Сжатие без потерь проще для понимания. Это метод сжатия данных, когда при восстановлении данных возвращается точная копия исходных данных. Такой тип сжатия используется программой **PKZIP™**: распаковка упакованного файла приводит к созданию файла, который имеет в точности то же

содержимое, что и оригинал перед его сжатием. И напротив, сжатие с потерями не позволяет при восстановлении получить те же исходные данные. Это кажется недостатком, но для определенных типов данных, таких как данные изображений и звука, различие между восстановленными и исходными данными не имеет особого значения: наши зрение и слух не в состоянии уловить образовавшиеся различия. В общем случае алгоритмы сжатия с потерями обеспечивают более эффективное сжатие, чем алгоритмы сжатия без потерь (в противном случае их не стоило бы использовать вообще). Для примера можно сравнить предназначенный для хранения изображений формат с потерями JPEG с форматом без потерь GIF. Множество форматов потокового аудио и видео, используемых в Internet для загрузки мультимедиа-материалов, являются алгоритмами сжатия с потерями.

В случае экспериментов с подбрасыванием монеты было очень легко определить наилучший способ хранения набора данных. Но для других данных эта задача становится более сложной. При этом можно применить несколько алгоритмических подходов. Два класса сжатия, которые будут рассмотрены в этой главе, представляют собой алгоритмы сжатия без потерь и называются *кодированием с минимальной избыточностью* (*minimum redundancy coding*) и *сжатием с применением словаря* (*dictionary compression*).

Кодирование с минимальной избыточностью – это метод кодирования байтов (или, более строго, *символов*), при котором чаще встречающиеся байты кодируются меньшим количеством битов, чем те, которые встречаются реже. Например, в тексте на английском языке буквы E, T и A встречаются чаще, нежели буквы Q, X и Z. Поэтому, если бы удалось закодировать буквы E, T и A меньшим количеством битов, чем 8 (как должно быть в соответствии со стандартом ASCII), а буквы Q, X и Z – большим, текст на английском языке удалось бы сохранить с использованием меньшего количества битов, чем при соблюдении стандарта ASCII.

При использовании сжатия с применением словаря данные разбиваются на большие фрагменты (называемые *лексемами*), чем символы. Затем применяется алгоритм кодирования лексем определенным минимальным количеством битов. Например, слова "the", "and" и "to" будут встречаться чаще, чем такие слова, как "electric", "ambiguous" и "irresistible", поэтому их нужно закодировать меньшим количеством битов, чем требовалось бы при кодировании в соответствии со стандартом ASCII.

Потоки битов

Прежде чем приступить к исследованию реальных алгоритмов сжатия, необходимо кратко рассмотреть задачу манипулирования битами. При использовании большинства алгоритмов сжатия, которые будут рассмотрены, сжатие данных выполняется с использованием переменного количества битов, независимо от того, рассматриваются ли данные в качестве последовательности символов или лексем. Нельзя считать, что байты всегда будут состоять из групп по 8 битов.

Нам потребуется выполнять две базовых операции: считывание отдельного бита и запись отдельного бита. На основе этих операций можно было бы построить операции, выполняющие считывание и запись сразу нескольких битов. Поэтому мы разработаем и создадим *поток битов (bit stream)* – структуру данных, содержащую в себе набор битов. Понятно, что поток битов будет использовать еще одну структуру данных, в которой данные битов хранятся в виде последовательности байтов. Эта структура будет извлекать биты в соответствии с байтами в данных, на основе которых она построена. Поскольку мы используем Delphi, в качестве базовой структуры данных потока битов мы выберем объект **TStream** (или производный от него). В результате, например, мы смогли бы рассматривать поток памяти или поток файла как поток битов. Фактически, поскольку потоки битов будут использоваться только в качестве последовательных групп битов, мы создадим два различных типа: входной поток битов и выходной поток битов. Кроме того, можно избавиться от обычно используемого метода **Seek**, поскольку поиск в потоке битов мы выполнять не будем.

Код интерфейса классов **TtdInputBitStream** и **TtdOutputBitStream** приведен в листинге 11.1.

Листинг 11.1. Интерфейс классов потоков битов

```
type
  TtdInputBitStream = class
    private
      FAccum : byte;
      FBufEnd : integer;
      FBuffer : PAnsiChar;
      FBufPos : integer;
      FMask : byte;
      FName : TtdNameString;
      FStream : TStream;
    protected
      procedure ibsError(aErrorCode : integer;
        const aMethodName : TtdNameString);
      procedure ibsReadBuffer;
    public
      constructor Create(aStream : TStream);
      destructor Destroy; override;
      function ReadBit : boolean;
      procedure ReadBits(var aBitString : TtdBitString;
        aBitCount : integer);
      function ReadByte : byte;
      property Name : TtdNameString read FName write FName;
  end;
```

```

TtdOutputBitStream = class
  private
    FAccum : byte;
    FBuffer : PAnsiChar;
    FBufPos : integer;
    FMask : byte;
    FName : TtdNameString;
    FStream : TStream;
    FStrmBroken : boolean;
  protected
    procedure obsError(aErrorCode : integer;
                     const aMethodName : TtdNameString);
    procedure obsWriteBuffer;
  public
    constructor Create(aStream : TStream);
    destructor Destroy; override;
    procedure WriteBit(aBit : boolean);
    procedure WriteBits(const aBitString : TtdBitString);
    procedure WriteByte(aByte : byte);
    property Name : TtdNameString read FName write FName;
end;

```

Оба конструктора **Create** требуют передачи им в качестве параметра уже созданного производного объекта **TStream**. Из этого потока байтов класс потока битов будет извлекать или сохранять отдельные байты. Код конструкторов **Create** и деструкторов **Destroy** этих классов приведен в листинге 11.2.

Листинг 11.2. Создание и уничтожение объектов потока битов

```

constructor TtdInputBitStream.Create(aStream : TStream);
begin
  inherited Create;
  FStream := aStream;
  GetMem(FBuffer, StreamBufferSize);
end;

destructor TtdInputBitStream.Destroy;
begin
  if (FBuffer <> nil) then
    FreeMem(FBuffer, StreamBufferSize);
  inherited Destroy;
end;

constructor TtdOutputBitStream.Create(aStream : TStream);
begin
  inherited Create;
  FStream := aStream;
  GetMem(FBuffer, StreamBufferSize);
  FMask := 1; {подготовиться к записи первого бита}
end;

```

```
destructor TtdOutputBitStream.Destroy;
begin
  if (FBuffer <> nil) then begin
    {если значение Mask не равно 1, это означает присутствие в аккумуляторной
    переменной каких-то бит, которые требуется записать в буфер. Следует
    убедиться, что буфер записывается в базовый поток}
    if not FStrmBroken then begin
      if (FMask <> 1) then begin
        byte(FBuffer[FBufPos]) := FAccum;
        inc(FBufPos);
      end;
      if (FBufPos > 0) then
        obsWriteBuffer;
    end;
    FreeMem(FBuffer, StreamBufferSize);
  end;
  inherited Destroy;
end;
```

Обратите внимание, что оба конструктора **Create** выделяют большой буфер байтов (размер которого не меньше 4 Кб), чтобы базовый поток был доступен только для блоков данных. Иначе говоря, мы будем осуществлять буферизацию базового потока. Следовательно, метод **Destroy** должен освобождать этот буфер, убедившись, что на момент вывода потока битов любые все еще буферизованные данные записаны в базовый поток.

Обратите внимание на ссылку на своеобразное поле класса **FStrmBroken**. Оно служит средством обхода возможного условия ошибки. Предположим, что базовым потоком был экземпляр **TFileStream**, и что во время использования выходного потока битов имело место переполнение диска. В этом случае требуется запись выходного потока битов, сигнализирующего о подобной проблеме как об исключительной ситуации. Как только это исключение сгенерировано, дальнейшие попытки записи в базовый поток лишены всякого смысла, поэтому код устанавливает значение поля **FStrmBroken** равным **true**, сигнализируя о прерывании потока.

После того, как мы научились создавать и уничтожать потоки битов, следует рассмотреть задачу считывания и записи отдельного бита. Код выполнения считывания отдельного бита показан в листинге 11.3. Метод **ReadBit** возвращает булево значение – **true**, если следующий считанный из потока бит был установлен, и **false** в противном случае.

Мы используем байт маски (**FMask**), содержащий единственный бит установки и выполняем операцию AND (И) для этой маски и текущего байта (**FAccum**) из базового потока. Если результат отличен от нуля, бит в байте был установлен, и мы должны вернуть значение **true**. Если он равен нулю, бит в байте был очищен, и мы возвращаем значение **false**. Затем мы выполняем сдвиг маски влево на один бит, чтобы выдвинуть единственный бит маски на одну позицию. Если в момент

начала процесса маска была нулевой, это означает, что нужно выполнить считывание нового байта из буфера и сбросить маску. Если буфер был пуст или был полностью считан, необходимо выполнить считывание из базового потока с целью заполнения следующего буфера.

Листинг 11.3. Считывание отдельного бита из объекта `TtdInputBitStream`

```
function TtdInputBitStream.ReadBit : boolean;
begin
    {если в текущей аккумуляторной переменной никаких битов не осталось,
     необходимо выполнить считывание следующего байта аккумуляторной переменной
     и сбросить значение маски}
    if (FMask = 0) then begin
        if (FBufPos >= FBufEnd) then
            ibsReadBuffer;
        FAccum := byte(FBuffer[FBufPos]);
        inc(FBufPos);
        FMask := 1;
    end;
    {извлечь следующий бит}
    Result := (FAccum and FMask) <> 0;
    FMask := FMask shl 1;
end;
```

После того, как мы выяснили, как выполняется считывание отдельного бита, покажем, что запись отдельного бита – тот же самый процесс, только выполняемый в обратном порядке. Код метода `WriteBit`, в котором единственный бит передается как булево значение – `true`, если бит установлен, и `false`, если он очищен – приведен в листинге 11.4.

Листинг 11.4. Запись отдельного бита в объект `TtdOutputBitStream`

```
procedure TtdOutputBitStream.WriteBit(aBit : boolean);
begin
    {установить следующий свободный бит}
    if aBit then
        FAccum := (FAccum or FMask);
    FMask := FMask shl 1;
    {при отсутствии свободных битов в текущей аккумуляторной переменной ее
     значение нужно записать в буфер и сбросить значение аккумуляторной
     переменной и маски}
    if (FMask = 0) then begin
        byte(FBuffer[FBufPos]) := FAccum;
        inc(FBufPos);
        if (FBufPos >= StreamBufferSize) then
            obsWriteBuffer;
        FAccum := 0;
        FMask := 1;
    end;
end;
```

Поскольку обработка всегда начинается при значении аккумуляторного байта (**FAccum**) равном нулю, нужно всего лишь записать эти биты установки, а не очистить их. Мы снова используем маску (**FMask**), содержащую единственный бит установки, но на этот раз чтобы установить соответствующий бит, после чего выполняем операцию OR (ИЛИ) между маской и значением аккумуляторной переменной. Затем мы сдвигаем маску влево на один бит, подготавливая к обработке следующий бит. Однако если теперь значение маски равно нулю, потребуется сохранить аккумуляторный байт в буфере (записывая буфер в базовый поток, если буфер полон), а затем сбросить значение аккумуляторного байта и маски.

Полный код обоих классов **TtdInputBitStream** и **TtdOutputBitStream** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDStrms.pas**. Полный код содержит также подпрограммы одновременного считывания и записи нескольких битов – либо восьми битов отдельного байта (**ReadByte** и **WriteByte**), либо переменного числа байтов из массива байтов (**ReadBits** и **WriteBits**). Для доступа к отдельным битам все эти дополнительные подпрограммы используют одну и ту же методологию манипуляции битами. Просто соответствующие операции выполняются в цикле.

Сжатие с минимальной избыточностью

Теперь, когда в нашем распоряжении имеется класс потока битов, им можно воспользоваться при рассмотрении алгоритмов сжатия и восстановления данных. Мы начнем с исследования алгоритмов кодирования с минимальной избыточностью, а затем рассмотрим более сложное сжатие с применением словаря.

Мы приведем подробное описание трех алгоритмов кодирования с минимальной избыточностью: кодирование *Шеннона-Фано* (*Shannon-Fano*), кодирование *Хаффмана* (*Huffman*) и *сжатие с применением скошенного дерева* (*splay tree compression*), однако рассмотрим реализации только последних двух алгоритмов (алгоритм кодирования Хаффмана ни в чем не уступает, а кое в чем даже превосходит алгоритм кодирования Шеннона-Фано). При использовании каждого из этих алгоритмов входные данные анализируются как поток байтов, и различным значениям байтов тем или иным способом присваиваются различные последовательности битов.

Кодирование Шеннона-Фано

Первый алгоритм сжатия, который мы рассмотрим – кодирование Шеннона-Фано, названное так по имени двух исследователей, которые одновременно и независимо друг от друга разработали этот алгоритм: Клода Шеннона (Claude Shannon) и Р. М. Фано (R. M. Fano). Алгоритм анализирует входные данные и на их основе строит бинарное дерево минимального кодирования. Используя это дерево, затем можно выполнить повторное считывание входных данных и закодировать их.

Чтобы проиллюстрировать работу алгоритма, выполним сжатие предложения "How much wood could a woodchuck chuck?" ("Сколько дров мог бы заготовить дровосек?") Прежде всего, предложение необходимо проанализировать. Просмотрим данные и вычислим, сколько раз в предложении встречается каждый символ. Занесем результаты в таблицу (см. таблицу 11.1).

Таблица 11.1. Частота появления символов в примере предложения

Символ	Количество появлений
Пробел	6
c	6
o	6
u	4
d	3
h	3
w	3
k	2
H	1
a	1
l	1
m	1
?	1

Теперь разделим таблицу на две части, чтобы общее число появлений символов в верхней половине таблицы приблизительно равнялось общему числу появлений в нижней половине. Предложение содержит 38 символов, следовательно, верхняя половина таблицы должна отражать приблизительно 19 появлений символов. Это просто: достаточно поместить разделительную линию между строкой o и строкой u. В результате этого верхняя половина таблицы будет отражать появление 18 символов, а нижняя – 20. Таким образом, мы получаем таблицу 11.2.

Таблица 11.2. Начало построения дерева Шеннона-Фано

Символ	Количество появлений
Пробел	6
c	6
o	6
-----разделительная линия 1	
u	4
d	3
h	3
w	3
k	2
H	1

Таблица 11.2. (Продолжение)

Символ	Количество появлений
a	1
l	1
m	1
?	1

Теперь сделаем то же с каждой из частей таблицы: вставим линию между строками так, чтобы разделить каждую из частей. Продолжим этот процесс, пока все буквы не окажутся разделенными одна от другой. Результирующее дерево Шеннона-Фано представлено в таблице 11.3.

Таблица 11.3. Завершенное дерево Шеннона-Фано

Символ	Количество появлений
Пробел	6
-----разделительная линия 2	
c	6
-----разделительная линия 3	
o	6
-----разделительная линия 1	
u	4
-----разделительная линия 3	
d	3
-----разделительная линия 4	
h	3
-----разделительная линия 2	
w	3
-----разделительная линия 4	
k	2
-----разделительная линия 3	
H	1
-----разделительная линия 5	
a	1
-----разделительная линия 4	
l	1
-----разделительная линия 5	
m	1
-----разделительная линия 6	
?	1

Я намеренно изобразил разделительные линии различными по длине, чтобы разделительная линия 1 была самой длинной, разделительная линия 2 немного короче и так далее, вплоть до самой короткой разделительной линии 6. Этот подход обусловлен тем, что разделительные линии образуют повернутое на 90° бинарное дерево (чтобы убедиться в этом, поверните таблицу на 90° против часовой стрелки). Разделительная линия 1 является корневым узлом дерева, разделительные линии 2 – двумя его дочерними узлами и т.д. Символы образуют листья дерева. Результирующее дерево в обычной ориентации показано на рис. 11.1

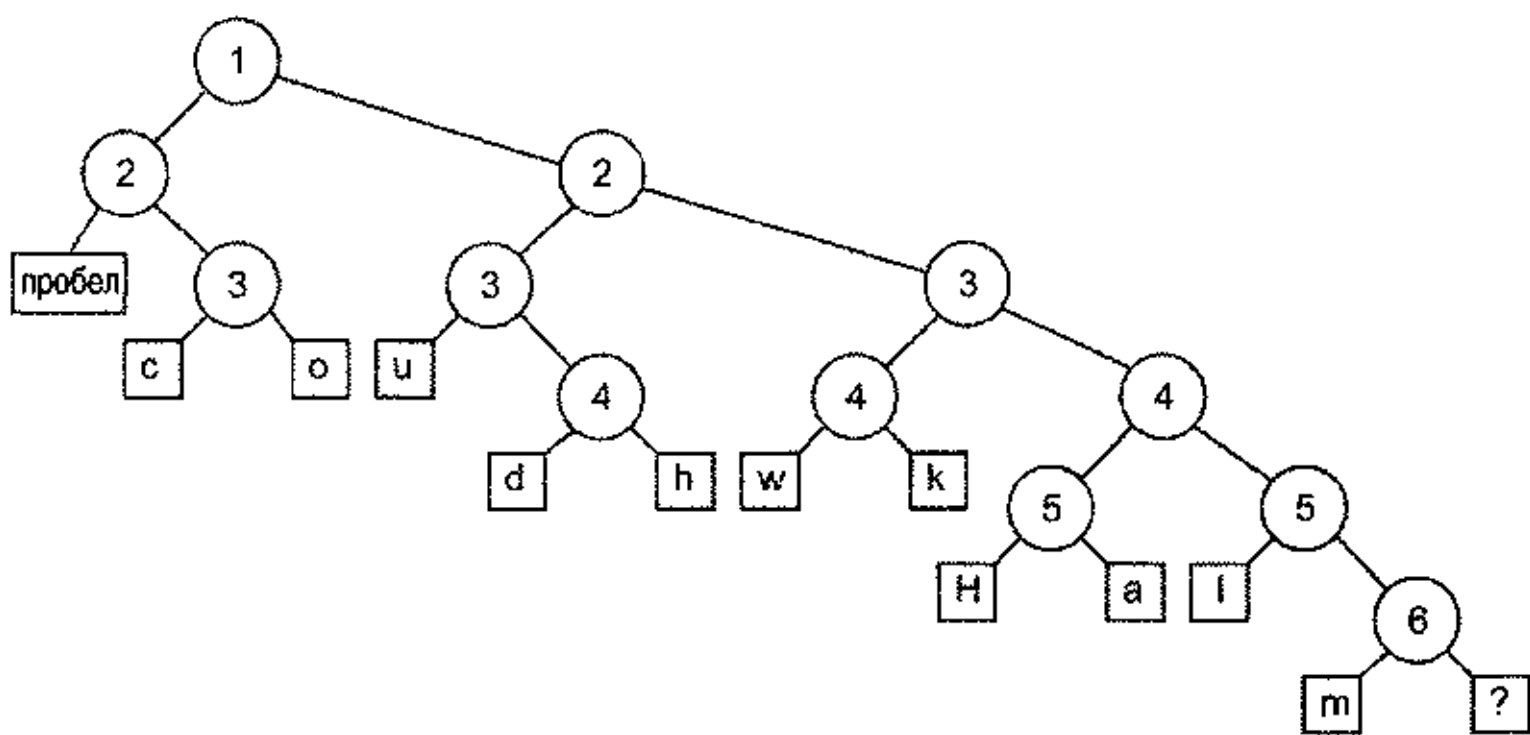


Рисунок 11.1. Дерево Шеннона-Фано

Все это очень хорошо, но как оно помогает решить задачу кодирования каждого символа и выполнения сжатия? Что ж, чтобы добраться до символа пробела, мы начинаем с конечного узла, перемещаемся влево, а затем снова влево. Чтобы добраться до символа **с**, мы смещаемся влево из корневого узла, затем вправо, а затем влево. Для перемещения к символу **о** потребуется сместиться влево, а затем два раза вправо. Если принять, что перемещение влево эквивалентно нулевому биту, а вправо – единичному, можно создать таблицу кодирования, приведенную в таблице 11.4.

Таблица 11.4. Коды Шеннона-Фано для примера предложения

Символ	Количество появлений
Пробел	00
с	010
о	011
и	100
д	1010
h	1011
w	1100
k	1101
H	11100
a	11101
l	11110
m	111110
?	111111

Сейчас мы можем вычислить код для всей фразы. Он начинается с 11100011110000111110100010101100... и содержит всего 131 бит. Если мы предполагаем, что исходная фраза закодирована кодом ASCII, т.е. один байт на символ, то оригинальная фраза заняла бы 256 байт, т.е. мы получаем коэффициент сжатия 54%.

Для декодирования сжатого потока битов мы строим то же дерево, которое было построено на этапе сжатия. Мы начинаем с корневого узла и выбираем из сжатого потока битов по одному биту. Если бит является нулевым, мы перемещаемся влево, если единичным – вправо. Мы продолжаем этот процесс до тех пор, пока не достигнем листа, т.е. символа, после чего выводим символ в поток восстановленных данных. Затем мы снова начинаем процесс с корневого узла дерева с целью извлечения следующего бита. Обратите внимание, что поскольку символы расположены только в листьях дерева, код одного символа не образует первую часть кода другого символа. Благодаря этому, неправильное декодирование сжатых данных невозможно. (Бинарное дерево, в котором данные размещены только в листьях, называется *префиксным деревом (prefix tree)*.)

Однако при этом возникает небольшая проблема: как распознать конец потока битов? В конце концов, внутри класса мы будем объединять восемь битов в байт, после чего выполнять запись байта. Маловероятно, чтобы поток битов содержал количество битов строго кратное 8. Существует два возможных решения этой дилеммы. Первое – закодировать специальный символ, отсутствующий в исходных данных, и назвать его символом конца файла. Второе – записать в сжатый поток длину несжатых данных перед тем, как приступить к сжатию самих данных. Первое решение вынуждает нас найти отсутствующий в исходных данных символ и использовать его (это предполагает передачу этого символа в составе сжатых данных программе восстановления, чтобы она знала, *что* следует искать). Или же можно было бы принять, что хотя символы данных имеют размер, равный размеру одного байта, символ конца файла имеет длину, равную длине слова (и заданное значение, например 256). Однако мы будем использовать второе решение. Перед сжатыми данными мы будем сохранять длину несжатых данных, и таким образом во время восстановления будет в точности известно, сколько символов нужно декодировать.

Еще одна проблема применения кодирования Шеннона-Фано, на которую до сих пор мы не обращали внимания, связана с деревом. Обычно сжатие данных выполняется в целях экономии объема памяти или уменьшения времени передачи данных. Как правило, сжатие и восстановление данных разнесено во времени и пространстве. Однако алгоритм восстановления требует использования дерева. В противном случае невозможно декодировать закодированный поток. Нам доступны две возможности. Первая – сделать дерево статическим. Иначе говоря, одно и то же дерево будет использоваться для сжатия *всех* данных. Для некоторых данных результирующее сжатие будет достаточно оптимальным, для других – весьма далеким от приемлемого. Вторая возможность состоит в том, чтобы тем или иным способом присоединить само дерево к сжатому потоку битов. Конечно, присоединение дерева к сжатым данным ведет к снижению коэффициента сжатия, но с этим ничего нельзя поделать. Вскоре, при рассмотрении следующего алгоритма сжатия, мы покажем, как можно добавить дерево к сжатым данным.

Кодирование Хаффмана

Алгоритм кодирования Хаффмана очень похож на алгоритм сжатия Шеннона-Фано. Этот алгоритм был изобретен Дэвидом Хаффманом (David Huffman) в 1952 году ("A method for the Construction of Minimum-Redundancy Codes" ("Метод создания кодов с минимальной избыточностью")), и оказался еще более удачным, чем алгоритм Шеннона-Фано. Это обусловлено тем, что алгоритм Хаффмана математически гарантированно создает наименьший по размеру код для каждого из символов исходных данных.

Аналогично применению алгоритма Шеннона-Фано, нужно построить бинарное дерево, которое также будет префиксным деревом, где все данные хранятся в листьях. Но в отличие от алгоритма Шеннона-Фано, который является нисходящим, на этот раз построение будет выполняться снизу вверх. Вначале мы выполняем просмотр входных данных, подсчитывая количество появлений значений каждого байта, как это делалось и при использовании алгоритма Шеннона-Фано. Как только эта таблица частоты появления символов будет создана, можно приступить к построению дерева.

Будем считать эти пары символ-количество "пулом" узлов будущего дерева Хаффмана. Удалим из этого пула два узла с наименьшими значениями количества появлений. Присоединим их к новому родительскому узлу и установим значение счетчика родительского узла равным сумме счетчиков его двух дочерних узлов. Поместим родительский узел обратно в пул. Продолжим этот процесс удаления двух узлов и добавления вместо них одного родительского узла до тех пор, пока в пуле не останется только один узел. На этом этапе можно удалить из пула один узел. Он является корневым узлом дерева Хаффмана.

Описанный процесс не очень нагляден, поэтому создадим дерево Хаффмана для предложения "How much wood could a woodchuck chuck?" Мы уже вычислили количество появлений символов этого предложения и представили их в виде таблицы 11.1, поэтому теперь к ней потребуется применить описанный алгоритм с целью построения полного дерева Хаффмана. Выберем два узла с наименьшими значениями. Существует несколько узлов, из которых можно выбрать, но мы выберем узлы "m" и "?". Для обоих этих узлов число появлений символов равно 1. Создадим родительский узел, значение счетчика которого равно 2, и присоединим к нему два выбранных узла в качестве дочерних. Поместим родительский узел обратно в пул. Повторим цикл с самого начала. На этот раз мы выбираем узлы "a" и "l", объединяем их в мини-дерево и помещаем родительский узел (значение счетчика которого снова равно 2) обратно в пул. Снова повторим цикл. На этот раз в нашем распоряжении имеется единственный узел, значение счетчика которого равно 1 (узел "h") и три узла со значениями счетчиков, равными 2 (узел "k" и два родительских узла, которые были добавлены перед этим). Выберем узел "k", присоединим его к узлу "h" и снова добавим в пул родительский узел, значение счетчика которого равно 3. Затем выберем два родительских узла со значениями счет-

чиков, равными 2, присоединим их к новому родительскому узлу со значением счетчика, равным 4, и добавим этот родительский узел в пул. Несколько первых шагов построения дерева Хаффмана и результирующее дерево показаны на рис. 11.2.

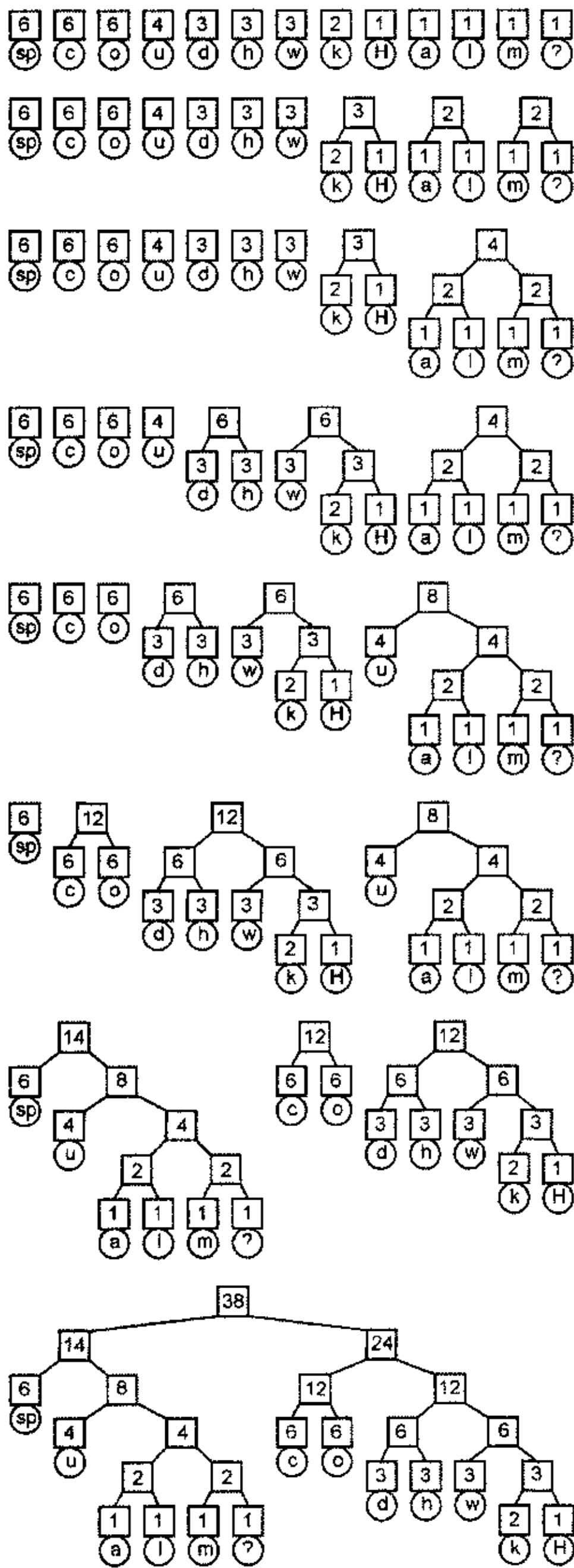


Рисунок 11.2. Построение дерева Хаффмана

Используя это дерево точно так же, как и дерево, созданное для кодирования Шеннона-Фано, можно вычислить код для каждого из символов в исходном предложении и построить таблицу 11.5.

Таблица 11.5. Коды Хаффмана для символов примера предложения

Символ	Количество появлений
Пробел	00
c	100
o	101
u	010
d	1100
h	1101
w	1110
k	11110
H	11111
a	01100
l	01101
m	01110
?	01111

Обратите внимание, что эта таблица кодов – не единственная возможная. Каждый раз, когда имеется три или больше узлов, из числа которых нужно выбрать два, существуют альтернативные варианты результирующего дерева и, следовательно, результирующих кодов. Но на практике все эти возможные варианты деревьев и кодов будут обеспечивать максимальное сжатие. Все они эквивалентны.

Теперь можно вычислить код для всего предложения. Он начинается с битов:

1111110111100001110010100...

и содержит всего 131 бит. Если бы исходное предложение было закодировано кодами ASCII, по одному байту на символ, оно содержало бы 286 битов. Таким образом, в данном случае коэффициент сжатия составляет приблизительно 54%.

Повторим снова, что, как и при применении алгоритма Шеннона-Фано, необходимо каким-то образом сжать дерево и включить его в состав сжатых данных.

Восстановление выполняется совершенно так же, как при использовании кодирования Шеннона-Фано: необходимо восстановить дерево из данных, хранящихся в сжатом потоке, и затем воспользоваться им для считывания сжатого потока битов.

Рассмотрим кодирование Хаффмана с высокоуровневой точки зрения. В ходе реализации каждого из методов сжатия, которые будут описаны в этой главе, мы создадим простую подпрограмму, которая принимает как входной, так и выходной поток, и сжимает все данные входного потока и помещает их в выходной поток.

Эта высокоуровневая подпрограмма **TDHuffmanCompress**, выполняющая кодирование Хаффмана, приведена в листинге 11.5.

Листинг 11.5. Высокоуровневая подпрограмма кодирования Хаффмана

```
procedure TDHuffmanCompress(aInStream, aOutStream : TStream);  
var  
    HTree : THuffmanTree;  
    HCodes : PHuffmanCodes;  
    BitStrm : TtdOutputBitStream;  
    Signature : longint;  
    Size : longint;  
begin  
    { вывести информацию заголовка (сигнатуру и размер несжатых данных) }  
    Signature := TDHuffHeader;  
    aOutStream.WriteBuffer(Signature, sizeof(longint));  
    Size := aInStream.Size;  
    aOutStream.WriteBuffer(Size, sizeof(longint));  
    { при отсутствии данных для сжатия необходимо выйти из подпрограммы }  
    if (Size = 0) then  
        Exit;  
    { подготовка }  
    HTree := nil;  
    HCodes := nil;  
    BitStrm := nil;  
    try  
        { создать сжатый поток битов }  
        BitStrm := TtdOutputBitStream.Create(aOutStream);  
        BitStrm.Name := 'Huffman compressed stream';  
        { распределить память под дерево Хаффмана }  
        HTree := THuffmanTree.Create;  
        { определить распределение символов во входном потоке и выполнить  
          восходящее построение дерева Хаффмана }  
        HTree.CalcCharDistribution(aInStream);  
        { вывести дерево в поток битов для облегчения задачи программы  
          восстановления данных }  
        HTree.SaveToBitStream(BitStrm);  
        { если корневой узел дерева Хаффмана является листом, входной поток состоит  
          лишь из единственного повторяющегося символа, и следовательно, задача  
          выполнена. В противном случае необходимо выполнить сжатие входного  
          потока }  
        if not HTree.RootIsLeaf then begin  
            { распределить память под массив кодов }  
            New(HCodes);  
            { вычислить все коды }  
            HTree.CalcCodes(HCodes^);  
            { сжать символы входного потока в поток битов }  
            DoHuffmanCompression(aInStream, BitStrm, HCodes^);  
        end;
```

```
finally  
  BitStrm.Free;  
  HTree.Free;  
  if (HCodes <> nil) then  
    Dispose(HCodes);  
end;  
end;
```

Код содержит множество элементов, которые мы еще не рассматривали. Но мы вполне можем вначале рассмотреть работу программы в целом, а затем приступить к рассмотрению каждого отдельного этапа. Прежде всего, мы записываем в выходной поток небольшой заголовок, за которым следует значение длины входного потока. Впоследствии эта информация упростит задачу восстановления данных, гарантируя, что сжатый поток соответствует созданному нами. Затем мы создаем объект потока битов, содержащий выходной поток. Следующий шаг – создание экземпляра класса **THuffmanTree**. Этот класс, как вскоре будет показано, будет использоваться для создания дерева Хаффмана и содержит различные методы, помогающие в решении этой задачи. Один из методов этого нового объекта, вызываемых в первую очередь, метод **CalcCharDistribution**, определяет статистическую информацию распределения символов во входном потоке, а затем строит префиксное дерево Хаффмана.

После того, как дерево Хаффмана построено, можно вызвать метод **SaveToBitStream**, чтобы записать структуру дерева в выходной поток.

Затем мы выполняем обработку особого случая и небольшую оптимизацию. Если входной поток состоит всего лишь из нескольких повторений одного и того же символа, корневой узел дерева Хаффмана будет листом. Все префиксное дерево состоит всего из одного узла. В этом случае выходной поток битов будет содержать уже достаточно информации, чтобы программа восстановления могла восстановить исходный файл (мы уже записали в поток битов размер входного потока и единственный бит).

В противном случае входной поток должен содержать, по меньшей мере, два различных символа, и дерево Хаффмана имеет вид обычного дерева, а не единственного узла. В этом случае мы выполняем оптимизацию: вычисляем таблицу кодов для каждого символа, встречающегося во входном потоке. Это позволит сэкономить время на следующем этапе, когда будет выполняться реальное сжатие, поскольку нам не придется постоянно перемещаться по дереву для выполнения кодирования каждого символа. Массив **HCodes** – простой 256-элементный массив, содержащий коды всех символов и построенный посредством вызова метода **CalcCodes** объекта дерева Хаффмана.

И, наконец, когда все эти структуры данных определены, мы вызываем подпрограмму **DoHuffmanCompression**, выполняющую реальное сжатие данных. Код этой подпрограммы приведен в листинге 11.6.

Листинг 11.6. Цикл сжатия Хаффмана

```

procedure DoHuffmanCompression(aInStream : TStream;
                                aBitStream: TtdOutputBitStream;
                                var aCodes : THuffmanCodes);
var
    i : integer;
    Buffer : PByteArray;
    BytesRead : longint;
begin
    GetMem(Buffer, HuffmanBufferSize);
    try
        { сбросить входной поток в начальное состояние }
        aInStream.Position := 0;
        { считать первый блок из входного потока }
        BytesRead := aInStream.Read(Buffer^, HuffmanBufferSize);
        while (BytesRead <> 0) do begin
            { записать строку битов для каждого символа блока }
            for i := 0 to pred(BytesRead) do
                aBitStream.WriteBits(aCodes[Buffer^[i]]);
            { считать следующий блок из входного потока }
            BytesRead := aInStream.Read(Buffer^, HuffmanBufferSize);
        end;
    finally
        FreeMem(Buffer, HuffmanBufferSize);
    end;
end;

```

Подпрограмма **DoHuffmanCompression** распределяет большой буфер для хранения считываемых из входного потока блоков данных, и будет постоянно считывать блоки из входного потока, сжимая их, до тех пор, пока поток не будет исчерпан. Такая буферизация данных служит простым методом оптимизации с целью повышения эффективности всего процесса. Для каждого символа блока подпрограмма записывает соответствующий код, полученный из массива **aCodes**, в выходной поток битов.

После того, как мы ознакомились с выполнением сжатия Хаффмана на высоком уровне, следует рассмотреть класс, выполняющий большую часть вычислений. Это внутренний класс **THuffmanTree**. Объявление связанных с ним типов показано в листинге 11.7.

Вначале мы объявляем узел дерева Хаффмана **THuffmanNode** и массив этих узлов **THuffmanNodeArray** фиксированного размера. Этот массив будет использоваться для создания реальной структуры дерева и будет содержать ровно 511 элементов. Почему именно это количество?

Это число определяется небольшой теоремой (или леммой) о свойствах бинарного дерева, которая еще не упоминалась.

Листинг 11.7. Класс дерева Хаффмана

```

type
  PHuffmanNode = ^THuffmanNode;
  THuffmanNode = packed record
    hnCount : longint;
    hnLeftInx : longint;
    hnRightInx : longint;
    hnIndex : longint;
  end;
  PHuffmanNodeArray = ^THuffmanNodeArray;
  THuffmanNodeArray = array [0..510] of THuffmanNode;
type
  THuffmanCodeStr = string[255];
type
  PHuffmanCodes = ^THuffmanCodes;
  THuffmanCodes = array [0..255] of TtdBitString;
type
  THuffmanTree = class
    private
      FTree : THuffmanNodeArray;
      FRoot : integer;
    protected
      procedure htBuild;
      procedure htCalcCodesPrim( aNodeInx : integer;
                               var aCodeStr : THuffmanCodeStr;
                               var aCodes : THuffmanCodes);
      function htLoadNode(aBitStream : TtdInputBitStream) : integer;
      procedure htSaveNode(aBitStream : TtdOutputBitStream; aNode : integer);
    public
      constructor Create;
      procedure CalcCharDistribution(aStream : TStream);
      procedure CalcCodes(var aCodes : THuffmanCodes);
      function DecodeNextByte(aBitStream : TtdInputBitStream) : byte;
      procedure LoadFromBitStream(aBitStream : TtdInputBitStream);
      function RootIsLeaf : boolean;
      procedure SaveToBitStream(aBitStream : TtdOutputBitStream);

      property Root : integer read FRoot;
  end;

```

Предположим, что дерево содержит только два типа узлов: внутренние, имеющие ровно по два дочерних узла, и листья, не имеющие узлов (иначе говоря, не существует узлов, имеющих только один дочерний узел, – именно такой вид имеет префиксное дерево). Сколько внутренних узлов имеет это дерево, если оно содержит n листьев? Лемма утверждает, что такое дерево содержит ровно $n - 1$ внутренних узлов. Это утверждение можно доказать методом индукции. Когда $n = 1$, лемма явно выполняется, поскольку дерево содержит только корневой узел.

Теперь предположим, что лемма справедлива для всех $i < n$, где $n < 1$, и рассмотрим случай, когда $i = n$. В этом случае дерево должно содержать, по меньшей мере, один внутренний узел – корневой. Этот корневой узел имеет два дочерних дерева: левое и правое. Если левое дочернее дерево имеет x листьев, то, согласно сделанному нами допущению, оно должно содержать $x - 1$ внутренних узлов, поскольку $x < n$. Аналогично, согласно сделанному допущению, если правое дочернее дерево имеет y листьев, оно должно содержать $y - 1$ внутренних узлов. Все дерево содержит n листьев, причем это число должно быть равно $x + y$ (вспомните, что корневой узел является внутренним). Следовательно, количество внутренних узлов равно $(x - 1) + (y - 1) + 1$, что составляет в точности $n - 1$.

Чем же эта лемма может нам помочь? В префиксном дереве все символы должны храниться в листьях. В противном случае было бы невозможно получить однозначные коды. Следовательно, независимо от его внешнего вида, префиксное дерево, подобное дереву Хаффмана, будет содержать не более 511 узлов: не более 256 листьев и не более 255 внутренних узлов. Следовательно, мы должны быть в состоянии реализовать дерево Хаффмана (по крайней мере, обеспечивающее кодирование значений байтов) в виде 511-элементного массива.

Структура узла включает в себя поле счетчика (содержащее значение общего количества появлений символов для самого узла и всех его дочерних узлов), индексы левого и правого дочерних узлов и, наконец, поле, содержащее индекс самого этого узла (эта информация облегчит построение дерева Хаффмана).

Причина выбора типов кода Хаффмана (**THuffmanCodeStr** и **THuffmanCodes**) станет понятной после рассмотрения генерации кодов для каждого из символов.

Конструктор **Create** класса дерева Хаффмана всего лишь выполняет инициализацию внутреннего массива дерева.

Листинг 11.8. Конструирование объекта дерева Хаффмана

```
constructor THuffmanTree.Create;  
var  
  i : integer;  
begin  
  inherited Create;  
  FillChar(FTree, sizeof(FTree), 0);  
  for i := 0 to 510 do  
    FTree[i].hnIndex := i;  
end;
```

Поскольку конструктор не распределяет никакой памяти, и никакое распределение памяти не выполняется ни в каком другом объекте класса, явному деструктору нечего делать. Поэтому по умолчанию класс использует метод **TObject.Destroy**.

Первым методом, вызываемым для дерева Хаффмана в подпрограмме сжатия, был метод **CalcCharDistribution**. Это метод считывает входной поток, вычисляет количество появлений каждого символа, а затем строит дерево.

Листинг 11.9. Вычисление количеств появлений символов

```

procedure THuffmanTree.CalcCharDistribution(aStream : TStream);
var
    i : integer;
    Buffer : PByteArray;
    BytesRead : integer;
begin
    {считывать все байты с поддержанием счетчиков появлений для каждого значения
     байта, начиная с начала потока}
    aStream.Position := 0;
    GetMem(Buffer, HuffmanBufferSize);
    try
        BytesRead := aStream.Read(Buffer^, HuffmanBufferSize);
        while (BytesRead <> 0) do begin
            for i := pred(BytesRead) downto 0 do
                inc(FTree[Buffer^[i]].hnCount);
            BytesRead := aStream.Read(Buffer^, HuffmanBufferSize);
        end;
    finally
        FreeMem(Buffer, HuffmanBufferSize);
    end;
    {построить дерево}
    htBuild;
end;

```

Как видно из листинга 11.9, большая часть кода метода вычисляет количества появлений символов и сохраняет эти значения в первых 256 узлах массива. Для повышения эффективности метод обеспечивает поблочное считывание входного потока (прежде чем выполнить цикл вычисления, он распределяет в куче большой блок памяти, а после вычисления освобождает его). И в завершение, в конце подпрограммы вызывается внутренний метод **htBuild**, выполняющий построение дерева.

Прежде чем изучить реализацию этого важного внутреннего метода, рассмотрим возможную реализацию алгоритма построения дерева. Вспомним, что мы начинаем с создания "пула" узлов, по одному для каждого символа. Мы выбираем два наименьших узла (т.е. два узла с наименьшими значениями счетчиков) и присоединяем их к новому родительскому узлу (устанавливая значение его счетчика равным сумме значений счетчиков его дочерних узлов), а затем помещаем родительский узел обратно в пул. Мы продолжаем этот процесс до тех пор, пока в пуле не останется только один узел. Если вспомнить описанное в главе 9, станет очевидным, какую структуру можно использовать для реализации этого аморфного "пула": очередь по приоритету. Строго говоря, мы должны использовать сортирующее дерево с выбором наименьшего элемента (обычно очередь по приоритету реализуется так, чтобы возвращать наибольший элемент).

Листинг 11.10. Построение дерева Хаффмана

```

function CompareHuffmanNodes(aData1, aData2 : pointer) : integer; far;
var
  Node1 : PHuffmanNode absolute aData1;
  Node2 : PHuffmanNode absolute aData2;
begin
  { ПРИМЕЧАНИЕ: эта подпрограмма сравнения предназначена для реализации
    очереди по приоритету Хаффмана, которая является *сортирующим деревом с
    выбором наименьшего элемента*. Поэтому она должна возвращать элементы в
    порядке, противоположном ожидаемому }
  if (Node1^.hnCount) > (Node2^.hnCount) then
    Result := -1
  else if (Node1^.hnCount) = (Node2^.hnCount) then
    Result := 0
  else
    Result := 1;
end;

procedure THuffmanTree.htBuild;
var
  i : integer;
  PQ : TtdPriorityQueue;
  Node1 : PHuffmanNode;
  Node2 : PHuffmanNode;
  RootNode : PHuffmanNode;
begin
  { создать очередь по приоритету }
  PQ := TtdPriorityQueue.Create(CompareHuffmanNodes, nil);
  try
    PQ.Name := 'Huffman tree minheap';
    { добавить в очередь все ненулевые узлы }
    for i := 0 to 255 do
      if (FTree[i].hnCount <> 0) then
        PQ.Enqueue(@FTree[i]);
    { ОСОБЫЙ СЛУЧАЙ: существует только один ненулевой узел, т.е. входной поток
      состоит только из одного символа, повторяющегося один или более раз.
      В этом случае значение корневого узла устанавливается равным значению
      индекса узла единственного символа }
    if (PQ.Count = 1) then begin
      RootNode := PQ.Dequeue;
      FRoot := RootNode^.hnIndex;
    end
    { в противном случае имеет место обычный случай наличия множества различных
      символов }
    else begin
      { до тех пор, пока в очереди присутствует более одного элемента,
        необходимо выполнять удаление двух наименьших элементов, присоединять их
        к новому родительскому узлу и добавлять его в очередь }

```

```

FRoot := 255;
while (PQ.Count > 1) do begin
  Node1 := PQ.Dequeue;
  Node2 := PQ.Dequeue;
  inc(FRoot);
  RootNode := @FTree[FRoot];
  with RootNode^ do begin
    hnLeftInx := Node1^.hnIndex;
    hnRightInx := Node2^.hnIndex;
    hnCount := Node1^.hnCount + Node2^.hnCount;
  end;
  PQ.Enqueue(RootNode);
end;
end;
finally
  PQ.Free;
end;
end;

```

Мы начинаем с создания экземпляра класса **TtdPriorityQueue**. Мы передаем ему подпрограмму **CompareHuffmanNodes**. Вспомним, что в созданной в главе 9 очереди по приоритету подпрограмма сравнения использовалась для возврата элементов в порядке убывания. Для создания сортирующего дерева с выбором наименьшего элемента, необходимой для создания дерева Хаффмана, мы изменяем цель подпрограммы сравнения, чтобы она возвращала положительное значение, если первый элемент меньше второго, и отрицательное, если он больше.

Как только очередь по приоритету создана, мы помещаем в нее все узлы с ненулевыми значениями счетчиков. В случае существования только одного такого узла, значение поля корневого узла дерева Хаффмана устанавливается равным индексу этого единственного узла. В противном случае мы применяем алгоритм Хаффмана, причем обращение к первому родительскому узлу осуществляется по индексу, равному 256. Удаляя из очереди два узла и помещая в нее новый родительский узел, мы поддерживаем значение переменной **FRoot**, чтобы она указывала на последний родительский узел. В результате по окончании процесса нам известен индекс элемента, представляющего корневой узел дерева.

И, наконец, мы освобождаем объект очереди по приоритету. Теперь дерево Хаффмана полностью построено.

Следующий метод, вызываемый в высокоуровневой подпрограмме сжатия – метод, который выполняет запись дерева Хаффмана в выходной поток битов. По существу, нам необходимо применить какой-либо алгоритм, выполняющий запись достаточного объема информации, чтобы можно было восстановить дерево. Одна из возможностей предусматривает запись символов и их значений счетчика появлений. При наличии этой информации программа восстановления может без труда восстановить дерево Хаффмана, просто вызывая метод **htBuild**. Это кажется здра-

вой идеей, если не учитывать объем, занимаемый таблицей символов и количеств их появлений в сжатом выходном потоке. В этом случае каждый символ занимал бы в выходном потоке полный байт, а его значение счетчика занимало бы определенное фиксированное количество байтов (например, два байта на символ, чтобы можно было подсчитывать вплоть до 65535 появлений). При наличии во входном потоке 100 отдельных символов вся таблица занимала бы 300 байт. Если бы во входном потоке присутствовали все возможные символы, таблица занимала бы 768 байт.

Другой возможный способ – хранение значений счетчика для каждого символа. В этом случае для всех символов, в том числе для отсутствующих во входном потоке, требуется два фиксированных байта. В результате общий размер таблицы во всех ситуациях составил бы 512 байт. Честно говоря, этот результат не многим лучше предыдущего.

Конечно, если бы входной поток был достаточно большим, некоторые из значений счетчиков могли бы превысить размер 2-байтового слова, и для каждого символа пришлось бы использовать по три или даже четыре байта.

Более рациональный подход – игнорировать значения счетчиков символов и сохранять реальную *структуру* дерева. Префиксное дерево содержит два различных вида узлов: внутренние с двумя дочерними узлами и внешние, не имеющие дочерних узлов. Внешние узлы – это узлы, содержащие символы. Выполним обход дерева, применив один из обычных методов обхода (фактически, мы будем использовать метод обхода в ширину). Для каждого достигнутого узла будем записывать нулевой бит, если узел является внутренним, или единичный бит, если узел является внешним, за которым будет следовать представляемый узлом символ. Код реализации метода **SaveToBitStream** и вызываемого им рекурсивного метода **htSaveNode**, который выполняет реальный обход дерева и запись информации в поток битов, представлен в листинге 11.11.

Листинг 11.11. Запись дерева Хаффмана в поток битов

```
procedure THuffmanTree.htSaveNode(aBitStream : TtdOutputBitStream;
                                   aNode : integer);
begin
  {если этот узел является внутренним, выполнить запись нулевого бита, затем
   левого дочернего дерева, а затем – правого дочернего дерева}
  if (aNode >= 256) then begin
    aBitStream.WriteBit(false);
    htSaveNode(aBitStream, FTree[aNode].hnLeftInx);
    htSaveNode(aBitStream, FTree[aNode].hnRightInx);
  end
  {в противном случае узел является листом и нужно записать единичный бит,
   а затем символ}
  else begin
    aBitStream.WriteBit(true);
    aBitStream.WriteByte(aNode); {aNode – символ}
  end;
end;
```

```

procedure THuffmanTree.SaveToBitStream(aBitStream : TtdOutputBitStream);
begin
    htSaveNode(aBitStream, FRoot);
end;

```

Если бы во входном потоке присутствовало 100 отдельных символов, он содержал бы 99 внутренних узлов, и требовалось бы всего 199 битов для хранения информации об узлах плюс 100 байтов для хранения самих символов – всего около 125 байтов. Если бы во входном потоке были представлены все символы, требовалось бы 511 битов для хранения информации об узлах плюс место для хранения 256 символов. Таким образом, всего для хранения дерева требовалось бы 320 байтов.

Полный код подпрограммы сжатия дерева Хаффмана можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHuffmn.pas**.

После того, как мы рассмотрели реализацию сжатия Хаффмана, приступим к вопросу решения задачи восстановления данных. Код подпрограммы **TDHuffmanDecompress**, управляющей этим процессом, приведен в листинге 11.12.

Листинг 11.12. Подпрограмма TDHuffmanDecompress

```

procedure TDHuffmanDecompress(aInStream, aOutStream : TStream);
var
    Signature : longint;
    Size : longint;
    HTree : THuffmanTree;
    BitStrm : TtdInputBitStream;
begin
    {выполнить проверку на предмет того, что входной поток является потоком,
     правильно закодированным методом Хаффмана}
    aInStream.Seek(0, soFromBeginning);
    aInStream.ReadBuffer(Signature, sizeof(Signature));
    if (Signature <> TDHuffHeader) then
        raise EtdHuffmanException.Create(
            FmtLoadStr(tdeHuffBadEncodedStrm, [UnitName, 'TDHuffmanDecompress']));
    aInStream.ReadBuffer(Size, sizeof(longint));
    {если данные для восстановления отсутствуют, осуществить выход из
     подпрограммы}
    if (Size = 0) then
        Exit;
    {подготовиться к восстановлению}
    HTree := nil;
    BitStrm := nil;
    try
        {создать поток битов}
        BitStrm := TtdInputBitStream.Create(aInStream);
        BitStrm.Name := 'Huffman compressed stream';
    finally

```

```

{создать дерево Хаффмана}
HTree := THuffmanTree.Create;
{считать данные дерева из входного потока}
HTree.LoadFromBitStream(BitStrm);
{если корневой узел дерева Хаффмана является листом, исходный поток состоит
  только из повторений одного символа}
if HTree.RootIsLeaf then
  WriteMultipleChars(aOutputStream, AnsiChar(HTree.Root), Size)
{в противном случае выполнить восстановление символов входного потока
  посредством использования дерева Хаффмана}
else
  DoHuffmanDecompression(BitStrm, aOutputStream, HTree, Size);
finally
  BitStrm.Free;
  HTree.Free;
end;
end;

```

Прежде всего, мы проверяем, начинается ли поток с корректной сигнатуры. Если нет, не имеет смысла продолжать процесс, поскольку поток явно содержит ошибки.

Затем выполняется считывание длины несжатых данных, и если она равна нулю, задача выполнена. В противном случае необходимо проделать определенную работу. В этом случае мы создаем входной поток битов, содержащий входной поток. Затем мы создаем объект дерева Хаффмана, который будет выполнять большую часть работы, и вынуждаем его выполнить собственное считывание из входного потока битов (вызывая для этого метод **LoadFromBitStream**). Если дерево Хаффмана представляет единственный символ, исходный поток восстанавливается в виде повторений этого символа. В противном случае мы вызываем подпрограмму **DoHuffmanDecompression** для выполнения восстановления данных. Код этой подпрограммы приведен в листинге 11.13.

Листинг 11.13. Подпрограмма DoHuffmanDecompression

```

procedure DoHuffmanDecompression( aBitStream : TtdInputBitStream;
                                   aOutputStream : TStream;
                                   aHTree : THuffmanTree;
                                   aSize : longint);

var
  CharCount : longint;
  Ch : byte;
  Buffer : PByteArray;
  BufEnd : integer;

```

```

begin
  GetMem(Buffer, HuffmanBufferSize);
  try
    {предварительная установка переменных цикла}
    BufEnd := 0;
    CharCount := 0;
    {повторять процесс до тех пор, пока не будут восстановлены все символы}
    while (CharCount < aSize) do begin
      {считать следующий байт}
      Ch := aHTree.DecodeNextByte(aBitStream);
      Buffer^[BufEnd] := Ch;
      inc(BufEnd);
      inc(CharCount);
      {если буфер заполнен, необходимо выполнить его запись}
      if (BufEnd = HuffmanBufferSize) then begin
        aOutStream.WriteBuffer(Buffer^, HuffmanBufferSize);
        BufEnd := 0;
      end;
    end;
    {если в буфере остались какие-либо данные, необходимо выполнить его запись}
    if (BufEnd <> 0) then
      aOutStream.WriteBuffer(Buffer^, BufEnd);
  finally
    FreeMem(Buffer, HuffmanBufferSize);
  end;
end;

```

По существу подпрограмма представляет собой цикл, внутри которого многократно выполняется декодирование байтов и заполнение буфера. Когда буфер заполняется, мы записываем его в выходной поток и начинаем заполнять его снова. Декодирование выполняется при помощи метода **DecodeNextByte** класса **THuffmanTree**.

Листинг 11.14. Метод DecodeNextByte

```

function THuffmanTree.DecodeNextByte(aBitStream : TtdInputBitStream) : byte;
var
  NodeInx : integer;
begin
  NodeInx := FRoot;
  while (NodeInx >= 256) do begin
    if not aBitStream.ReadBit then
      NodeInx := FTree[NodeInx].hnLeftInx
    else
      NodeInx := FTree[NodeInx].hnRightInx;
    end;
  Result := NodeInx;
end;

```

Этот метод крайне прост. Он просто начинает обработку с корневого узла дерева Хаффмана, а затем для каждого бита, считанного из входного потока битов, в зависимости от того, был ли он нулевым или единичным, выполняет переход по левой или правой связи. Как только подпрограмма достигает листа, она возвращает индекс достигнутого узла (его значение будет меньше или равно 255). Этот узел является декодированным байтом.

Полный код выполнения восстановления дерева Хаффмана можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDHuffman.pas**.

Кодирование с использованием скошенного дерева

Как было показано, и кодирование Шеннона-Фано, и кодирование Хаффмана связано со значительной проблемой – необходимостью поставлять дерево вместе со сжатыми данными. Это является недостатком, поскольку трудно добиться существенного сжатия дерева, что ведет к снижению коэффициента сжатия данных. Еще один недостаток применения этих методов состоит в том, что входные данные приходится считывать дважды: первый раз для вычисления частоты появления символов в данных, а второй – сразу после построения дерева при выполнении действительного кодирования данных.

Не существует ли какой-либо способ исключения необходимости поставки дерева или двукратного считывания входных данных (желательно было бы избавиться от обоих этих недостатков)? Существует вариант сжатия Хаффмана, называемый адаптивным кодированием Хаффмана, который позволяет это сделать. Однако в этой главе мы рассмотрим малоизвестную адаптивную технологию, использующую скошенные деревья, с которыми мы впервые встретились в главе 8.

Дуглас В. Джонс (Douglas W. Jones) разработал сжатие с использованием скошенного дерева в 1988 году [8]. Если помните, в главе 8 говорилось, что скошенные деревья – это метод балансировки дерева бинарного поиска посредством скоса достигнутого узла к корневому узлу. Таким образом, после отыскания узла он перемещается к корневому узлу с помощью ряда поворотов, называемых операциями двустороннего и одностороннего поворота. В результате скоса узлы, обращение к которым осуществляется наиболее часто, оказываются, как правило, в верхней части дерева, а узлы, обращение к которым происходит реже – ближе к листьям. Если применить эту стратегию к префиксному дереву и закодировать символы, как это делалось при использовании алгоритмов Хаффмана и Шеннона-Фано (левая связь кодируется нулевым битом, а правая единичным), окажется, что со временем кодирование данного символа будет меняться. Дерево будет приспосабливаться к частоте появления закодированных символов. Более того, наиболее часто используемые символы будут располагаться вблизи вершины дерева и, следовательно, как правило, их коды будут короче кодов реже используемых символов.

Следует признать, что скошенные деревья были разработаны для оптимизации деревьев бинарного поиска, т.е. упорядоченных деревьев. Частично их полезность обусловлена тем, что операции скоса поддерживают упорядоченность во время различных поворотов и трансформаций. Префиксные деревья не упорядочены, поэтому можно избежать большей части сложного манипулирования указателями, связанного с поворотами. Кроме того, потребуется обеспечить, чтобы листья оставались листьями. В противном случае дерево перестало бы быть префиксным. Поэтому мы будем использовать скос, в результате которого родительский узел листа перемещается ближе к корневому узлу.

Код реализации базового алгоритма выполнения сжатия выглядит подобно приведенному в листинге 11.15.

Листинг 11.15. Базовый алгоритм сжатия с использованием скошенного дерева

```
procedure TDSplayCompress(aInStream, aOutStream : TStream);
var
  STree : TSplayTree;
  BitStrm : TtdOutputBitStream;
  Signature : longint;
  Size : longint;
begin
  { вывести информацию заголовка (сигнатуру и размер несжатых данных) }
  Signature := TDSplayHeader;
  aOutStream.WriteBuffer(Signature, sizeof(longint));
  Size := aInStream.Size;
  aOutStream.WriteBuffer(Size, sizeof(longint));
  { в случае отсутствия данных для сжатия выйти из подпрограммы }
  if (Size = 0) then
    Exit;
  { подготовка }
  STree := nil;
  BitStrm := nil;
  try
    { создать сжатый поток битов }
    BitStrm := TtdOutputBitStream.Create(aOutStream);
    BitStrm.Name := 'Splay compressed stream';
    { создать скошенное дерево }
    STree := TSplayTree.Create;
    { сжать символы входного потока и поместить их в поток битов }
    DoSplayCompression(aInStream, BitStrm, STree);
  finally
    BitStrm.Free;
    STree.Free;
  end;
end;
```

Для пометки выходного потока как сжатого с использованием скошенного дерева в выходной поток мы записываем сигнатуру типа длинного целого, а затем записываем размер несжатого потока. Если входной поток пуст, выполняется выход из подпрограммы, – в этом случае задача выполнена. В противном случае мы создаем выходной поток битов, который будет содержать выходной поток и скошенное дерево. Затем для выполнения реального сжатия мы вызываем метод **DoSplayCompression**. Код этой подпрограммы приведен в листинге 11.16.

Листинг 11.16. Цикл выполнения сжатия с использованием скошенного дерева

```
procedure DoSplayCompression(aInStream : TStream;
                             aBitStream : TtdOutputBitStream;
                             aTree : TSplayTree);

var
  i : integer;
  Buffer : PByteArray;
  BytesRead : longint;
  BitString : TtdBitString;
begin
  GetMem(Buffer, SplayBufferSize);
  try
    { сбросить входной поток в исходное состояние }
    aInStream.Position := 0;
    { считать первый блок из входного потока }
    BytesRead := aInStream.Read(Buffer^, SplayBufferSize);
    while (BytesRead <> 0) do begin
      { записать строку битов для каждого символа в блоке }
      for i := 0 to pred(BytesRead) do
        aTree.EncodeByte(aBitStream, Buffer^[i]);
      { считать следующий блок из входного потока }
      BytesRead := aInStream.Read(Buffer^, SplayBufferSize);
    end;
  finally
    FreeMem(Buffer, SplayBufferSize);
  end;
end;
```

Фактически эта подпрограмма представляется собой подпрограмму выполнения вложенного цикла. Во внешнем цикле выполняется поблочное считывание входного потока, а во внутреннем (через вызов метода **EncodeByte** скошенного дерева) – кодирование каждого байта текущего блока и запись результирующего кода в выходной поток битов.

Теперь пора рассмотреть внутренний класс **TSplayTree**, который выполняет основную часть работы по реализации алгоритма сжатия с использованием скошенного дерева. Код интерфейса этого класса показан в листинге 11.17.

Листинг 11.17. Класс сжатия с использованием скошенного дерева

```

type
  PSplayNode = ^TSplayNode;
  TSplayNode = packed record
    hnParentInx: longint;
    hnLeftInx : longint;
    hnRightInx : longint;
    hnIndex : longint;
  end;
  PSplayNodeArray = ^TSplayNodeArray;
  TSplayNodeArray = array [0..510] of TSplayNode;
type
  TSplayTree = class
    private
      FTree : TSplayNodeArray;
      FRoot : integer;
    protected
      procedure stConvertCodeStr(const aRevCodeStr : ShortString;
                               var aBitString : TtdBitString);

      procedure stInitialize;
      procedure stSplay(aNodeInx : integer);
    public
      constructor Create;
      procedure EncodeByte(aBitStream : TtdOutputBitStream; aValue : byte);
      function DecodeByte(aBitStream : TtdInputBitStream) : byte;
  end;

```

Хотя можно было бы воспользоваться ориентированным на узлы деревом, как это делалось в главе 8, поскольку нам известно количество символов в используемом алфавите (в общем случае используется алфавит, содержащий 256 символов), проще отдать предпочтение применению ориентированной на массивы системе, подобной структуре данных типа сортирующего дерева и дерева Хаффмана. Еще один аргумент в пользу перехода на использование других структур данных состоит в том, что в случае применения неадаптивных методов сжатия можно было строить таблицу кодов, так как они были статическими. При использовании сжатия с применением скошенного дерева битовый код символа зависит от состояния скошенного дерева и момента времени кодирования символа. В этом случае мы больше не можем использовать статическую таблицу. Следовательно, одно из выдвигаемых требований – возможность быстрого и эффективного поиска символа в дереве (предпочтительно при помощи алгоритма типа $O(1)$ – мы *не хотим* его искать). Как только символ и его узел листа определены, можно легко выполнить обход вверх по дереву до корневого узла с целью вычисления кода символа (вообще говоря, мы получим битовый код с обратным порядком следования битов, но с помощью стека его легко можно изменить на противоположный).

Обработка начинается с известного состояния дерева. Можно было бы определить дерево, отражающее частоту употребления букв английского алфавита или какое либо иное распределение символов, но на практике значительно проще создать идеально сбалансированное дерево. В этом случае каждый узел имеет три "указателя", которые в действительности являются всего лишь индексами других узлов в массиве, и мы определяем его таким же образом, как делали при работе с сортирующим деревом: дочерние узлы узла с индексом n располагаются в позициях $2n + 1$ и $2n + 2$, а его родительский узел – в позиции $(n - 1)/2$. Поскольку в действительности узлы не будут перемещаться в массив (мы собираемся манипулировать только индексами), позиции листьев всегда будут известны. Они всегда будут занимать одни и те же позиции в массиве: **#0** всегда будет находиться в позиции с индексом 255, **#1** – в позиции с индексом 256 и т.д. Код метода, выполняющего инициализацию дерева, показан в листинге 11.18. Этот метод вызывается из конструктора **Create**.

Листинг 11.18. Метод `stInitialize`

```

procedure TSplayTree.stInitialize;
var
  i : integer;
begin
  {создать полностью сбалансированное дерево; корневой узел будет
   соответствовать нулевому элементу; родительский узел узла n будет
   располагаться в позиции (n-1)/2, а его дочерние узлы -
   в позициях 2n+1 и 2n+2}
  FillChar(FTree, sizeof(FTree), 0);
  for i := 0 to 254 do begin
    FTree[i].hnLeftInx := (2 * i) + 1;
    FTree[i].hnRightInx := (2 * i) + 2;
  end;
  for i := 1 to 510 do
    FTree[i].hnParentInx := (i - 1) div 2;
end;

constructor TSplayTree.Create;
begin
  inherited Create;
  stInitialize;
end;

```

При сжатии символа мы находим его узел в дереве. Затем мы выполняем переходы вверх по дереву, сохраняя соответствующие биты в стеке (левой связи соответствует нулевой бит, а правой – единичный). По достижении корневого узла можно вытолкнуть биты из стека. Они определяют код символа (в коде, приведенном в листинге 11.19, в качестве стека используется короткая строка).

Затем выполняется скос родительского узла по направлению к корневому узлу. Мы не выполняем скос к корню самого узла символа ввиду того, что требуется

сохранить размещение символов в узлах листьев. В противном случае было бы совершенно исключено, чтобы код одного символа становился началом кода следующего. Скос родительского узла повлечет "перетаскивание" вместе с ним и дочернего узла. В результате чаще используемые символы окажутся ближе к верхушке дерева.

Листинг 11.19. Методы EncodeByte и stSplay

```
procedure TSplayTree.EncodeByte(aBitStream : TtdOutputBitStream;
                                aValue : byte);

var
  NodeInx : integer;
  ParentInx : integer;
  RevCodeStr : ShortString;
  BitString : TtdBitString;
begin
  {начиная с узла aValue, сохранить на каждом шаге (0) бит при перемещении вверх
   по дереву по левой связи и (1) бит при перемещении по правой связи}
  RevCodeStr := '';
  NodeInx := aValue + 255;
  while (NodeInx <> 0) do begin
    ParentInx := FTree[NodeInx].hnParentInx;
    inc(RevCodeStr[0]);
    if (FTree[ParentInx].hnLeftInx = NodeInx) then
      RevCodeStr[length(RevCodeStr)] := '0'
    else
      RevCodeStr[length(RevCodeStr)] := '1';
    NodeInx := ParentInx;
  end;
  {преобразовать строковый код в строку битов}
  stConvertCodeStr(RevCodeStr, BitString);
  {записать строку битов в поток битов}
  aBitStream.WriteBits(BitString);
  {выполнить скос узла}
  stSplay(aValue + 255);
end;

procedure TSplayTree.stConvertCodeStr(const aRevCodeStr : ShortString;
                                       var aBitString : TtdBitString);

var
  ByteNum : integer;
  i : integer;
  Mask : byte;
  Accum : byte;
begin
  {подготовиться к выполнению цикла преобразования}
  ByteNum := 0;
  Mask := 1;
  Accum := 0;
```



```

{преобразовать порядок следования битов на противоположный}
for i := length(aRevCodeStr) downto 1 do begin
  if (aRevCodeStr[i] = '1') then
    Accum := Accum or Mask;
  Mask := Mask shl 1;
  if (Mask = 0) then begin
    aBitString.bsBits[ByteNum] := Accum;
    inc(ByteNum);
    Mask := 1;
    Accum := 0;
  end;
end;
{сохранить биты, расположенные слева от текущего}
if (Mask <> 1) then
  aBitString.bsBits[ByteNum] := Accum;
{сохранить двоичный код в массиве кодов}
aBitString.bsCount := length(aRevCodeStr);
end;

procedure TSplayTree.stSplay(aNodeInx : integer);
var
  Dad : integer;
  GrandDad : integer;
  Uncle : integer;
begin
  {выполнить скос узла}
  repeat
    {извлечь родительский узел данного узла}
    Dad := FTree[aNodeInx].hnParentInx;
    {если родительский узел является корневым, задача выполнена}
    if (Dad = 0) then
      aNodeInx := 0
    {в противном случае необходимо выполнить поворот узла на 90 градусов с целью
    его перемещения вверх по дереву}
  else begin
    {извлечь родительский узел родительского узла}
    GrandDad := FTree[Dad].hnParentInx;
    {выполнить поворот на 90 градусов (т.е. поменять местами узел и его
    узел-дядю)}
    if (FTree[GrandDad].hnLeftInx = Dad) then begin
      Uncle := FTree[GrandDad].hnRightInx;
      FTree[GrandDad].hnRightInx := aNodeInx;
    end
    else begin
      Uncle := FTree[GrandDad].hnLeftInx;
      FTree[GrandDad].hnLeftInx := aNodeInx;
    end;
    if (FTree[Dad].hnLeftInx = aNodeInx) then
      FTree[Dad].hnLeftInx := Uncle

```

```

else
    FTree[Dad].hnRightInx := Uncle;
    FTree[Uncle].hnParentInx := Dad;
    FTree[aNodeInx].hnParentInx := GrandDad;
    {возобновить цикл с узла-деда}
    aNodeInx := GrandDad;
end;
until (aNodeInx = 0);
end;

```

При восстановлении мы устанавливаем дерево в исходную конфигурацию, как это делалось на этапе сжатия. Затем мы по одному выбираем биты из потока битов и выполняем обычное перемещение вниз по дереву. По достижении листа, содержащего символ (который мы выводим в качестве восстановленных данных), мы будем выполнять скос родительского узла данного узла к корню дерева. При условии, что обновление дерева выполняется одинаково и во время сжатия, и во время восстановления, алгоритм декодирования может поддерживать дерево в том же состоянии, что и на соответствующем этапе выполнения алгоритма кодирования.

Листинг 11.20. Базовый алгоритм восстановления скошенного дерева

```

procedure TDSplayDecompress(aInStream, aOutStream : TStream);
var
    Signature : longint;
    Size : longint;
    STree : TSplayTree;
    BitStrm : TtdInputBitStream;
begin
    {выполнить проверку того, что входной поток является корректно
     закодированным с использованием скошенного дерева}
    aInStream.Seek(0, soFromBeginning);
    aInStream.ReadBuffer(Signature, sizeof(Signature));
    if (Signature <> TDSplayHeader) then
        raise EtdSplayException.Create(FmtLoadStr(tdeSplyBadEncodedStrm,
            [UnitName, 'TDSplayDecompress']));
    aInStream.ReadBuffer(Size, sizeof(longint));
    {при отсутствии данных для восстановления выйти из подпрограммы}
    if (Size = 0) then
        Exit;
    {подготовиться к восстановлению}
    STree := nil;
    BitStrm := nil;
    try
        {создать поток битов}
        BitStrm := TtdInputBitStream.Create(aInStream);
        BitStrm.Name := 'Splay compressed stream';
    except

```

```

    {создать скошенное дерево}
    STree := TSplayTree.Create;
    {восстановить символы входного потока с использованием скошенного дерева}
    DoSplayDecompression(BitStrm, aOutStream, STree, Size);
finally
    BitStrm.Free;
    STree.Free;
end;
end;

```

В процессе восстановления потока вначале за счет проверки сигнатуры выполняется проверка того, что поток является сжатым с использованием скошенного дерева. Затем мы считываем размер несжатых данных и осуществляем выход из подпрограммы, если он равен нулю.

При наличии данных для восстановления мы создаем входной поток битов, который будет содержать входной поток и скошенное дерево. Затем для выполнения реального декодирования вызывается метод **DoSplayDecompression** (см. листинг 11.21).

Листинг 11.21. Цикл восстановления скошенного дерева

```

procedure DoSplayDecompression(aBitStream : TtdInputBitStream;
                               aOutStream : TStream;
                               aTree : TSplayTree;
                               aSize : longint);

var
    CharCount : longint;
    Ch : byte;
    Buffer : PByteArray;
    BufEnd : integer;
begin
    GetMem(Buffer, SplayBufferSize);
    try
        {предварительная установка значений переменных цикла}
        BufEnd := 0;
        CharCount := 0;
        {повторять цикл до тех пор, пока не будут восстановлены все символы}
        while (CharCount < aSize) do begin
            {считать следующий байт}
            Buffer^[BufEnd] := aTree.DecodeByte(aBitStream);
            inc(BufEnd);
            inc(CharCount);
            {записать буфер в случае его заполнения}
            if (BufEnd = SplayBufferSize) then begin
                aOutStream.WriteBuffer(Buffer^, SplayBufferSize);
                BufEnd := 0;
            end;
        end;
    end;

```

```

    {запустить любые оставшиеся в буфере данные}
    if (BufEnd <> 0) then
        aOutputStream.WriteBuffer(Buffer^, BufEnd);
    finally
        FreeMem(Buffer, SplayBufferSize);
    end;
end;

```

Как и в цикле декодирования дерева Хаффмана, буфер заполняется декодированными байтами с последующей их записью в выходной поток. Реальное декодирование и запись выполняется методом **DecodeByte** класса скошенного дерева.

Листинг 11.22. Метод **TSplayTree.DecodeByte**

```

function TSplayTree.DecodeByte(aBitStream : TtdInputBitStream) : byte;
var
    NodeInx : integer;
begin
    {переместиться вниз по дереву в соответствии с битами потока битов, начиная с
    корневого узла}
    NodeInx := 0;
    while NodeInx < 255 do begin
        if not aBitStream.ReadBit then
            NodeInx := FTree[NodeInx].hnLeftInx
        else
            NodeInx := FTree[NodeInx].hnRightInx;
    end;
    {вычислить байт, исходя из значения индекса конечного узла}
    Result := NodeInx - 255;
    {выполнить скос узла}
    stSplay(NodeInx);
end;

```

Этот метод всего лишь выполняет перемещение вниз по дереву, считывая биты из входного потока битов и осуществляя перемещение по левой или правой связи, в зависимости от того, является ли текущий бит нулевым или единичным. И, наконец, достигнутый узел листа скашивается по направлению к корневому узлу с целью повторения того, что произошло во время сжатия. Одинаковое выполнение скоса во время сжатия и восстановления гарантирует правильность декодирования данных.

Полный код реализации алгоритма сжатия с использованием скошенного дерева можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDSplyCm.pas**.

Сжатие с использованием словаря

Вплоть до 1977 года, основные усилия в области исследования алгоритмов сжатия концентрировались вокруг алгоритмов кодирования с минимальной избыточностью, подобных алгоритмам Шеннона-Фано или Хаффмана, и были посвящены либо преобразованию их в динамические (чтобы таблица кодов не являлась частью сжатого файла), либо повышению быстродействия, уменьшению объема используемой памяти или увеличению эффективности. Затем неожиданно два израильских исследователя, Якоб Зив (Jacob Ziv) и Абрахам Лемпель (Abraham Lempel), представили принципиально иной метод сжатия и положили начало исследованиям в совершенно другом направлении. Их основная идея заключалась в кодировании не отдельных символов, а *строк* символов. Они задались целью использовать *словарь* ранее встречавшихся в сжимаемом файле *фраз* для кодирования последующих фраз.

Предположим, что имеется обычный словарь какого-либо языка. Каждое встречающееся в данном текстовом файле слово должно быть представлено в словаре. Если бы и программа сжатия, и программа восстановления имели доступ к электронной версии этого словаря, кодирование отдельных слов в текстовом файле можно было бы выполнить путем указания номера страницы и номера слова на этой странице. Вполне можно было считать, что 2-байтового целочисленного значения окажется достаточно для хранения номеров страниц (найдется не особенно много словарей, содержащих более 65536 страниц), а байта должно быть достаточно для хранения номера слова на странице (как и в предыдущем случае, обычно на одной странице словаря приводится определение не более 256 слов). Следовательно, независимо от реальной длины слова в текстовом файле, оно замещалось бы тремя байтами. Понятно, что сжатие коротких слов, таких как "в", "из", "на" и тому подобных, приводило бы к увеличению размера сжатых данных, а не к уменьшению, однако большинство слов содержит три и больше букв. Поэтому, как правило, общий размер сжатого файла должен быть меньше размера исходного файла.

Описание сжатия LZ77

В основе алгоритма, разработанного Зивом и Лемпелем, лежит сжатие с использованием строк словаря. Однако вместо того, чтобы использовать статический, заранее сгенерированный словарь, предложенный ими алгоритм генерирует словарь "на лету", на основе данных, которые программа сжатия уже встретила во входном файле. А вместо использования номеров страниц и слов они предложили выводить значения расстояния и длины. Работа алгоритма выполняется следующим образом: в ходе считывания входного файла предпринимается попытка сопоставить набор символов в текущей позиции с чем-либо уже встречавшимся во входном файле. При обнаружении совпадения вычисляется расстояние совпадающей строки от текущей позиции и количество совпадающих байтов (длина). В случае обнаружения нескольких совпадений выбирается самое длинное из них.

Рассмотрим краткий пример. Предположим, что мы выполняем сжатие предложения:

```
a cat is a cat is a cat
```

Первый символ **"a"** не совпадает ни с одной уже встречавшейся строкой (да просто потому, что ни одна строка еще не встречалась!), поэтому мы выводим его в существующем виде в сжатый поток битов. Это же следовало бы сделать с последующим пробелом и символом **"c"**. Следующий символ **"a"** совпадает с предшествующим символом **"a"**, но на этом соответствие заканчивается. Мы не можем сопоставить никакие другие строки. Примем правило, что прежде чем делать что-нибудь другое, необходимо устанавливать соответствие не менее чем для трех символов. Поэтому мы выводим в выходной поток символ **"a"**, а также символы **"t"**, пробел, **"i"**, **"s"** и пробел. Текущую ситуацию можно представить следующим образом:

```
-----+
a cat is |a cat is a cat
-----+^
```

где встретившиеся символы заключены в рамку (в программировании такую рамку называют *скользящим окном (sliding window)*), а текущая позиция обозначена знаком вставки (^).

Теперь становится действительно интересно. Набор символов **"a cat is"** в текущей позиции совпадает со строкой, уже встречавшейся ранее. Совпадающая строка начинается за девять символов до текущей позиции, причем совпадают девять символов. Поэтому можно вывести пару значений расстояние/длина, которые в данном случае представлены строкой **<9, 9>** (или определенной последовательностью битов), в выходной файл, а затем продвинуться на девять символов. Теперь текущее состояние можно представить следующим образом:

```
-----+
a cat is a cat is |a cat
-----+^
```

Но теперь снова набор символов в текущей позиции можно сопоставить с уже встречавшейся строкой. Можно сопоставить пять символов с пятью символами, расположенными либо на 9, либо на 18 символов раньше. Выберем первую возможность, **<9, 5>**. В результате окончательно сжатый поток будет выглядеть следующим образом:

```
a cat is <9, 9><9, 5>
```

Восстановление этого потока битов также не представляет особой сложности. В процессе восстановления мы строим буфер восстановленных символов, позволяющий декодировать пары значений расстояние/длина или коды. Литеральные символы в сжатом потоке выводятся в восстановленный поток в том виде, каком они записаны.

Первые девять символов в сжатом потоке являются литеральными, поэтому они выводятся в восстановленный поток в существующем виде. Одновременно создается буфер (называемый скользящим окном). На этом этапе буфер выглядит следующим образом:

```
-----+  
a cat is |  
-----+
```

Следующий код в сжатом потоке – пара расстояние/длина, **<9,9>**. Это означает, что нужно "вывести девять символов, расположенные в буфере в девяти байтах от его конца". Этими девятью символами являются **"a cat is"**, поэтому они выводятся в восстановленный поток и добавляются в буфер. Иначе говоря, скользящее окно приобретает вид:

```
-----+  
a cat is a cat is |  
-----+
```

И снова, следующий код в сжатом потоке – пара расстояние/длина, **<9,5>**. Уверен, что читатели без труда смогут выполнить декодирование, используя описанный буфер.

При рассмотрении этого небольшого примера мы по существу не нуждались в словаре, как таковом. Мы всего лишь воспользовались своим воображением для отыскания наиболее длинной строки в ранее просмотренном наборе символов. На практике ранее просмотренные фразы (или лексемы) нужно было бы добавлять в словарь (который в действительности представляет собой хеш-таблицу), а затем просматривать лексему в текущей позиции этого словаря и пытаться устанавливать соответствие с ранее встречавшимися символами.

Между прочим, определение буфера ранее встречавшихся символов как "скользящего окна" означает, что при попытке найти возможное соответствие рассматриваются только *n* байт. Обычно *n* равно 4 или 8 Кб (используемый в программе **PKZIP** алгоритм Deflate (понижения порядка) может использовать скользящее окно размером до 32 Кб). При перемещении текущей позиции вперед скользящее окно перемещается вперед по уже просмотренным данным. Возникает вопрос, зачем это делается? Почему бы не использовать весь ранее просмотренный текст? Ответ на этот вопрос обусловлен общей структурой текста. В общем случае считываемый и записываемый текст подчиняются так называемому правилу *локальности ссылок* (*locality of reference*). Это означает, что, как правило, в текстовом файле более вероятно совпадение близко расположенных символов, а не расположенных вдали друг от друга. Например, в романе описываемые действующие лица и места протекания событий обычно группируются в главы или разделы глав. В то же время часто используемые слова и фразы типа "и", "в" и "он сказал" могут встречаться в любом месте романа.

Для других текстов, таких как учебные пособия, подобные этому, также характерна локальность ссылок. Поэтому согласно правилу локальности ссылок имеет

смысл ограничить объем ранее встречавшегося текста, просматриваемого с целью установки соответствия между строками. Еще одна веская причина ограничения размера скользящего окна связана с необходимостью замедлить сжатие с увеличением объема просматриваемого текста.

Теперь рассмотрим, как выполняется кодирование пары значений расстояние/длина. До сих пор мы не обращали на это внимание, однако целесообразно сжимать их как можно больше. Если скользящее окно охватывает последние 4 Кб текста, значение расстояния можно закодировать 12 битами (2^{12} как раз и составляет 4 Кб). Если максимальную длину проверяемых и сопоставляемых строк ограничить 15 символами, это значение можно закодировать 4 битами, а пару расстояние/длина – двумя полными байтами. Можно было бы использовать также окно с размером равным 8 Кб и максимум семь сопоставляемых символов, и при этом длина кода по-прежнему не превышала бы 2 байтов. Сжатый поток можно рассматривать как поток байтов, а не как явно более сложный поток битов, который мы использовали при генерировании кодов минимальной длины. Кроме того, если длину кодов значений расстояние/длина ограничить 2 байтами, можно сжимать строки, содержащие, по меньшей мере, три символа и совпадающие с какими-то уже встречавшимися строками, при этом не обращать внимания на совпадения одного или двух символов, поскольку сжиматься они не будут.

Мы кратко описали суть алгоритма Зива и Лемпеля (Лемпеля-Зива), на который в настоящее время ссылаются как на алгоритм LZ77.

Особенности кодирования литеральных символов и пар расстояние/длина

В предыдущих разделах ничего не было сказано о небольшом нюансе реализации алгоритма: как в процессе считывания сжатых данных отличить литеральный символ от кода расстояние/длина? В конце концов, не существует никакого принципиального различия между литеральным символом и первым байтом кода пары значений расстояние/длина. Одно возможное решение – вывод одиночного бита флага перед литеральным символом или кодом расстояние/длина. Если бит флага является нулевым, следующий считываемый код будет литеральным символом. Если флаг является единичным, следующий считываемый код будет парой расстояние/длина. Однако применение этого метода привело бы к необходимости вывода одиночных битов, сводя на нет преимущество использования одних только байтов.

Общий способ избавления от этого недостатка состоит в применении флага, состоящего из восьми битов, указывающих, чем должны быть следующие восемь кодов. При этом первый бит определяет, чем тип первого кода, следующего за байтом флага, второй бит – второго кода, и так далее для 8 битов и кодов. Затем будет выводиться следующий байт флага. Используя эту схему, можно записывать (и считывать) сжатый поток в виде последовательности байтов.

Аналогичная схема использовалась в программе **EXPAND.EXE** компании Microsoft, которая применялась в составе MS-DOS и Windows 3.1 (в современных

программных продуктах компании Microsoft вместо нее применяются CAB-файлы). Возможно, читатели помнят, что часто файлы на дискетах DOS имели имена наподобие **FILENAME.EX_**, и программа **EXPAND.EXE** должна была их распаковывать и подставлять последний символ в расширении восстановленного файла. В версии алгоритма LZ77, применявшейся компанией Microsoft, коды пар значений расстояние/длина всегда имели размер, равный 2 байтам. При этом 12 бит использовались для указания значения расстояния (в действительности в этой версии использовалась циклическая очередь байтов, и значение расстояния представляло собой величину смещения от начала очереди), а остальные 4 бита служили для определения значения длины.

После того, как мы ознакомились с теорией, пора подумать о реализации и сформулировать ряд правил. Мы будем считать, что размер кода пары расстояние/длина будет всегда равен 2 байтам – длине одного слова – причем старшие 13 бит будут использоваться для указания значения расстояния, а 3 младших бита – для определения значения длины. Поскольку для указания значения расстояния используются 13 бит, теоретически можно закодировать расстояния от 0 до 8191 байта. Следовательно, размер скользящего окна составит 8 Кб. Обратите внимание, что при определении расстояния мы никогда не будем использовать значение, равное 0 (в противном случае соответствие устанавливалось бы с текущей позицией). Таким образом, эти 13 бит будут интерпретироваться как значения от 1 до 8192, а не от 0 до 8191, что будет достигаться за счет простого добавления единицы.

Теперь рассмотрим значение длины. Теоретически, тремя битами можно закодировать значения только от 0 до 7. Однако вспомним, что в пары значений расстояние/длина будут преобразовываться только совпадающие строки, состоящие из трех и более символов. Поэтому за счет простого добавления 3 целесообразно интерпретировать 3 бита как значения длины от 3 до 10 байтов.

Следовательно, чтобы преобразовать значение расстояния и длины в значение слова, нужно было бы записать определение, подобное следующему:

```
Code := ((Distance-1) shl 3)
      + (Length-3);
```

А для восстановления значений расстояния и длины потребовалось бы использовать следующий код:

```
Length := (Code and $7)+3;
Distance := (Code shr 3)+1;
```

Восстановление с применением алгоритма LZ77

Прежде чем приступить к рассмотрению сжатия данных, реализуем алгоритм восстановления, поскольку его концепция проще для визуализации. В процессе восстановления мы считываем байт флага, а затем используем его для определения способа считывания из потока следующих восьми кодов. Если текущий бит в

байте флага является нулевым, мы считываем из потока 1 байт и интерпретируем его как литеральный символ, который должен быть записан непосредственно в выходной поток. И напротив, если текущий бит является единичным, мы считываем из входного потока 2 байта и разбиваем это значение на значения расстояния и длины. Затем эти значения используются с текущим скользящим окном ранее декодированных данных для определения того, какой символ должен быть записан в выходной поток.

При каждом декодировании отдельного символа или набора от трех до 10 символов, их нужно не только записать в выходной поток, но и добавить в конец буфера скользящего окна и сдвинуть начало скользящего окна на соответствующее расстояние, чтобы его размер не превышал 8192 байтов. Естественно, нежелательно, чтобы приходилось восстанавливать буфер при каждом декодировании символа или строки символов — это занимало бы слишком много времени. На практике используется циклическая очередь — очередь фиксированного размера, начало и конец которой определяются индексами. Поскольку на этапе сжатия будет использоваться аналогичное скользящее окно (как именно, мы вскоре рассмотрим), целесообразно создать реализацию класса, которая могла бы использоваться в обоих процессах.

Прежде чем приступить к описанию методов восстановления, которые потребуются для этого класса, я хочу описать небольшой прием, используемый методом Deflate программы **PKZIP**. Еще раз взгляните на пример предложения, сжатие которого было выполнено ранее. На одном из этапов описания алгоритма возникла следующая ситуация:

```
-----+
a cat is |a cat is a cat
-----+^
```

и мы вычислили пару значений расстояние/длина **<9,9>**. Однако можно применить небольшую хитрость. Почему мы должны прекращать сопоставление на 9 символах? В действительности можно сопоставить значительно больше символов, выйдя за пределы правой границы скользящего окна и продолжая сопоставление с текущим символом и с символами, расположенными справа от него. Фактически, можно было бы установить соответствие 14 символов, получив при этом код **<9,14>**, в котором значение длины превышает значение расстояния. Все это замечательно и достаточно разумно, но что при этом происходит во время декодирования? В момент декодирования кода **<9,14>** скользящее окно выглядит следующим образом:

```
-----+
a cat is |
-----+^
```

Мы возвращаемся в скользящем окне на 9 символов назад и начинаем по одному копировать символы, пока не будет достигнут 14-й символ. В результате мы копируем символы, которые нам удалось определить в одной и той же операции.

После копирования девяти символов мы получаем

```
-----+
a cat is |a cat is
-----+^      ^
          от      до
```

где показаны позиции, от которой и до которой выполняется копирование. Как видите, копирование остальных пяти символов может быть выполнено вообще без возникновения каких-либо проблем. Следовательно, значение длины вполне может превышать значение расстояния (хотя приходится признать, что для копирования данных нельзя было просто воспользоваться процедурой **Move**).

Во время восстановления мы передадим класс скользящего окна выходному потоку, в который должны записываться данные. В результате, когда объект определяет, что активные данные в буфере требуется передвинуть обратно к началу, вначале он копирует в поток данные, которые должны замещаться в буфере. Для выполнения восстановления требуются два основных метода: добавление одиночного символа и преобразование пары расстояние/длина. Обратите внимание, что эти действия выполняются классом скользящего окна, поскольку обновление скользящего окна и перемещение вперед по данным должны выполняться в обоих случаях. Кроме того, класс – лучший агент выполнения преобразования значений расстояния и длины. Код реализации интерфейса класса, служебных процедур и вывода соответствующего кода приведен в листинге 11.23.

Листинг 11.23. Код, связанный с выводом, класса скользящего окна

```
type
  TtdLZSlidingWindow = class
  private
    FBuffer : PAnsiChar; {циклический буфер}
    FBufferEnd : PAnsiChar; {конечная точка буфера}
    FCompressing : boolean; {true=сжатие данных}
    FCurrent : PAnsiChar; {текущий символ}
    FLookAheadEnd : PAnsiChar; {конец упреждающего просмотра}
    FMidPoint : PAnsiChar; {средняя точка буфера}
    FName : TtdNameString; {имя скользящего окна}
    FStart : PAnsiChar; {начало скользящего окна}
    FStartOffset : longint; {смещение потока для FStart}
    FStream : TStream; {базовый поток}
  protected
    procedure swAdvanceAfterAdd(aCount : integer);
    procedure swReadFromStream;
    procedure swSetCapacity(aValue : longint);
    procedure swWriteToStream(aFinalBlock : boolean);
  public
    constructor Create(aStream : TStream; aCompressing : boolean);
    destructor Destroy; override;
    {методы, используемые как во время сжатия, так и во время восстановления}
    procedure Clear;
```



```

    {методы, используемые во время сжатия}
    procedure Advance(aCount : integer);
    function Compare(aOffset : longint; var aDistance : integer) : integer;
    procedure GetNextSignature(var aMS : TtdLZSignature; var aOffset : longint);
    {методы, используемые во время восстановления}
    procedure AddChar(aCh : AnsiChar);
    procedure AddCode(aDistance : integer; aLength : integer);
    property Name : TtdNameString read FName write FName;
end;

constructor TtdLZSlidingWindow.Create(aStream : TStream;
                                     aCompressing : boolean);

begin
    inherited Create;
    {сохранить параметры}
    FCompressing := aCompressing;
    FStream := aStream;
    {установить размер скользящего окна: согласно принятого определения размер
    скользящего окна равен 8192 байтам плюс 10 байтов для упрещающего
    просмотра}
    swSetCapacity(tdcLZSlidingWindowSize + tdcLZLookAheadSize);
    {сбросить буфер и, если выполняется сжатие, считать определенные данные из
    сжимаемого потока}
    Clear;
    if aCompressing then
        swReadFromStream;
end;

destructor TtdLZSlidingWindow.Destroy;
begin
    if Assigned(FBuffer) then begin
        {завершить запись в выходной поток, если выполняется сжатие}
        if not FCompressing then
            swWriteToStream(true);
        {освободить буфер}
        FreeMem(FBuffer, FBufferEnd - FBuffer);
    end;
    inherited Destroy;
end;

procedure TtdLZSlidingWindow.AddChar(aCh : AnsiChar);
begin
    {добавить символ в буфер}
    FCurrent^ := aCh;
    {сместить скользящее окно на один символ}
    swAdvanceAfterAdd(1);
end;

```



```
procedure TtdLZSlidingWindow.AddCode(aDistance : integer; aLength : integer);  
var  
    FromChar : PAnsiChar;  
    ToChar : PAnsiChar;  
    i : integer;  
begin  
    { установить указатели для выполнения копирования данных; обратите внимание,  
      что в данном случае нельзя использовать процедуру Move, поскольку часть  
      копируемых данных может быть определена реальным копированием данных }  
    FromChar := FCurrent - aDistance;  
    ToChar := FCurrent;  
    for i := 1 to aLength do begin  
        ToChar^ := FromChar^;  
        inc(FromChar);  
        inc(ToChar);  
    end;  
    { сдвинуть начало скользящего окна }  
    swAdvanceAfterAdd(aLength);  
end;  
  
procedure TtdLZSlidingWindow.swAdvanceAfterAdd(aCount : integer);  
begin  
    { при необходимости сдвинуть начало скользящего окна }  
    if ((FCurrent - FStart) >= tdcLZSlidingWindowSize) then begin  
        inc(FStart, aCount);  
        inc(FStartOffset, aCount);  
    end;  
    { сдвинуть текущий указатель }  
    inc(FCurrent, aCount);  
    { проверить смещение в зону переполнения }  
    if (FStart >= FMidPoint) then begin  
        { записать дополнительные данные в поток (от позиции FBuffer до позиции  
          FStart) }  
        swWriteToStream(false);  
        { переместить текущие данные обратно в начало буфера }  
        Move(FStart^, FBuffer^, FCurrent - FStart);  
        { сбросить различные указатели }  
        dec(FCurrent, FStart - FBuffer);  
        FStart := FBuffer;  
    end;  
end;  
  
procedure TtdLZSlidingWindow.swSetCapacity(aValue : longint);  
var  
    NewQueue : PAnsiChar;  
begin  
    { округлить запрошенный объем до ближайшего значения, кратного 64 байтам }  
    aValue := (aValue + 63) and $7FFFFFFC0;  
    { распределить новый буфер }  
    GetMem(NewQueue, aValue * 2);
```

```

{уничтожить старый буфер}
if (FBuffer <> nil) then
    FreeMem(FBuffer, FBufferEnd - FBuffer);
{установить начальный/конечный и другие указатели}
FBuffer := NewQueue;
FStart := NewQueue;
FCurrent := NewQueue;
FLookAheadEnd := NewQueue;
FBufferEnd := NewQueue + (aValue * 2);
FMidPoint := NewQueue + aValue;
end;

procedure TtdLZSlidingWindow.swWriteToStream(aFinalBlock : boolean);
var
    BytesToWrite : longint;
begin
    {записать данные перед текущим скользящим окном}
    if aFinalBlock then
        BytesToWrite := FCurrent - Fbuffer
    else
        BytesToWrite := FStart - FBuffer;
    FStream.WriteBuffer(FBuffer^, BytesToWrite);
end;

```

Метод **AddChar** добавляет одиночный литеральный символ в скользящее окно и сдвигает это окно вперед на один байт. Внутренний метод **swAdvanceAfterAdd** выполняет реальный сдвиг и после сдвига окна проверяет, может ли еще один блок быть записан в выходной поток. Метод **AddCode** добавляет пару расстояние/длина в скользящее окно, по одному копируя символы из уже декодированной части скользящего окна в текущую позицию. Затем скользящее окно сдвигается вперед.

Теперь достаточно легко создать код восстановления. (Создавать код восстановления раньше кода сжатия кажется несколько неестественным, но в действительности формат сжатых данных настолько определен, что это можно сделать. Кроме того, это проще!) Мы реализуем основной цикл в виде машины состояний с тремя состояниями: считывание и обработка байта флага, считывание и обработка символа и, наконец, считывание и обработка кода расстояния/длины. Код показан в листинге 11.24. Обратите внимание, что определение момента завершения восстановления осуществляется по количеству байтов в несжатом потоке, записанному программой сжатия в начало сжатого потока.

После проверки того, что входной поток является закодированным с применением алгоритма LZ77, программа считывает количество несжатых данных. Затем осуществляется вход в простую машину состояний, состояния которой определяются байтом флага, считываемого из входного потока. Если текущее состояние – **dsGetFlagByte**, программа считывает из входного потока следующий байт флага. Если состояние – **dsGetChar**, программа считывает из входного потока литераль-

ный символ и добавляет его в скользящее окно. В противном случае состоянием будет **dsGetDistLen**, и программа считывает из входного потока пару расстояние/длина и добавляет ее в скользящее окно. Этот процесс продолжается до тех пор, пока не будут восстановлены все данные входного потока.

Листинг 11.24. Основной код программы сжатия, использующей алгоритм LZ77

```
procedure TDLZDecompress(aInStream, aOutStream : TStream);
type
  TDecodeState = (dsGetFlagByte, dsGetChar, dsGetDistLen);
var
  SlideWin : TtdLZSlidingWindow;
  BytesUnpacked : longint;
  TotalSize : longint;
  LongValue : longint;
  DecodeState : TDecodeState;
  FlagByte : byte;
  FlagMask : byte;
  NextChar : AnsiChar;
  NextDistLen : longint;
  CodeCount : integer;
  Len : integer;
begin
  SlideWin := TtdLZSlidingWindow.Create(aOutStream, false);
  try
    SlideWin.Name := 'LZ77 Decompress sliding window';
    {считать из потока заголовок: символы 'TDLZ', за которыми следует размер
      входного потока}
    aInStream.ReadBuffer(LongValue, sizeof(LongValue));
    if (LongValue <> TDLZHeader) then
      RaiseError(tdeLZBadEncodedStream, 'TDLZDecompress');
    aInStream.ReadBuffer(TotalSize, sizeof(TotalSize));
    {подготовиться к восстановлению}
    BytesUnpacked := 0;
    NextDistLen := 0;
    DecodeState := dsGetFlagByte;
    CodeCount := 0;
    FlagMask := 1;
    {до тех пор, пока остаются байты для восстановления...}
    while (BytesUnpacked < TotalSize) do begin
      {считывать следующий элемент в данном состоянии декодирования}
      case DecodeState of
        dsGetFlagByte :
          begin
            aInStream.ReadBuffer(FlagByte, 1);
            CodeCount := 0;
            FlagMask := 1;
          end;
```

```

dsGetChar :
  begin
    aInStream.ReadBuffer(NextChar, 1);
    SlideWin.AddChar(NextChar);
    inc(BytesUnpacked);
  end;
dsGetDistLen :
  begin
    aInStream.ReadBuffer(NextDistLen, 2);
    Len := (NextDistLen and tdcLZLengthMask) + 3;
    SlideWin.AddCode((NextDistLen shr tdcLZDistanceShift) + 1, Len);
    inc(BytesUnpacked, Len);
  end;
end;
{ вычислить следующее состояние декодирования }
inc(CodeCount);
if (CodeCount > 8) then
  DecodeState := dsGetFlagByte
else begin
  if ((FlagByte and FlagMask) = 0) then
    DecodeState := dsGetChar
  else
    DecodeState := dsGetDistLen;
    FlagMask := FlagMask shl 1;
  end;
end;
finally
  SlideWin.Free;
end; { try..finally }
end;

```

Сжатие LZ77

Теперь пора рассмотреть реализацию сжатия. При этом очень быстро мы сталкиваемся со следующей проблемой: поиском наиболее длинного соответствия между строкой в текущей позиции и предшествующими 8192 байтами. От одного возможного метода – поиска во всем буфере – придется полностью отказаться из-за его слишком низкой эффективности.

В своей первоначальной работе Зив и Лемпель не предложили почти никаких решений. Кое-кто использует дерево бинарного поиска, построенное поверх скользящего окна, для хранения максимальных встречавшихся совпадающих строк (примером может служить реализация, созданная Марком Нельсоном (Mark Nelson) [15]). Однако применение этого метода приводит к возникновению проблем, связанных с тем, что нужно беспокоиться о балансировке дерева и об избавлении от строк, которые должны быть удалены из скользящего окна. Поэтому мы воспользуемся советом, приведенным в онлайн-овом документе Deflate Compressed

Data Format Specification (Спецификация формата данных, сжатых методом Deflate) (RFC1951) и применим хеш-таблицу.

Алгоритм выглядит следующим образом: мы просматриваем три символа в текущей позиции – будем называть их *сигнатурой* (*signature*). Сигнатура хешируется с применением одного из методов, а затем хеш-значение используется для доступа к элементу в хеш-таблице, использующему связывание. Цепочки или связные списки в каждом элементе хеш-таблицы будут состоять из последовательностей элементов, каждый из которых состоит из трехсимвольных сигнатур и значения смещения сигнатуры во входном потоке.

Итак, мы получаем сигнатуру текущей позиции и хешируем ее в связный список, представляющий собой одну из цепочек в хеш-таблице. Затем мы просматриваем связный список и сравниваем хранящуюся в нем сигнатуру каждого элемента с имеющимися сигнатурами. При обнаружении совпадающей сигнатуры мы переходим в скользящее окно, используя значение смещения элемента, а затем сравниваем символы в скользящем окне с символами в текущей позиции. Мы повторяем этот процесс для каждого элемента в связном списке, совпадающего с данной сигнатурой, и запоминаем наиболее длинное найденное соответствие.

После выполнения этого поиска, независимо от того, был ли он успешным или безуспешным, текущую сигнатуру нужно добавить в хеш-таблицу, чтобы ее можно было найти при сравнении последующими сигнатурами. Сигнатура добавляется в начало связного списка, чем обеспечивается сортировка связного списка в порядке убывания значений смещения.

Однако если не принять никаких специальных мер, количество элементов в хеш-таблице будет только возрастать, в то время как в действительности нам не нужны элементы, которые больше не присутствуют в 8-Кбайтном скользящем окне. Первым возможным решением этой проблемы может быть удаление элемента, который при следующем сдвиге скользящего окна должен исчезнуть из него. Для этого нужно было бы найти сигнатуру позиции (или, точнее говоря, позиций), которая должна исчезнуть из начала скользящего окна, хешировать ее, пройти по связному списку, хранящемуся в этой позиции хеш-таблицы, с целью отыскания соответствующего элемента, а затем удалить его.

Более эффективный способ, однако, влекущий за собой увеличение количества элементов в хеш-таблице – сокращение связного списка во время поиска в нем текущей сигнатуры. Вспомните, что связные списки сортируются в порядке убывания значений смещения. Если при перемещении по связному списку в ходе поиска максимального соответствия текущей позиции будет обнаружен элемент, значение смещения которого указывает, что он больше не присутствует в скользящем окне, этот элемент и все последующие элементы данного связного списка должны быть удалены. При использовании этого метода удаление старых элементов откладывается до момента выполнения подпрограммы поиска в связном списке – фактически, до момента, когда мы оказываемся в середине связного списка. Это означает,

что хеш-таблица содержит больше элементов, нежели нужно, но этот недостаток весьма незначителен по сравнению с преимуществом ускорения работы алгоритма.

Естественно, необходимо выбрать функцию хеширования. Вместо того чтобы использовать одну из подпрограмм хеширования общего назначения, описанных в главе 7, мы воспользуемся тем фактом, что сигнатуры содержат три символа. В качестве сигнатуры мы определим три младших байта значения типа длинного целого (**longint**), старший байт которого является нулевым. В результате мы получаем хеш-значение, не требующее никаких вычислений. Как обычно, размер хеш-таблицы должен быть равен простому числу. Я выбрал значение 521 – наименьшее простое число, превышающее 512. Это означает, что в среднем 16 сигнатур из 8-Кбайтного скользящего окна будут преобразовываться в один и тот же номер элемента, образуя для просмотра во время поиска связный список приемлемого размера.

Для восстановления LZ77 целесообразно создать специальный класс хеш-таблицы, поскольку в ней должен выполняться ряд специализированных операций. Код этого класса хеш-таблицы приведен в листинге 11.25.

Листинг 11.25. Хеш-таблица LZ77

```

type
  TtdLZSigEnumProc = procedure (aExtraData : pointer;
                                const aSignature : TtdLZSignature;
                                aOffset : longint);

  PtdLZHashNode = ^TtdLZHashNode;
  TtdLZHashNode = packed record
    hnNext : PtdLZHashNode;
    hnSig : TtdLZSignature;
    hnOffset : longint;
  end;

type
  TtdLZHashTable = class
    private
      FHashTable : TList;
      FName : TtdNameString;
    protected
      procedure htError(aErrorCode : integer; const aMethodName : TtdNameString);
      procedure htFreeChain(aParentNode : PtdLZHashNode);
    public
      constructor Create;
      destructor Destroy; override;
      procedure Empty;
      function EnumMatches(const aSignature : TtdLZSignature;
                        aCutOffset : longint;
                        aAction : TtdLZSigEnumProc;
                        aExtraData : pointer) : boolean;
      procedure Insert(const aSignature : TtdLZSignature; aOffset : longint);
      property Name : TtdNameString read FName write FName;
  end;

```



```

constructor TtdLZHashTable.Create;
var
    Inx : integer;
begin
    inherited Create;
    if (LZHashNodeManager = nil) then begin
        LZHashNodeManager := TtdNodeManager.Create(sizeof(TtdLZHashNode));
        LZHashNodeManager.Name := 'LZ77 node manager';
    end;
    {создать хеш-таблицу, преобразовать все элементы в связанные списки с
    фиктивным заглавным узлом}
    FHashTable := TList.Create;
    FHashTable.Count := LZHashTableSize;
    for Inx := 0 to pred(LZHashTableSize) do
        FHashTable.List^[Inx] := LZHashNodeManager.AllocNodeClear;
end;

destructor TtdLZHashTable.Destroy;
var
    Inx : integer;
begin
    {полностью уничтожить хеш-таблицу, включая фиктивные заглавные узлы}
    if (FHashTable <> nil) then begin
        Empty;
        for Inx := 0 to pred(FHashTable.Count) do
            LZHashNodeManager.FreeNode(FHashTable.List^[Inx]);
        FHashTable.Free;
    end;
    inherited Destroy;
end;

procedure TtdLZHashTable.Empty;
var
    Inx : integer;
begin
    for Inx := 0 to pred(FHashTable.Count) do
        htFreeChain(PtdLZHashNode(FHashTable.List^[Inx]));
end;

function TtdLZHashTable.EnumMatches(const aSignature : TtdLZSignature;
                                     aCutOffset : longint;
                                     aAction : TtdLZSigEnumProc;
                                     aExtraData : pointer) : boolean;

var
    Inx : integer;
    Temp : PtdLZHashNode;
    Dad : PtdLZHashNode;
begin
    {предположим, что ни один элемент не найден}
    Result := false;

```

```

{вычислить индекс хеш-таблицы для этой сигнатуры}
Inx := (aSignature.AsLong shr 8) mod LZHashTableSize;
{выполнить обход цепочки, расположенной в позиции с этим индексом}
Dad := PtdLZHashNode(FHashTable.List^[Inx]);
Temp := Dad^.hnNext;
while (Temp <> nil) do begin
    {если смещение этого узла меньше значения смещения, по которому выполняется
    усеечение, оставшая часть цепочки удаляется, и выполняется выход из
    подпрограммы}
    if (Temp^.hnOffset < aCutoffset) then begin
        htFreeChain(Dad);
        Exit;
    end;
    {если сигнатура узла совпадает с данной сигнатурой, выполняется вызов
    подпрограммы, выполняющей действие}
    if (Temp^.hnSig.AsLong = aSignature.AsLong) then begin
        Result := true;
        aAction(aExtraData, aSignature, Temp^.hnOffset);
    end;
    {перейти к следующему узлу}
    Dad := Temp;
    Temp := Dad^.hnNext;
end;
end;

procedure TtdLZHashTable.htFreeChain(aParentNode : PtdLZHashNode);
var
    Walker, Temp : PtdLZHashNode;
begin
    Walker := aParentNode^.hnNext;
    aParentNode^.hnNext := nil;
    while (Walker <> nil) do begin
        Temp := Walker;
        Walker := Walker^.hnNext;
        LZHashNodeManager.FreeNode(Temp);
    end;
end;

procedure TtdLZHashTable.Insert(const aSignature : TtdLZSignature;
                                aOffset : longint);
var
    Inx : integer;
    NewNode : PtdLZHashNode;
    HeadNode : PtdLZHashNode;
begin
    {вычислить индекс хеш-таблицы для этой сигнатуры}
    Inx := (aSignature.AsLong shr 8) mod LZHashTableSize;
    {распределить новый узел и вставить его в начало цепочки, расположенной в
    позиции хеш-таблицы, определяемой этим индексом; тем самым обеспечивается
    упорядочение узлов в цепочке в порядке убывания значений смещения}

```

```

NewNode := LZHashNodeManager.AllocNode;
NewNode^.hnSig := aSignature;
NewNode^.hnOffset := aOffset;
HeadNode := PtdLZHashNode(FHashTable.List^[Inx]);
NewNode^.hnNext := HeadNode^.hnNext;
HeadNode^.hnNext := NewNode;
end;

```

В целях повышения эффективности в хеш-таблице используется диспетчер узлов, поскольку придется распределить и освободить несколько тысяч узлов. Это выполняется внутри конструктора **Create**. Через непродолжительное время метод **EnumMatches** вызывается снова. Он просматривает все элементы в хеш-таблице на предмет совпадения с конкретной сигнатурой и для каждого найденного такого элемента вызывает процедуру **aAction**. Так реализуется основная логика установления соответствия алгоритма LZ77.

Класс скользящего окна выполняет также ряд важных функций, кроме сохранения ранее встречавшихся байтов. Во-первых, во время кодирования скользящее окно считывает данные из входного потока большими боками, чтобы об этом не нужно было беспокоиться во время выполнения подпрограммы сжатия. Во-вторых, оно возвращает текущую сигнатуру и ее смещение во входном потоке. Третий метод выполняет сопоставление: он принимает смещение во входном потоке, преобразует его в смещение в буфере скользящего окна, а затем сравнивает хранящиеся там символы с символами в текущей позиции. Метод будет возвращать количество совпадающих символов и значение расстояния, что позволяет создать пару расстояние/длина. Заключительный фрагмент кода реализации этого класса скользящего окна приведен в листинге 11.26 (код остальных методов можно найти в листинге 11.23).

Листинг 11.26. Методы скользящего окна, используемые во время сжатия

```

procedure TtdLZSlidingWindow.Advance(aCount : integer);
var
  ByteCount : integer;
begin
  {при необходимости сдвинуть начало скользящего окна}
  if ((FCurrent - FStart) >= tdcLZSlidingWindowSize) then begin
    inc(FStart, aCount);
    inc(FStartOffset, aCount);
  end;
  {сдвинуть текущий указатель}
  inc(FCurrent, aCount);
  {проверить смещение в зону переполнения}
  if (FStart >= FMidPoint) then begin
    {переместить текущие данные обратно в начало буфера}
    ByteCount := FLookAheadEnd - FStart;
    Move(FStart^, FBuffer^, ByteCount);
  end;
end;

```

```

    { сбросить различные указатели }
    ByteCount := FStart - FBuffer;
    FStart := FBuffer;
    dec(FCurrent, ByteCount);
    dec(FLookAheadEnd, ByteCount);
    { выполнить считывание дополнительных данных из потока }
    swReadFromStream;
end;
end;

function TtdLZSlidingWindow.Compare(aOffset : longint;
                                     var aDistance : integer) : integer;
var
    MatchStr : PAnsiChar;
    CurrentCh : PAnsiChar;
begin
    { Примечание: когда эта подпрограмма вызывается, она предполагает, что между
      переданной и текущей позицией будет найдено по меньшей мере три совпадающих
      символа }
    { вычислить позицию в скользящем окне, соответствующую переданному смещению и
      ее расстоянию от текущей позиции }
    MatchStr := FStart + (aOffset - FStartOffset);
    aDistance := FCurrent - MatchStr;
    inc(MatchStr, 3);
    { вычислить длину строки совпадающих символов между данной и текущей
      позицией. Эта длина не должна превышать максимальной длины. Для конца
      входного потока определен специальный случай }
    Result := 3;
    CurrentCh := FCurrent + 3;
    if (CurrentCh <> FLookAheadEnd) then begin
        while (Result < tdcLZMaxMatchLength) and
              (MatchStr^ = CurrentCh^) do begin
            inc(Result);
            inc(MatchStr);
            inc(CurrentCh);
            if (CurrentCh = FLookAheadEnd) then
                Break;
        end;
    end;
end;

procedure TtdLZSlidingWindow.GetNextSignature(var aMS : TtdLZSignature;
                                              var aOffset : longint);
var
    P : PAnsiChar;
    i : integer;
begin
    { вычислить длину совпадающей строки; обычно она равна 3, но в конце входного
      потока она может быть равна 2 или менее. }

```

```

if ((FLookAheadEnd - FCurrent) < 3) then
  aMS.AsString[0] := AnsiChar(FLookAheadEnd - FCurrent)
else
  aMS.AsString[0] := #3;
P := FCurrent;
for i := 1 to length(aMS.AsString) do begin
  aMS.AsString[i] := P^;
  inc(P);
end;
aOffset := FStartOffset + (FCurrent - FStart);
end;

procedure TtdLZSlidingWindow.swReadFromStream;
var
  BytesRead : longint;
  BytesToRead : longint;
begin
  {выполнить считывание дополнительных данных в зону упреждающего просмотра}
  BytesToRead := FBufferEnd - FLookAheadEnd;
  BytesRead := FStream.Read(FLookAheadEnd^, BytesToRead);
  inc(FLookAheadEnd, BytesRead);
end;

```

Теперь, когда наш арсенал пополнился этими классами, можно создать подпрограмму сжатия, показанную в листинге 11.27. Она слегка осложняется необходимостью накапливать коды сжатия по восемь. Это делается для того, чтобы можно было вычислить байт флага для всех восьми байтов, а затем записать байт флага, за которым следуют восемь кодов. Именно этой цели служит массив **Encodings**. Однако, поскольку мы рассмотрели достаточно много вспомогательных подпрограмм, сама эта подпрограмма не слишком сложна для понимания.

Листинг 11.27. Подпрограмма ZL77

```

type
  PEnumExtraData = ^TEnumExtraData;
  TEnumExtraData = packed record
    edSW : TtdLZSlidingWindow;
    edMaxLen : integer;
    edDistMaxMatch : integer;
  end;

type
  TEncoding = packed record
    AsDistLen : cardinal;
    AsChar : AnsiChar;
    IsChar : boolean;
    {$IFDEF Delphi1}
    Filler : word;
    {$ENDIF}
  end;

```

```

TEncodingArray = packed record
  eaData : array [0..7] of TEncoding;
  eaCount: integer;
end;

procedure MatchLongest(aExtraData : pointer;
                      const aSignature : TtdLZSignature;
                      aOffset : longint); far;

var
  Len : integer;
  Dist : integer;
begin
  with PEnumExtraData(aExtraData)^ do begin
    Len := edSW.Compare(aOffset, Dist);
    if (Len > edMaxLen) then begin
      edMaxLen := Len;
      edDistMaxMatch := Dist;
    end;
  end;
end;

procedure WriteEncodings(aStream : TStream; var aEncodings : TEncodingArray);
var
  i : integer;
  FlagByte : byte;
  Mask : byte;
begin
  {построить байт флага и записать его в поток}
  FlagByte := 0;
  Mask := 1;
  for i := 0 to pred(aEncodings.eaCount) do begin
    if not aEncodings.eaData[i].IsChar then
      FlagByte := FlagByte or Mask;
      Mask := Mask shl 1;
    end;
  aStream.WriteBuffer(FlagByte, sizeof(FlagByte));
  {записать коды}
  for i := 0 to pred(aEncodings.eaCount) do begin
    if aEncodings.eaData[i].IsChar then
      aStream.WriteBuffer(aEncodings.eaData[i].AsChar, 1)
    else
      aStream.WriteBuffer(aEncodings.eaData[i].AsDistLen, 2);
    end;
  aEncodings.eaCount := 0;
end;

procedure AddCharToEncodings(aStream : TStream;
                             aCh : AnsiChar;
                             var aEncodings : TEncodingArray);

```



```

begin
  with aEncodings do begin
    eaData[eaCount].AsChar := aCh;
    eaData[eaCount].IsChar := true;
    inc(eaCount);
    if (eaCount = 8) then
      WriteEncodings(aStream, aEncodings);
  end;
end;

procedure AddCodeToEncodings(aStream : TStream;
                             aDistance : integer;
                             aLength : integer;
                             var aEncodings : TEncodingArray);

begin
  with aEncodings do begin
    eaData[eaCount].AsDistLen :=
      (pred(aDistance) shl tdcLZDistanceShift) + (aLength - 3);
    eaData[eaCount].IsChar := false;
    inc(eaCount);
    if (eaCount = 8) then
      WriteEncodings(aStream, aEncodings);
  end;
end;

procedure TDLZCompress(aInStream, aOutStream : TStream);
var
  HashTable : TtdLZHashTable;
  SlideWin : TtdLZSlidingWindow;
  Signature : TtdLZSignature;
  Offset : longint;
  Encodings : TEncodingArray;
  EnumData : TEnumExtraData;
  LongValue : longint;
  i : integer;
begin
  HashTable := nil;
  SlideWin := nil;
  try
    HashTable := TtdLZHashTable.Create;
    HashTable.Name := 'LZ77 Compression hash table';
    SlideWin := TtdLZSlidingWindow.Create(aInStream, true);
    SlideWin.Name := 'LZ77 Compression sliding window';
    {записать заголовок в поток: 'TDLZ', за которым следует размер несжатого
     исходного потока}
    LongValue := TDLZHeader;
    aOutStream.WriteBuffer(LongValue, sizeof(LongValue));
    LongValue := aInStream.Size;
    aOutStream.WriteBuffer(LongValue, sizeof(LongValue));
    {подготовка к сжатию}
    Encodings.eaCount := 0;
  except
  end;
end;

```

```

EnumData.edSW := SlideWin;
{получить первую сигнатуру}
SlideWin.GetNextSignature(Signature, Offset);
{до тех пор, пока длина сигнатуры равна трем символам...}
while (length(Signature.AsString) = 3) do begin
    {выполнить поиск в скользящем окне самой длинной совпадающей строки с
    использованием хеш-таблицы для идентификации соответствий}
    EnumData.edMaxLen := 0;
    if HashTable.EnumMatches(Signature,
                            Offset - tdcLZSlidingWindowSize,
                            MatchLongest,
                            @EnumData) then begin
        {имеется по меньшей мере одно соответствие: необходимо сохранить пару
        расстояние/длина самой длинной совпадающей строки и сдвинуть
        скользящее окно на расстояние, равное этой длине}
        AddCodeToEncodings(aOutputStream,
                           EnumData.edDistMaxMatch,
                           EnumData.edMaxLen,
                           Encodings);
        SlideWin.Advance(EnumData.edMaxLen);
    end
    else begin
        {соответствие отсутствует: необходимо сохранить текущий символ и
        сдвинуть скользящее окно на один символ}
        AddCharToEncodings(aOutputStream,
                           Signature.AsString[1],
                           Encodings);
        SlideWin.Advance(1);
    end;
    {добавить эту сигнатуру в хеш-таблицу}
    HashTable.Insert(Signature, Offset);
    {извлечь следующую сигнатуру}
    SlideWin.GetNextSignature(Signature, Offset);
end;
{если последняя сигнатура содержала не более двух символов, их нужно
сохранить как коды литеральных символов}
if (length(Signature.AsString) > 0) then begin
    for i := 1 to length(Signature.AsString) do
        AddCharToEncodings(aOutputStream,
                           Signature.AsString[i],
                           Encodings);
end;
{обеспечить запись заключительных кодов}
if (Encodings.eaCount > 0) then
    WriteEncodings(aOutputStream, Encodings);
finally
    SlideWin.Free;
    HashTable.Free;
end; {try..finally}
end;

```

Подпрограмма сжатия работает следующим образом. Мы создаем хеш-таблицу и скользящее окно. После этого мы записываем в выходной поток сигнатуру, за которой следует значение длины несжатых данных. Затем осуществляется вход в цикл. После каждого выполнения цикла мы получаем текущую сигнатуру и пытаемся сопоставить ее с чем-либо уже встречавшимся ранее (для этого используется метод **EnumMatches** хеш-таблицы). Если какие-либо соответствия отсутствуют, литеральный символ добавляется в массив кодов и скользящее окно сдвигается на один символ. В противном случае в скользящее окно добавляется пара расстояние/длина, соответствующая наиболее длинной совпадающей строке, и скользящее окно сдвигается на расстояние, равное количеству совпадающих символов.

Код программы сжатия LZ77 разбит на несколько файлов: **TDLZBase.pas** содержит несколько общих констант, **TDLZHash.pas** создает специализированную хеш-таблицу, **TDLZSWin** – класс скользящего окна, а **TDLZCmpr.pas** – код выполнения сжатия и восстановления. Все перечисленные файлы можно найти на Web-сайте издательства, в разделе материалов.

После того, как мы ознакомились с алгоритмом и кодом реализации сжатия и восстановления LZ77, можно теоретически оценить возможные значения коэффициентов сжатия. Если бы можно было сжать все 10 байтовые строки в файле до 2 байт – иначе говоря, каждый раз получать максимальное соответствие – для каждых 80 байтов файла можно было бы записывать по 17 байт (один байт флага и восемь 2-байтовых кодов). В этом случае коэффициент сжатия равнялся бы 79 процентам. С другой стороны, если бы соответствия в файле вообще не удалось бы найти, для каждых восьми байтов исходного файла в действительности пришлось бы записывать по девять байтов. В этом случае коэффициент сжатия составил бы -13 процентов. В общем случае, как правило, сжатие файлов с применением этого метода позволяет получать коэффициенты сжатия, лежащие между упомянутыми крайними значениями.

Резюме

В этой главе мы провели исследования методов сжатия данных. Мы начали рассмотрение с двух статических алгоритмов кодирования с минимальной избыточностью: кодирования Шеннона-Фано и кодирования Хаффмана. Мы рассмотрели недостатки этих методов – необходимость двукратного считывания входных данных и какого-либо кодирования дерева, чтобы его можно было поставлять со сжатыми данными. Затем мы ознакомились с адаптивным алгоритмом – сжатия с использованием скошенного дерева – позволяющим устранить обе упомянутых проблемы. И в заключение мы рассмотрели сжатие с применением алгоритма LZ77, в котором используется словарь, позволяющий сжимать строки символов, а не отдельные символы. Хотя все четыре рассмотренных алгоритма представляют интерес и сами по себе, для их реализации мы воспользовались рядом более простых алгоритмов и структур данных, которые были описаны в предшествующих главах.



Дополнительные темы

В этой главе мы отойдем от некоторых стандартных классических алгоритмов и рассмотрим ряд более сложных вопросов. Иногда в этой главе будут использоваться некоторые более простые алгоритмы и структуры данных, но во всех таких случаях они будут служить ступенями к реализации усложненных алгоритмов. Именно так и следует использовать классические алгоритмы и структуры данных – в качестве строительных блоков новых алгоритмов, обеспечивающих реализацию конкретных проектов (в конце концов, проект – это всего лишь эскиз специализированного алгоритма).

Алгоритм считывания-записи

В многопоточных приложениях 32-разрядной операционной системы Windows приходится решать целый ряд проблем, которые в однопоточных программах просто не возникают. Действительно, первая проблема, с которой приходится сталкиваться – определение способа запуска и останова потоков. Но в основном она решается на уровне операционной системы: достаточно внимательно прочесть программную документацию операционной системы и правильно применить почерпнутые сведения.

Этот раздел адресован только тем программистам, которые работают в среде 32-разрядной Windows. Delphi I вообще не поддерживает многопоточную обработку, в то время как Kylix и Linux не предоставляют необходимых примитивных объектов синхронизации, с помощью которых можно было бы решить проблему считывания-записи.

Более серьезная проблема – совместное использование данных несколькими потоками, независимо от того, являются ли данные отдельным целочисленным значением или более сложной структурой данных. По существу, приходится решать вопросы параллельного доступа. Если конкретный поток обновляет часть данных, считывание этих данных в это же время другим потоком лишено смысла. В этом случае считывающий поток (обычно называемый *программой считывания (reader)*) может получить частично обновленное значение, поскольку обновляющий поток (*программа записи (writer)*) еще не закончил обновление, но операционная система отключилась от него.

При наличии двух или более программ записи достаточно скоро могут возникнуть значительные проблемы, если эти программы обновляют одни и те же данные. Однако никакие проблемы параллельного доступа не должны возникать в случае считывания одних и тех же данных несколькими программами считывания.

На момент написания этой книги большинство пользователей использовало однопроцессорные персональные компьютеры (ПК). В таких компьютерах операционная система осуществляет очень быстрое циклическое переключение между потоками, останавливая один поток и запуская другой. Конкретный метод выполнения этого переключения не имеет значения (нецелесообразно создавать программу для конкретной схемы, поскольку она может зависеть от операционной системы), но следует сразу уяснить, что невозможно точно определить все характеристики переключения (такие, как момент переключения, являются ли определенные операции элементарными и т.п.). Один из лучших, когда-либо слышанных мною советов состоял в том, что многопоточные приложения всегда должны быть протестированы на многопроцессорном компьютере. На таком компьютере операционная система будет действительно одновременно выполнять два или более потока. Все неприятные аспекты проблем параллельной обработки неизбежно проявятся при запуске программы на ПК с двумя или более процессорами. Даже если тестовая программа успешно выполняется на однопроцессорном ПК (возможно, потому, что переключение потоков всегда выполняется в удачные моменты времени), на многопроцессорном компьютере код может разрушаться из-за каких-нибудь причудливых ошибок.

Поэтому требуется механизм блокировки. Программа записи должна иметь возможность "блокировать" определенные данные, чтобы во время их обновления никакая другая программа записи или считывания не могла получить к ним доступ. Аналогично, во время считывания данных программой считывания никакая программа записи не должна быть в состоянии их обновить, но другие программы считывания должны по-прежнему иметь к ним доступ.

Похоже, что в среде 32-разрядной Windows существует множество объектов, обеспечивающих синхронизацию: критический раздел, флаг синхронизации, семафор, событие, но ни один из них не подходит для решения поставленной задачи полностью. Критический раздел и флаг синхронизации подходят больше других,

но они не позволят нескольким программам считывания одновременно получать доступ к одним и тем же данным.

Если для работы с многопоточными данными совместного использования применяется класс **TList**, Delphi 3 и последующие версии языка предоставляет класс **TThreadedList**. В основном, применяемая в этом классе стратегия синхронизации реализуется следующим образом: каждое обращение к **TList** защищается критическим разделом или флагом синхронизации. Delphi-версия класса **TThreadedList** предоставляет метод **LockList**, который выполняет вход в критический раздел и возвращает внутренний класс **TList**. Затем поток может свободно использовать этот объект **TList** до момента своего завершения, после чего подпрограмма потока должна вызвать метод **UnlockList** для выхода из критического раздела.

Хотя это решение работает, и притом весьма успешно, ему присущ очевидный недостаток: в любой отдельный момент времени только один поток может иметь доступ к объекту **TList**. Класс не делает никакого различия между доступом для считывания (в процессе которого список не изменяется) и доступом для записи (при котором он изменяется). Как уже отмечалось, в любой отдельный момент времени может существовать много программ, осуществляющих одновременное считывание объекта **TList**. Но может существовать только одна программа, осуществляющая запись в него. Это решение, хотя его и просто реализовать, характеризуется избыточностью. Оно не позволяет с максимальной эффективностью использовать **TList** для многопоточной обработки.

Определим действия, которые должен был бы выполнять объект синхронизации. Нам требуется единый объект, который мог бы использоваться для синхронизации доступа к данным как программой считывания, так и программой записи. Он должен допускать одновременное существование нескольких активных потоков считывания. В любой данный момент времени он должен допускать существование только одного активного потока записи, и, если такой поток существует, не должно существовать ни одного активного потока считывания (они могут обращаться к каким-либо данным в структуре данных, которые в данный момент обновляются).

В идеале необходимо определить также следующее поведение. Если потоку требуется выполнить запись в структуру данных, он должен иметь возможность сообщить об этом объекту. В этом случае объект блокирует запуск любых новых потоков считывания до момента завершения всех текущих потоков считывания. Затем он позволит продолжить выполнение потока записи. Если никакого ожидающего своей очереди потока записи не существует, поток считывания должен получить беспрепятственный доступ к структуре данных. Необходимо каким-то способом обеспечить возможность постановки нескольких потоков записи в очередь. По существу, это требование означает, что объект синхронизации должен принудительно организовывать цикл использования объекта **TList** многими потоками считывания, затем единственным потоком записи, затем многими потоками считывания и т.д.

Из приведенного определения понятно, что должен существовать какой-то примитивный объект синхронизации, которому поток записи мог бы сигнализировать о завершении обновления, чтобы можно было запустить потоки записи. (Под *примитивным* понимается какой-либо объект, предоставляемый самой операционной системой). И наоборот, должен существовать объект синхронизации, которому последний поток в наборе потоков считывания мог бы сигнализировать о завершении своей работы, чтобы можно было предоставить свободу действий потоку записи.

Разрабатываемый нами комплексный объект нуждается, по меньшей мере, в четырех методах. Поток считывания вызывает первый метод, чтобы начать считывание (обратите внимание, что внутри этой подпрограммы может происходить блокировка, обеспечивающая ожидание окончания работы потока записи). Иногда этот метод называют *подпрограммой регистрации считывания* (*reader registration routine*). Как только поток считывания завершает свою работу, он должен вызвать другую подпрограмму для прекращения использования объекта синхронизации и, возможно, предоставления свободы действий потоку записи (подпрограмма отмены регистрации считывания). Аналогично такие же две подпрограммы должны существовать и для потока записи. Назовем эти четыре подпрограммы, соответственно, **StartReading**, **StopReading**, **StartWriting** и **StopWriting**.

Описать возможную работу этого объекта достаточно легко. Сложнее его действительно реализовать. Подпрограмма **StartReading** выполняет несколько задач. Вначале она должна проверить существование ожидающего своей очереди потока записи. При наличии хотя бы одного такого потока, подпрограмма должна перейти в режим ожидания поступления какого-либо объекта синхронизации. Наиболее подходящие кандидаты на роль такого объекта – семафор или событие (эти объекты допускают одновременный запуск нескольких потоков при поступлении сигнала, в то время как флаг синхронизации или критический раздел не допускают этого). Если в данный момент времени поток записи действительно выполняется, подпрограмма **StartReading** должна выполнять блокировку таким же образом. В отсутствии выполняющегося или ожидающего потока записи подпрограмма **StartReading** регистрирует поток как считывающий, осуществляет выход, после чего поток получает возможность немедленно продолжить свою работу.

Метод **StopReading** должен выяснить, выполняется ли в данный момент последний поток считывания. Если это так, метод должен предоставить свободу действий потоку записи, передавая ему ожидаемый им сигнализирующий объект. Если ожидающий поток записи отсутствует, могут существовать ожидающие потоки считывания. Поэтому метод должен оставить объект в таком состоянии, чтобы поток считывания или записи мог быть немедленно запущен при вызове соответствующей запускающей подпрограммы.

Метод **StartWriting** также выполняет несколько задач. Если поток записи активен, он ожидает поступление объекта синхронизации, который будет использо-

ваться для предоставления свободы действий следующему потоку записи. При наличии одного или более активных потоков считывания, метод действует так же. В противном случае он регистрируется как записывающий и выполняет выход, позволяя потоку записи продолжить работу.

Метод **StopWriting** отменяет регистрацию потока, выполняющегося в качестве записывающего, а затем проверяет существование одного или более готовых к запуску потоков считывания. Если такие потоки существуют, метод передает им ожидаемый ими объект синхронизации и завершает свою работу. Если какие-то потоки считывания отсутствуют, метод проверяет наличие ожидающего потока записи. Если такие потоки существуют, метод предоставляет одному из них свободу действий, передавая ему ожидаемый всеми этими потоками объект, а затем прекращает свою работу. Если ни одна из перечисленных ситуаций не имеет места, метод оставляет составной объект в состоянии, позволяющем немедленный запуск потока чтения или записи.

Исходя из приведенного описания, можно сделать ряд выводов. Во-первых, нам требуется переменная для хранения числа ожидающих потоков считывания. Во-вторых, требуется переменная для хранения числа ожидающих потоков записи. В-третьих, нам нужна переменная для хранения числа выполняющихся в текущий момент времени потоков считывания. В-четвертых, нам нужен булев флаг, свидетельствующий о выполнении потока записи. И, наконец, нам требуются определенные примитивные объекты синхронизации, содержащие все перечисленные компоненты.

Поскольку имеется четыре тесно связанных между собой переменных, вызовы для выполнения их считывания и обновления следует поместить внутрь критического раздела или флага синхронизации. Мы будем использовать критический раздел, поскольку эти компоненты эффективнее. Итак, это будет первым объектом синхронизации. Первым шагом выполнения каждого из четырех описанных методов будет запрос критического раздела, последним – его освобождение. Однако вспомните, что методы, которые позволяют запустить поток считывания, могут блокироваться внутри подпрограммы. Если бы этот программный блок оказался между процедурами вызова и освобождения управляющего критического раздела, возникла бы тупиковая ситуация. Поэтому необходимо обеспечить, чтобы блокировка выполнялась снаружи, после того, как критический раздел освобожден.

Поскольку одновременно только один поток записи может быть активным, может показаться целесообразным поместить объект синхронизации, который ставит потоки записи в очередь, также в критический раздел, поскольку этот раздел может принадлежать только одному потоку. Однако на практике проще воспользоваться семафором. Причина этого проста: в действительности не требуется вызов объекта синхронизации, поскольку не существует подходящего места для его освобождения. Действительно, вы убедитесь, что придется дожидаться семафора в од-

ном потоке и освободить его в другом. Такой подход невозможен при использовании критического раздела: поток, обращающийся к критическому разделу, владеет им.

А каким должен быть объект синхронизации для потоков считывания? Больше всего подошли бы семафор или событие сброса вручную. Как и в предыдущем случае, лучше использовать семафор, поскольку применение объекта события привело бы возникновению проблем (при получении сигнала будут освобождаться только ожидающие его прихода потоки; в данной реализации поток может находиться в состоянии, в котором он еще не вызвал подпрограмму **WaitFor**).

Код интерфейса создаваемого нами класса синхронизации **TtdReadWriteSync** приведен в листинге 12.1. Он содержит ряд частных полей, которые будут использоваться в четырех основных методах.

Листинг 12.1. Интерфейс класса **TtdReadWriteSync**

```
type
  TtdReadWriteSync = class
    private
      FActiveReaders : integer;
      FActiveWriter : boolean;
      FBlockedReaders : THandle; {семафор}
      FBlockedWriters : THandle; {семафор}
      FController : TRTLCriticalSection;
      FWaitingReaders : integer;
      FWaitingWriters : integer;
    protected
    public
      constructor Create;
      destructor Destroy; override;

      procedure StartReading;
      procedure StartWriting;
      procedure StopReading;
      procedure StopWriting;
    end;
```

Приватное поле **FBlockedReaders** семафора предназначено для ожидающих потоков считывания, а поле **FBlockedWriters** – для ожидающих потоков записи. Поле **FController** – основной компонент, обеспечивающий последовательный доступ к объектам (к сожалению, применение подобного механизма последовательной обработки необходимо для обеспечения того, чтобы каждый поток получал целостное и неискаженное изображение всего класса).

Код метода **StartReading** приведен в листинге 12.2.

Листинг 12.2. Метод StartReading

```
procedure TtdReadWriteSync.StartReading;  
var  
    HaveToWait : boolean;  
begin  
    {перехватить управление критическим разделом}  
    EnterCriticalSection(FController);  
    {если существует выполняющийся поток записи или хотя бы один ожидающий своей  
    очереди поток записи, метод добавляет себя в качестве ожидающего метода  
    записи, обеспечивая переход в состояние ожидания}  
    if FActiveWriter or (FWaitingWriters <> 0) then begin  
        inc(FWaitingReaders);  
        HaveToWait := true;  
    end  
    {в противном случае он добавляет себя в качестве еще одного выполняющегося  
    потока считывания и обеспечивает отсутствие состояния ожидания}  
    else begin  
        inc(FActiveReaders);  
        HaveToWait := false;  
    end;  
    {освободить управление критическим разделом}  
    LeaveCriticalSection(FController);  
    {при необходимости ожидания нужно выполнить следующее}  
    if HaveToWait then  
        WaitForSingleObject(FBlockedReaders, INFINITE);  
end;
```

Прежде всего, мы перехватываем управление критическим разделом. После этого можно осуществлять управление значениями внутренних полей. При наличии выполняющегося в текущий момент или хотя бы одного ожидающего потока записи метод увеличивает число ожидающих потоков считывания, освобождает управление критическим разделом, а затем переходит в состояние ожидания семафора "заблокированные потоки считывания". При отсутствии ожидающих или выполняющихся потоков записи метод увеличивает число выполняющихся потоков считывания и освобождает критический раздел. По выходу из этого метода программа либо освобождается от необходимости ожидать прихода семафора, либо сразу пропускает состояние ожидания. Обратите внимание, что во втором случае метод увеличил число выполняющихся потоков считывания, а в первом нет. Это может показаться программной ошибкой, но вскоре мы покажем, как можно решить возникающую при этом проблему.

Рассмотрим метод **StopReading**, код которого приведен в листинге 12.3.

Листинг 12.3. Метод **StopReading**

```
procedure TtdReadWriteSync.StopReading;  
begin  
  {перехватить управление критическим разделом}  
  EnterCriticalSection(FController);  
  {считывание завершено}  
  dec(FActiveReaders);  
  {если выполняется последний поток считывания и при наличии по меньшей мере  
    одного ожидающего потока записи ему необходимо предоставить свободу  
    действий}  
  if (FActiveReaders = 0) and (FWaitingWriters <> 0) then begin  
    dec(FWaitingWriters);  
    FActiveWriter := true;  
    ReleaseSemaphore(FBlockedWriters, 1, nil);  
  end;  
  {освободить управление критическим разделом}  
  LeaveCriticalSection(FController);  
end;
```

Как обычно, прежде всего, мы перехватываем управление критическим разделом. Этот поток стремится прекратить свои действия по считыванию, поэтому он уменьшает значение счетчика выполняющихся потоков считывания. Если результирующее значение не равно нулю, это свидетельствует о наличии других активных потоков считывания. Поэтому метод просто освобождает управление критическим разделом и осуществляет выход. Однако если этот поток был последним активным потоком считывания, теперь значение счетчика равно нулю и нужно предоставить свободу действий ожидающему потоку записи (если таковой существует). Для этого метод освобождает семафор заблокированных потоков записи. Иначе говоря, метод увеличивает значение счетчика на единицу, в результате чего система предоставит свободу действий одному, и только одному, заблокированному потоку записи, после чего немедленно снова уменьшит значение счетчика до нуля, обеспечивая блокировку всех остальных потоков записи. Однако непосредственно перед тем метод **StopReading** уменьшает значение счетчика ожидающих потоков записи и увеличивает значение счетчика выполняющихся потоков записи. Общий результат выполнения этого кода состоит в том, что поток записи освобождается, а значения двух счетчиков потоков записи обновляются.

Перейдем к рассмотрению метода **StartWriting**, код которого приведен в листинге 12.4.

Вначале снова необходимо перехватить управление критическим разделом. При наличии любых выполняющихся потоков считывания или записи метод увеличивает значение счетчика ожидающих потоков записи, освобождает управление критическим разделом, а затем ожидает освобождения семафора заблокированных потоков записи.

Листинг 12.4. Метод StartWriting

```

procedure TtdReadWriteSync.StartWriting;
var
    HaveToWait : boolean;
begin
    {перехватить управление критическим разделом}
    EnterCriticalSection(FController);
    {при наличии еще одного запущенного потока записи или активных потоков
    считывания, метод добавляет себя в качестве ожидающего потока считывания и
    обеспечивает переход в состояние ожидания}
    if FActiveWriter or (FActiveReaders <> 0) then begin
        inc(FWaitingWriters);
        HaveToWait := true;
    end
    {в противном случае метод должен добавить себя в качестве еще одного
    выполняющегося потока записи и обеспечить отсутствие состояния ожидания}
    else begin
        FActiveWriter := true;
        HaveToWait := false;
    end;
    {освободить управление критическим разделом}
    LeaveCriticalSection(FController);
    {при необходимости ожидания нужно выполнить следующее}
    if HaveToWait then
        WaitForSingleObject(FBlockedWriters, INFINITE);
end;

```

При отсутствии каких-либо других выполняющихся потоков можно сразу начать запись. Метод увеличивает значение счетчика выполняющихся потоков записи, освобождает управление критическим разделом и осуществляет выход из подпрограммы. В любом случае, сразу по выходу из подпрограммы значение счетчика активных потоков записи оказывается установленным равным единице (либо самим этим методом, либо методом **StopReading** – если помните, это происходит непосредственно перед передачей семафора заблокированных потоков записи).

И, наконец, можно приступить к рассмотрению метода **StopWriting**, код которого приведен в листинге 12.5.

Как и ранее, первоначальная задача состоит в перехвате управления критическим разделом. Затем, поскольку запись завершена, метод уменьшает значение счетчика активных потоков записи. Теперь выполняется проверка количества ожидающих потоков считывания. Мы входим в цикл, который уменьшает значение счетчика активных потоков считывания и освобождает семафор. Семафор, в свою очередь, освобождает от ожидания один поток считывания. Со временем, по завершении цикла, все потоки считывания будут освобождены и смогут считаться активными (обратите внимание, что они все будут использовать соответствующее обращение к методу **StartReading**). Если, с другой стороны, не существует ника-

ких ожидающих потоков считывания, метод выполняет проверку на наличие каких-либо ожидающих потоков записи. Если такие потоки существуют, метод освобождает только один поток записи таким же образом, как уже было описано при рассмотрении метода **StopReading**. И, наконец, независимо ни от чего, метод освобождает управление критическим разделом.

Листинг 12.5. Метод **StopWriting**

```
procedure TtdReadWriteSync.StopWriting;
var
  i : integer;
begin
  {перехватить управление критическим разделом}
  EnterCriticalSection(FController);
  {запись завершена}
  FActiveWriter := false;
  {если имеется хотя бы один ожидающий поток записи, освободить их всех}
  if (FWaitingReaders <> 0) then begin
    FActiveReaders := FWaitingReaders;
    FWaitingReaders := 0;
    ReleaseSemaphore(FBlockedReaders, FActiveReaders, nil);
  end
  {в противном случае, при наличии по меньшей мере одного ожидающего потока
  записи, ему необходимо предоставить свободу действий}
  else if (FWaitingWriters <> 0) then begin
    dec(FWaitingWriters);
    FActiveWriter := true;
    ReleaseSemaphore(FBlockedWriters, 1, nil);
  end;
  {освободить управление критическим разделом}
  LeaveCriticalSection(FController);
end;
```

Нам осталось рассмотреть только два метода: конструктор **Create** и деструктор **Destroy**. Код реализации этих методов показан в листинге 12.6.

Листинг 12.6. Создание и уничтожение объекта синхронизации

```
constructor TtdReadWriteSync.Create;
var
  NameZ : array [0..MAX_PATH] of AnsiChar;
begin
  inherited Create;
  {создать примитивные объекты синхронизации}
  GetRandomObjName(NameZ, 'tdRW.BlockedReaders');
  FBlockedReaders := CreateSemaphore(nil, 0, MaxReaders, NameZ);
  GetRandomObjName(NameZ, 'tdRW.BlockedWriters');
  FBlockedWriters := CreateSemaphore(nil, 0, 1, NameZ);
  InitializeCriticalSection(FController);
end;
```

```
destructor TtdReadWriteSync.Destroy;  
begin  
  CloseHandle(FBlockedReaders);  
  CloseHandle(FBlockedWriters);  
  DeleteCriticalSection(FController);  
  inherited Destroy;  
end;
```

Как видите, конструктор **Create** будет создавать три примитивных объекта синхронизации, а деструктор **Destroy** будет, соответственно, их уничтожать.

Полный исходный код класса **TtdReadWriteSync** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDRWSync.pas**.

Алгоритм производителей-потребителей

Еще один многопоточный алгоритм, тесно связанный с проблемой потоков считывания и записи – алгоритм, решающий проблему производителей и потребителей.

Этот раздел адресован только тем программистам, которые работают в среде 32-разрядной Windows. Delphi I вообще не поддерживает многопоточную обработку, в то время как Kylix и Linux не предоставляют необходимых примитивных объектов синхронизации, с помощью которых можно было бы решить проблему производителей-потребителей.

В этой ситуации имеется один или более потоков, создающих данные (их называют *производителями (producers)*), которые будут использоваться или потребляться одним или большим количеством других потоков (называемых *потребителями (consumers)*). Как видите, эта задача тесно связана с алгоритмом потоков считывания-записи: потребителей можно считать потоками считывания данных, записанных производителями. Примером использования этого алгоритма может послужить программа потокового видео: в этом случае будет существовать поток, который загружает видео из какого-то Web-сайта, и поток, который воспроизводит загруженное видео. Ни один из этих потоков не должен беспокоиться о том, что должен делать второй.

Мы симитируем этот процесс подпрограммой копирования нескольких потоков. Производитель будет копировать данные из потока в очередь буферов. Затем потребитель будет копировать данные из буферов в другой поток. Например, мог бы существовать производитель, считывающий несжатые данные из потока, и два потребителя данных: один, сжимающий данные в другой поток с помощью одного алгоритма, и второй, сжимающий их с помощью другого алгоритма, что теоретически позволяет выбирать более плотно упакованные данные. В этом случае производитель может продолжать работу и пытаться максимально быстро заполнять буфера в очереди, а потребители, в свою очередь, могут пытаться максимально быстро их считывать. Работа производителя будет тормозиться, если потребители

работают недостаточно быстро и очередь заполняется непрочитанными буферами. Аналогично, работа потребителей будет замедляться, если производитель работает медленно и очередь опустошается.

Модель с одним производителем и одним потребителем

Вначале рассмотрим модель с одним производителем и одним потребителем. Затем мы ее расширим до модели с одним производителем и несколькими потребителями. Нам необходимо, чтобы сразу после генерирования производителем "достаточного" объема данных потребитель мог начинать использовать уже сгенерированные данные. Поэтому необходимо рассмотреть три ситуации: производитель и потребитель работают согласованно; потребитель прекращает свою работу или блокируется, поскольку производитель не создал достаточный объем данных; производитель блокируется, поскольку потребитель не успел выполнить считывание уже созданных данных.

В примере с копированием потока производитель будет прекращать работу, если ему удастся заполнить все буферы прежде, чем потребитель успеет считать и обработать первый буфер. Потребитель будет блокироваться, если ему удастся обработать все буферы прежде, чем производитель успеет заполнить еще один буфер.

Следовательно, разрабатываемый нами класс синхронизации должен содержать четыре метода: вызываемый производителем, чтобы начать генерирование данных; вызываемый при наличии каких-либо данных, готовых для использования потребителем; вызываемый потребителем, чтобы начать потребление данных; и, наконец, вызываемый потребителем по завершении потребления им объема данных, достаточного для возобновления генерации данных производителем. Как и в случае потоков считывания-записи, оба метода запуска могут блокировать вызывающие их потоки.

Полный код интерфейса и реализации класса производителя-потребителя приведен в листинге 12.7. Как видите, реализация весьма проста.

Листинг 12.7. Класс синхронизации одного производителя и одного потребителя

```
type
  TtdProduceConsumeSync = class
  private
    FHasData : THandle; {семафор}
    FNeedsData : THandle; {семафор}
  protected
  public
    constructor Create(aBufferCount : integer);
    destructor Destroy; override;
    procedure StartConsuming;
    procedure StartProducing;
    procedure StopConsuming;
    procedure StopProducing;
  end;
```

Первым делом, мы рассмотрим метод **StartProducing** (см. листинг 12.8), вызываемый производителем для запуска генерирования данных. Метод будет вызывать блокировку, если потребитель не успел использовать достаточно данных, чтобы производитель мог заменить их новыми. Метод достаточно прост: он просто ожидает передачи семафора "требуется данные". Как мы увидим, этот семафор будет передаваться потребителем.

Листинг 12.8. Метод StartProducing

```
procedure TtdProduceConsumeSync.StartProducing;  
begin  
    { чтобы генерирование было начато, должен быть передан семафор  
      "требуется данные" }  
    WaitForSingleObject(FNeedsData, INFINITE);  
end;
```

Производитель будет вызывать второй метод, **StopProducing** (см. листинг 12.9), сообщающий потребителю о том, что он сгенерировал определенные (возможно все) данные, и что, следовательно, существуют данные, которые нужно использовать. Его реализация также проста: код просто передает семафор "имеются данные", ожидаемый потребителем.

Листинг 12.9. Метод StopProducing

```
procedure TtdProduceConsumeSync.StopProducing;  
begin  
    { при генерировании каких-либо дополнительных данных потребителю нужно  
      сообщить о необходимости их использования }  
    ReleaseSemaphore(FHasData, 1, nil);  
end;
```

Третий метод, **StartConsuming** (листинг 12.10), вызывается потребителем перед тем, как он приступит к потреблению сгенерированных производителем данных. Метод будет вызывать блокировку на время ожидания семафора "имеются данные", который будет передаваться немедленно, если производитель уже сгенерировал какие-либо данные.

Листинг 12.10. Метод StartConsuming

```
procedure TtdProduceConsumeSync.StartConsuming;  
begin  
    { чтобы можно было начать потребление данных, должен быть передан семафор  
      "имеются данные" }  
    WaitForSingleObject(FHasData, INFINITE);  
end;
```

Последний метод, **StopConsuming** (листинг 12.11), вызывается потребителем при считывании им достаточного объема (или всех) данных, чтобы производитель мог сгенерировать дополнительные данные. Очевидно, что этот метод всего лишь пе-

редает семафор "требуется данные", который будет предоставлять свободу действий производителю, если тот находится в состоянии ожидания.

Листинг 12.11. Метод `StopConsuming`

```
procedure TtdProduceConsumeSync.StopConsuming;  
begin  
    {если какие-либо данные были использованы, нужно сигнализировать производи-  
    телю о необходимости генерации дополнительных данных}  
    ReleaseSemaphore(FNeedsData, 1, nil);  
end;
```

Полный исходный код класса **TtdProduceConsumeSync** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDPCSync.pas**.

Обратите внимание, что при использовании объекта семафора Windows неявно предполагается, что данные могут храниться только в 127 или меньшем количестве буферов, поскольку каждый раз, когда производитель сообщает, что потребитель может использовать какие-либо дополнительные данные, значение семафора "имеются данные" увеличивается на единицу (а его максимальное значение ограничено величиной, равной 127). Аналогичные соображения справедливы и по отношению к семафору "требуется данные". Однако в целом, это не столь уж большое ограничение. Во множестве сценариев с применением производителя-потребителя для передачи данных используется всего один буфер, а подпрограмма копирования потока, которую мы будем рассматривать, использует очередь буферов, содержащую 20 элементов.

Очередь буферов, используемая в рассматриваемом примере копирования потока, реализована в виде циклической очереди. Очередь создается с заранее выделенными всеми ее буферами. Код реализации этого класса приведен в листинге 12.12.

Обратите внимание, что мы не будем использовать диспетчер кучи во время процесса копирования потока, поскольку критический раздел защищает диспетчер кучи в многопоточной подпрограмме. Если начать вызывать подпрограммы распределения и освобождения памяти из потоков, они слишком легко смогут блокировать одна другую и, возможно, препятствовать достижению основной цели применения класса синхронизации производителя-потребителя.

Производитель будет заполнять буфер в начале очереди, а затем перемещать указатель начала очереди. С другой стороны, потребитель будет считывать данные из буфера в конце очереди, а затем перемещать конец очереди. Процессы заполнения и считывания могут происходить одновременно, поскольку они используют различные буферы.

Листинг 12.12. Класс TQueuedBuffers, предназначенный для выполнения копирования потока

```

type
  PBuffer = ^TBuffer;
  TBuffer = packed record
    bCount : longint;
    bBlock : array [0..pred(BufferSize)] of byte;
  end;
  PBufferArray = ^TBufferArray;
  TBufferArray = array [0..1023] of PBuffer;
type
  TQueuedBuffers = class
    private
      FBufCount : integer;
      FBuffers : PBufferArray;
      FHead : integer;
      FTail : integer;
    protected
      function qbGetHead : PBuffer;
      function qbGetTail : PBuffer;
    public
      constructor Create(aBufferCount : integer);
      destructor Destroy; override;
      procedure AdvanceHead;
      procedure AdvanceTail;
      property Head : PBuffer read qbGetHead;
      property Tail : PBuffer read qbGetTail;
    end;
constructor TQueuedBuffers.Create(aBufferCount : integer);
var
  i : integer;
begin
  inherited Create;
  {распределить буферы}
  FBuffers := AllocMem(aBufferCount * sizeof(pointer));
  for i := 0 to pred(aBufferCount) do
    GetMem(FBuffers^[i], sizeof(TBuffer));
  FBufCount := aBufferCount;
end;
destructor TQueuedBuffers.Destroy;
var
  i : integer;
begin
  {освободить буферы}
  if (FBuffers <> nil) then begin
    for i := 0 to pred(FBufCount) do
      if (FBuffers^[i] <> nil) then
        FreeMem(FBuffers^[i], sizeof(TBuffer));
    FreeMem(FBuffers, FBufCount * sizeof(pointer));
  end;
  inherited Destroy;
end;

```



```
procedure TQueuedBuffers.AdvanceHead;
begin
    inc(FHead);
    if (FHead = FBufCount) then
        FHead := 0;
end;
procedure TQueuedBuffers.AdvanceTail;
begin
    inc(FTail);
    if (FTail = FBufCount) then
        FTail := 0;
end;
function TQueuedBuffers.qbGetHead : PBuffer;
begin
    Result := FBuffers^[FHead];
end;
function TQueuedBuffers.qbGetTail : PBuffer;
begin
    Result := FBuffers^[FTail];
end;
```

Менее очевидно то, что указатели начала и конца очереди не должны быть защищены от изменений критическими разделами или какими-то аналогичными элементами. На первый взгляд это кажется противоречащим здравому смыслу и всем правилам совместного использования данных в различных потоках. Однако поток потребителя никогда не будет обращаться к указателю конца очереди. О наличии данных, которые нужно считать из указателя начала очереди, ему будет сообщать поток производителя (в этот момент времени указатели начала и конца очереди будут различными). Аналогично, поток производителя никогда не будет обращаться к указателю начала очереди, поскольку о наличии места для добавления данных в конце очереди ему будет сообщать поток потребителя.

Коды реализации классов производителя и потребителя приведены в листинге 12.13. Эти классы являются производными от класса **TThread**. Код реализации каждого из перекрытых методов **Execute** не отличается от ранее описанного. Поток производителя входит в цикл. На каждом шаге цикла он вызывает метод **StartProducer** объекта синхронизации, а затем считывает блок данных из исходного потока в буфер в конце очереди. После этого он смещает указатель конца очереди. И, в заключение, он вызывает метод **StopProducing** и повторяет цикл с начала. Выполнение цикла прекращается, как только поток производителя устанавливает буфер в состояние, соответствующее отсутствию в нем каких-либо данных (потребитель воспринимает это состояние в качестве признака "конец потока").

В свою очередь, цикл потока потребителя выполняется следующим образом. Вначале поток вызывает метод **StartConsuming** объекта синхронизации. Возврат из этого метода свидетельствует об отсутствии данных для считывания в объекте

поставленных в очередь буферов. Поток считывает данные из буфера, определяемого указателем начала очереди, и записывает их в поток назначения. Затем он смещает указатель начала очереди. Сразу после считывания всех данных из заполненного буфера он вызывает метод **StopConsuming** объекта синхронизации и повторяет цикл сначала. Работа потребителя останавливается при получении им пустого буфера.

Листинг 12.13. Классы производителя и потребителя

```

type
  TProducer = class (TThread)
    private
      FBuffers : TQueuedBuffers;
      FStream : TStream;
      FSyncObj : TtdProduceConsumeSync;
    protected
      procedure Execute; override;
    public
      constructor Create (aStream : TStream;
                          aSyncObj : TtdProduceConsumeSync;
                          aBuffers : TQueuedBuffers);

    end;
constructor TProducer.Create (aStream : TStream;
                              aSyncObj : TtdProduceConsumeSync;
                              aBuffers : TQueuedBuffers);

begin
  inherited Create (true);
  FStream := aStream;
  FSyncObj := aSyncObj;
  FBuffers := aBuffers;
end;
procedure TProducer.Execute;
var
  Tail : PBuffer;
begin
  { выполнять до момента опустошения потока... }
  repeat
    { сигнализировать о готовности к началу генерирования данных }
    FSyncObj.StartProducing;
    { считать блок из потока в конечный буфер }
    Tail := FBuffers.Tail;
    Tail^.bCount := FStream.Read (Tail^.bBlock, BufferSize);
    { переместить указатель конца очереди }
    FBuffers.AdvanceTail;
    { поскольку выполняется запись нового буфера, необходимо сигнализировать о
      созданных данных }
    FSyncObj.StopProducing;
  until (Tail^.bCount = 0);
end;

```

type

TConsumer = **class**(TThread)

private

FBuffers : TQueuedBuffers;

FStream : TStream;

FSyncObj : TtdProduceConsumeSync;

protected

procedure Execute; **override**;

public

constructor Create(aStream : TStream;

aSyncObj : TtdProduceConsumeSync;

aBuffers : TQueuedBuffers);

end;

constructor TConsumer.Create(aStream : TStream;

aSyncObj : TtdProduceConsumeSync;

aBuffers : TQueuedBuffers);

begin

inherited Create(true);

FStream := aStream;

FSyncObj := aSyncObj;

FBuffers := aBuffers;

end;

procedure TConsumer.Execute;

var

Head : PBuffer;

begin

{сигнализировать о готовности к началу потребления данных}

FSyncObj.StartConsuming;

{извлечь начальный буфер}

Head := FBuffers.Head;

{до тех пор, пока начальный буфер не опустошен...}

while (Head^.bCount <> 0) **do begin**

{выполнить запись блока из начального буфера в поток}

FStream.Write(Head^.bBlock, Head^.bCount);

{переместить указатель начала очереди}

FBuffers.AdvanceHead;

{поскольку было выполнено считывание и обработка буфера, необходимо сообщить о том, что данные были использованы}

FSyncObj.StopConsuming;

{сигнализировать о готовности снова приступить к потреблению данных}

FSyncObj.StartConsuming;

{извлечь начальный буфер}

Head := FBuffers.Head;

end;

end;

И, наконец, мы можем рассмотреть подпрограмму копирования потока, приведенную в листинге 12.14. Она принимает два параметра: входной поток и выходной поток. Подпрограмма создает специальный объект типа **TQueuedBuffers**. Этот объект содержит все ресурсы и методы, необходимые для реализации организованного в виде очереди набора буферов. Он создает также экземпляр класса **TtdProducerConsumerSync**, который будет действовать в качестве объекта синхронизации, обеспечивающего согласованную работу производителя и потребителя.

Листинг 12.14. Многопоточное копирование

```
procedure ThreadedCopyStream(aSrcStream, aDestStream : TStream);
var
    SyncObj : TtdProduceConsumeSync;
    Buffers : TQueuedBuffers;
    Producer : TProducer;
    Consumer : TConsumer;
    WaitArray : array [0..1] of THandle;
begin
    SyncObj := nil;
    Buffers := nil;
    Producer := nil;
    Consumer := nil;
    try
        {создать объект синхронизации, объект организованных в виде очереди
         буферов (с 20 буферами) и два потока}
        SyncObj := TtdProduceConsumeSync.Create(20);
        Buffers := TQueuedBuffers.Create(20);
        Producer := TProducer.Create(aSrcStream, SyncObj, Buffers);
        Consumer := TConsumer.Create(aDestStream, SyncObj, Buffers);
        {сохранить дескрипторы потоков, что обеспечивает возможность ожидания их
         передачи}
        WaitArray[0] := Producer.Handle;
        WaitArray[1] := Consumer.Handle;
        {запустить потоки}
        Consumer.Resume;
        Producer.Resume;
        {ожидать окончания потоков}
        WaitForMultipleObjects(2, @WaitArray, true, INFINITE);
    finally
        Producer.Free;
        Consumer.Free;
        Buffers.Free;
        SyncObj.Free;
    end;
end;
```

Затем подпрограмма копирования создает два потока, между которыми будет выполняться копирование, и возобновляет их выполнение (потоки создаются в приостановленном состоянии). Далее подпрограмма дожидается завершения обоих потоков и выполняет очистку. Полный код подпрограммы можно найти в файлах **TstCopy.dpr** и **TstCopyu.pas** на Web-сайте издательства, в разделе материалов.

Модель с одним производителем и несколькими потребителями

Реализовать рассмотренное приложение, в котором используется модель "производитель-потребитель", было достаточно просто. Теперь рассмотрим модель с одним производителем и несколькими потребителями. В этом случае имеется поток, который создает данные. Предположим, что существует несколько потоков, которым требуется считывать созданные данные. В упомянутом ранее примере использовались два потребителя, которые сжимали данные с применением разных алгоритмов. Еще одним примером мог бы служить браузер. Будем считать, что производитель выгружает Web-страницу из удаленного сайта, а один потребитель считывает HTML-код, чтобы выполнить его сохранение на диске, второй считывает код для его отображения на экране, а третий – с целью отображения индикатора выполнения. Создание этих процессов как отдельных потребителей упрощает написание кода, поскольку каждый процесс должен выполнять только одну задачу.

Итак, что же требуется, чтобы объект синхронизации поддерживал согласованную работу производителя и потребителей? Во-первых, производитель должен сообщать всем потребителям о наличии данных для считывания. Предположительно скорости работы потребителей будут различными, и поэтому они будут обрабатывать данные с различной частотой. Это предполагает существование по одному семафору "имеются данные" на каждый потребитель. Будем считать, что существует список буферов, которые производитель должен пополнять данными. И более того, этот список организован в виде циклической очереди. Следовательно, нам нужен единственный указатель конца очереди (управляемый исключительно производителем) и по одному указателю начала очереди для каждого потребителя, поскольку, по всей вероятности, каждый потребитель будет считывать буфера с различной частотой.

Так как же быть с производителем? Каким образом он узнает, что можно снова заполнять буфер данных? Понятно, что он может это делать только после того, как последний (предположительно самый медленный) потребитель прочитал достаточный объем данных, чтобы появилось место для его заполнения новыми данными (иначе говоря, как только буфер снова освободится). Это, в свою очередь, предполагает, что должен существовать счетчик потребителей для каждого буфера данных. Каждый раз, когда потребитель считывает данные из буфера, он уменьшает значение этого счетчика (число потребителей, которым еще только предстоит выполнить считывание данных из этого буфера). Таким образом, когда последний потребитель приступает к использованию каких-либо данных, известно, что он яв-

ляется последним, поскольку после уменьшения значение счетчика должно быть равно нулю. Обратите внимание, что потребители являются потоками и, следовательно, уменьшение значения счетчика следует выполнять безопасным для потоков образом.

Код этого расширенного класса **TtdProduceManyConsumeSync**, который позволяет нескольким потребителям потреблять данные, сгенерированные единственным производителем, приведен в листинге 12.15. Предполагается, что каждый поток потребителя имеет уникальный, начинающийся с нуля, идентификатор (на практике этого легко добиться, но при необходимости класс можно было бы расширить, чтобы потребители могли регистрироваться и отменять свою регистрацию, и чтобы идентификаторы присваивались им "на лету"). Затем потребитель использует этот идентификатор (числовое значение) при обращении к методам **StartConsumer** и **StopConsumer**.

Листинг 12.15. Класс синхронизации одного производителя и нескольких потребителей

```
type
  TtdProduceManyConsumeSync = class
    private
      FBufferCount : integer;      {счетчик буферов данных}
      FBufferInfo : TList;         {циклическая очередь информации о буферах}
      FBufferTail : integer;       {конец циклической очереди буферов}
      FConsumerCount : integer;    {счетчик потребителей}
      FConsumerInfo : TList;       {информация для каждого потребителя}
      FNeedsData : THandle;        {семафор}
    protected
    public
      constructor Create(aBufferCount : integer;
                        aConsumerCount : integer);
      destructor Destroy; override;
      procedure StartConsuming(aId : integer);
      procedure StartProducing;
      procedure StopConsuming(aId : integer);
      procedure StopProducing;
  end;
```

В этом классе предполагается, что производитель заполняет буфера, которые затем используются потребителями. Буфера не имеют никакой реальной реализации в самом классе. Их предоставление – задача пользователя класса.

Метод **StartProducing**, показанный в листинге 12.16, работает во многом аналогично описанному в предыдущем случае: он просто дожидается передачи ему семафора "требуется данные". (Этот семафор содержит значение, равное количеству буферов, что позволяет производителю заполнить все буфера.)

Метод **StopProducing**, также показанный в листинге 12.16, на этот раз должен выполнить несколько больший объем работы. Во-первых, счетчик использования потребителей только что заполненного им буфера должен быть установлен равным количеству потребителей. Обратите внимание, что поток производителя должен передавать все семафоры "имеются данные" (по одному для каждого потребителя), тем самым сообщая о наличии еще одного буфера, готового к использованию.

Листинг 12.16. Методы StartProducing и StopProducing

```

type
  PBufferInfo = ^TBufferInfo;
  TBufferInfo = packed record
    biToUseCount : integer; {счетчик потребителей, которым еще предстоит
                             использовать буфер}
  end;
type
  PConsumerInfo = ^TConsumerInfo;
  TConsumerInfo = packed record
    ciHasData : THandle; {семафор}
    ciHead : integer;    {указатель на начало очереди}
  end;
procedure TtdProduceManyConsumeSync.StartProducing;
begin
  {чтобы можно было начать генерацию данных, необходимо передать семафор
   "требуются данные"}
  WaitForSingleObject(FNeedsData, INFINITE);
end;
procedure TtdProduceManyConsumeSync.StopProducing;
var
  i : integer;
  BufInfo : PBufferInfo;
  ConsumerInfo : PConsumerInfo;
begin
  {в случае генерации каких-либо дополнительных данных необходимо установить
   счетчик потребителей буфера в конце очереди, чтобы тем самым обеспечить
   правильную обработку всех буферов}
  BufInfo := PBufferInfo(FBufferInfo[FBufferTail]);
  BufInfo^.biToUseCount := FConsumerCount;
  inc(FBufferTail);
  if (FBufferTail >= FBufferCount) then
    FBufferTail := 0;
  {теперь всем потребителям необходимо сообщить о наличии дополнительных
   данных}
  for i := 0 to pred(FConsumerCount) do begin
    ConsumerInfo := PConsumerInfo(FConsumerInfo[i]);
    ReleaseSemaphore(ConsumerInfo^.ciHasData, 1, nil);
  end;
end;

```

Чтобы разобраться с работой алгоритма с точки зрения потребителя, взгляните на листинг 12.17. Метод **StartConsuming** должен дожидаться передачи семафора "имеются данные", предназначенного для соответствующего потока потребителя (каждому потоку присвоен идентификатор потребителя). Метод **StopConsuming** – наиболее сложный во всем классе синхронизации. Вначале он извлекает информационную запись о буфере, соответствующую его собственному указателю на начало очереди. Затем он уменьшает значение счетчика потребителей, которым еще предстоит выполнить считывание (потребить) данный буфер. (Подпрограмма **InterlockedDecrement** – это составная часть интерфейса WIN32 API. Она уменьшает значение своего параметра безопасным для потоков образом и возвращает новое значение параметра.) Затем метод увеличивает указатель на начало очереди для данного потока потребителя и, если теперь число потребителей, которым еще предстоит выполнить считывание этого буфера, равно нулю, передает производителю семафор "требуется данные", чтобы побудить его сгенерировать новые данные.

Листинг 12.17. Методы **StartConsuming** и **StopConsuming**

```

procedure TtdProduceManyConsumeSync.StartConsuming(aId : integer);
var
    ConsumerInfo : PConsumerInfo;
begin
    { чтобы можно было начать потребление данных, потребителю с данным конкретным
      идентификатором должен быть передан семафор "имеются данные" }
    ConsumerInfo := PConsumerInfo(FConsumerInfo[aId]);
    WaitForSingleObject(ConsumerInfo^.ciHasData, INFINITE);
end;

procedure TtdProduceManyConsumeSync.StopConsuming(aId : integer);
var
    BufInfo : PBufferInfo;
    ConsumerInfo : PConsumerInfo;
    NumToRead : integer;
begin
    { мы выполнили считывание данных в буфере, на который указывает указатель
      начала очереди }
    ConsumerInfo := PConsumerInfo(FConsumerInfo[aId]);
    BufInfo := PBufferInfo(FBufferInfo[ConsumerInfo^.ciHead]);
    NumToRead := InterlockedDecrement(BufInfo^.biToUseCount);
    { переместить указатель начала очереди }
    inc(ConsumerInfo^.ciHead);
    if (ConsumerInfo^.ciHead >= FBufferCount) then
        ConsumerInfo^.ciHead := 0;
    { если данный поток был последним, который должен был использовать этот буфер,
      производителю нужно сигнализировать о необходимости генерирования новых
      данных }
    if (NumToRead = 0) then
        ReleaseSemaphore(FNeedsData, 1, nil);
end;

```

Конструктор и деструктор этого класса должны создавать и уничтожать большое количество объектов синхронизации, а также всю информацию о буфере и потребителе.

Листинг 12.18. Создание и уничтожение объекта синхронизации

```
constructor TtdProduceManyConsumeSync.Create(aBufferCount : integer;
                                             aConsumerCount : integer);

var
  NameZ : array [0..MAX_PATH] of AnsiChar;
  i : integer;
  BufInfo : PBufferInfo;
  ConsumerInfo : PConsumerInfo;
begin
  inherited Create;
  {создать семафор "требуется данные"}
  GetRandomObjName(NameZ, 'tdPMC.NeedsData');
  FNeedsData := CreateSemaphore(nil, aBufferCount, aBufferCount, NameZ);
  if (FNeedsData = INVALID_HANDLE_VALUE) then
    RaiseLastWin32Error;
  {создать циклическую очередь буферов и заполнить ее}
  FBufferCount := aBufferCount;
  FBufferInfo := TList.Create;
  FBufferInfo.Count := aBufferCount;
  for i := 0 to pred(aBufferCount) do begin
    New(BufInfo);
    BufInfo^.biToUseCount := 0;
    FBufferInfo[i] := BufInfo;
  end;
  {создать информационный список потребителей и заполнить его}
  FConsumerCount := aConsumerCount;
  FConsumerInfo := TList.Create;
  FConsumerInfo.Count := aConsumerCount;
  for i := 0 to pred(aConsumerCount) do begin
    New(ConsumerInfo);
    FConsumerInfo[i] := ConsumerInfo;
    GetRandomObjName(NameZ, 'tdPMC.HasData');
    ConsumerInfo^.ciHasData :=
      CreateSemaphore(nil, 0, aBufferCount, NameZ);
    if (ConsumerInfo^.ciHasData = INVALID_HANDLE_VALUE) then
      RaiseLastWin32Error;
    ConsumerInfo^.ciHead := 0;
  end;
end;
```

```

destructor TtdProduceManyConsumeSync.Destroy;
var
  i : integer;
  BufInfo : PBufferInfo;
  ConsumerInfo : PConsumerInfo;
begin
  {уничтожить семафор "требуется данные"}
  if (FNeedsData <> INVALID_HANDLE_VALUE) then
    CloseHandle(FNeedsData);
  {уничтожить информационный список потребителей}
  if (FConsumerInfo <> nil) then begin
    for i := 0 to pred(FConsumerCount) do begin
      ConsumerInfo := PConsumerInfo(FConsumerInfo[i]);
      if (ConsumerInfo <> nil) then begin
        if (ConsumerInfo^.ciHasData <> INVALID_HANDLE_VALUE) then
          CloseHandle(ConsumerInfo^.ciHasData);
        Dispose(ConsumerInfo);
      end;
    end;
    FConsumerInfo.Free;
  end;
  {уничтожить информационный список буферов}
  if (FBufferInfo <> nil) then begin
    for i := 0 to pred(FBufferCount) do begin
      BufInfo := PBufferInfo(FBufferInfo[i]);
      if (BufInfo <> nil) then
        Dispose(BufInfo);
    end;
    FBufferInfo.Free;
  end;
  inherited Destroy;
end;

```

Хотя, на первый взгляд, кажется, что в программе листинга 12.18 выполняется множество действий, в действительности все достаточно просто. Конструктор **Create** должен создать список буферов и заполнить его требуемым числом записей о буферах. Он должен также создать список потребителей и заполнить его соответствующим количеством записей о потребителях. Для каждой записи потребителя должен быть создан отдельный семафор. Деструктор **Destroy** должен уничтожить все эти объекты и освободить всю выделенную память.

Полный исходный код реализации класса **TtdProduceManyConsumeSync** можно найти на Web-сайте издательства, в разделе материалов. После загрузки материалов отыщите среди них файл **TDPCSync.pas**.

В качестве примера программы мы рассмотрим подпрограмму многопоточного копирования, выполняющую копирование потока в три других потока. Как и в случае примера, приведенного в листинге 12.14, производитель будет считывать

исходный поток в буфера, количество которых может доходить до 20. Потребители, количество которых теперь равняется трем, будут считывать буфера и выполнять запись в собственные потоки.

Класс **TQueuedBuffers** (листинг 12.19) должен быть несколько изменен, поскольку ему необходимо хранить указатель начала очереди для нескольких потребителей и, следовательно, он должен содержать массив таких указателей.

Листинг 12.19. Класс **TQueuedBuffers** для модели с несколькими потребителями

```

type
  PBuffer = ^TBuffer;
  TBuffer = packed record
    bCount : longint;
    bBlock : array [0..pred(BufferSize)] of byte;
  end;
  PBufferArray = ^TBufferArray;
  TBufferArray = array [0..pred(MaxBuffers)] of PBuffer;
  TQueuedBuffers = class
  private
    FBufCount : integer;
    FBuffers : PBufferArray;
    FConsumerCount : integer;
    FHead : array [0..pred(MaxConsumers)] of integer;
    FTail : integer;
  protected
    function qbGetHead(aInx : integer) : PBuffer;
    function qbGetTail : PBuffer;
  public
    constructor Create(aBufferCount : integer;
                      aConsumerCount : integer);
    destructor Destroy; override;
    procedure AdvanceHead(aConsumerId : integer);
    procedure AdvanceTail;
    property Head[aInx : integer] : PBuffer read qbGetHead;
    property Tail : PBuffer read qbGetTail;
    property ConsumerCount : integer read FConsumerCount;
  end;
  constructor TQueuedBuffers.Create(aBufferCount : integer;
                                    aConsumerCount : integer);

var
  i : integer;
begin
  inherited Create;
  {распределить буферы}
  FBuffers := AllocMem(aBufferCount * sizeof(pointer));
  for i := 0 to pred(aBufferCount) do
    GetMem(FBuffers^[i], sizeof(TBuffer));
  FBufCount := aBufferCount;
  FConsumerCount := aConsumerCount;
end;
```

```

destructor TQueuedBuffers.Destroy;
var
  i : integer;
begin
  {освободить буферы}
  if (FBuffers <> nil) then begin
    for i := 0 to pred(FBufCount) do
      if (FBuffers^[i] <> nil) then
        FreeMem(FBuffers^[i], sizeof(TBuffer));
    FreeMem(FBuffers, FBufCount * sizeof(pointer));
  end;
  inherited Destroy;
end;

procedure TQueuedBuffers.AdvanceHead(aConsumerId : integer);
begin
  inc(FHead[aConsumerId]);
  if (FHead[aConsumerId] = FBufCount) then
    FHead[aConsumerId] := 0;
end;

procedure TQueuedBuffers.AdvanceTail;
begin
  inc(FTail);
  if (FTail = FBufCount) then
    FTail := 0;
end;

function TQueuedBuffers.qbGetHead(aInx : integer) : PBuffer;
begin
  Result := FBuffers^[FHead[aInx]];
end;

function TQueuedBuffers.qbGetTail : PBuffer;
begin
  Result := FBuffers^[FTail];
end;

```

Следующей мы рассмотрим реализацию классов производителя и потребителя (листинг 12.20). Класс производителя претерпел не слишком много изменений по сравнению с предыдущей реализацией, в то время как класс потребителя теперь содержит идентификационный номер, посредством которого он обращается к объекту буферов для получения нужного указателя начала очереди.

Листинг 12.20. Классы производителя и потребителя

```

type
  TProducer = class(TThread)
    private
      FBuffers : TQueuedBuffers;
      FStream : TStream;
      FSyncObj : TtdProduceManyConsumeSync;
    protected
      procedure Execute; override;
    public
      constructor Create(aStream : TStream;
                        aSyncObj : TtdProduceManyConsumeSync;
                        aBuffers : TQueuedBuffers);

    end;
constructor TProducer.Create(aStream : TStream;
                             aSyncObj : TtdProduceManyConsumeSync;
                             aBuffers : TQueuedBuffers);

begin
  inherited Create(true);
  FStream := aStream;
  FSyncObj := aSyncObj;
  FBuffers := aBuffers;
end;
procedure TProducer.Execute;
var
  Tail : PBuffer;
begin
  {выполнять до тех пор, пока поток не будет исчерпан...}
  repeat
    {передать сигнал о готовности к началу генерации данных}
    FSyncObj.StartProducing;
    {выполнить считывание блока из потока в конечный буфер очереди}
    Tail := FBuffers.Tail;
    Tail^.bCount := FStream.Read(Tail^.bBlock, 1024);
    {переместить указатель конца очереди}
    FBuffers.AdvanceTail;
    {передать сигнал о прекращении генерации данных}
    FSyncObj.StopProducing;
  until (Tail^.bCount = 0);
end;
type
  TConsumer = class(TThread)
    private
      FBuffers : TQueuedBuffers;
      FID : integer;
      FStream : TStream;
      FSyncObj : TtdProduceManyConsumeSync;

```

```

protected
  procedure Execute; override;
public
  constructor Create(aStream : TStream;
                    aSyncObj : TtdProduceManyConsumeSync;
                    aBuffers : TQueuedBuffers;
                    aID : integer);

end;
constructor TConsumer.Create(aStream : TStream;
                             aSyncObj : TtdProduceManyConsumeSync;
                             aBuffers : TQueuedBuffers;
                             aID : integer);

begin
  inherited Create(true);
  FStream := aStream;
  FSyncObj := aSyncObj;
  FBuffers := aBuffers;
  FID := aID;
end;
procedure TConsumer.Execute;
var
  Head : PBuffer;
begin
  {передать сигнал о готовности к началу потребления данных}
  FSyncObj.StartConsuming(FID);
  {выполнить считывание начального буфера очереди}
  Head := FBuffers.Head[FID];
  {до тех пор, пока начальный буфер не пуст...}
  while (Head^.bCount <> 0) do begin
    {выполнить запись блока из начального буфера очереди в поток}
    FStream.Write(Head^.bBlock, Head^.bCount);
    {переместить указатель начала очереди}
    FBuffers.AdvanceHead(FID);
    {обработка этого буфера завершена}
    FSyncObj.StopConsuming(FID);
    {передать сигнал о повторной готовности к началу потребления данных}
    FSyncObj.StartConsuming(FID);
    {выполнить считывание начального буфера очереди}
    Head := FBuffers.Head[FID];
  end;
  {обработка последнего буфера завершена}
  FSyncObj.StopConsuming(FID);
end;

```

И, наконец, рассмотрим подпрограмму копирования потоков, код которой показан в листинге 12.21.

Листинг 12.21. Копирование потоков с применением модели "производитель-потребитель"

```

procedure ThreadedMultiCopyStream(aSrcStream : TStream;
                                   aDestCount : integer;
                                   aDestStreams : PStreamArray);

var
  i : integer;
  SyncObj : TtdProduceManyConsumeSync;
  Buffers : TQueuedBuffers;
  Producer : TProducer;
  Consumer : array [0..pred(MaxConsumers)] of TConsumer;
  WaitArray : array [0..MaxConsumers] of THandle;
begin
  SyncObj := nil;
  Buffers := nil;
  Producer := nil;
  for i := 0 to pred(MaxConsumers) do
    Consumer[i] := nil;
  for i := 0 to MaxConsumers do
    WaitArray[i] := 0;
  try
    {создать объект синхронизации}
    SyncObj := TtdProduceManyConsumeSync.Create(20, aDestCount);
    {создать объект буфера с очередью}
    Buffers := TQueuedBuffers.Create(20, aDestCount);
    {создать поток производителя и сохранить его дескриптор}
    Producer := TProducer.Create(aSrcStream, SyncObj, Buffers);
    WaitArray[0] := Producer.Handle;
    {создать потоки потребителей и сохранить их дескрипторы}
    for i := 0 to pred(aDestCount) do begin
      Consumer[i] := TConsumer.Create(
        aDestStreams^[i], SyncObj, Buffers, i);
      WaitArray[i+1] := Consumer[i].Handle;
    end;
    {запустить потоки}
    for i := 0 to pred(aDestCount) do
      Consumer[i].Resume;
    Producer.Resume;
    {ожидать завершения потоков}
    WaitForMultipleObjects(1+aDestCount, @WaitArray, true, INFINITE);
  finally
    Producer.Free;
    for i := 0 to pred(aDestCount) do
      Consumer[i].Free;
    Buffers.Free;
    SyncObj.Free;
  end;
end;

```

Большая часть кода предназначена для выполнения тех же рутинных задач, что и в модели с одним потребителем, представленной в листинге 12.14, за исключением того, что на этот раз необходимо заботиться о нескольких потребителях. Полный код подпрограммы находится в файлах **TstNCpy.dpr** и **TstNCpyu.pas** на Web-сайте издательства, в разделе материалов.

Поиск различий между двумя файлами

Рассмотрим следующую задачу. Имеются две версии исходного файла, одна из которых – более поздняя, содержащая ряд изменений. Как выяснить различия между этими двумя файлами? Какие строки были добавлены, а какие удалены? Какие строки изменились?

Существует множество программ, выполняющих подобные функции. В их числе и программа **diff**, которую можно считать прародительницей всех программ сравнения файлов. Пакет Microsoft Windows SDK содержит программу, названную **WinDiff**. Программа **Visual SourceSafe**, поставляемая компанией Microsoft, также предоставляет функцию, которая позволяет выбрать две версии файла, хранящиеся в базе данных, и просмотреть различия между ними.

Этот раздел адресован только тем программистам, которые работают в 32-разрядной среде. Рассмотренный здесь алгоритм является рекурсивным и интенсивно использует программный стек. Delphi 1 не поддерживает достаточно большой стек, чтобы с его помощью можно было реализовать этот алгоритм даже для сравнительно умеренных по размерам файлов.

Потратим несколько минут, и попытаемся определить требуемый для выполнения этой задачи алгоритм. Раньше я уже пытался сделать это, что оказалось достаточно трудно. Кое-что можно упростить сразу: изменение строки можно считать удалением старой строки и вставкой новой. Мы не будем углубляться в проблемы семантики, пытаюсь выяснить, насколько сильно изменилась строка. Мы всего лишь будем рассматривать все изменения в текстовом файле как набор удаленных строк и набор вставленных новых строк.

Вычисление LCS двух строк

Требуемый нам алгоритм известен под названием алгоритма определения *наиболее длинной общей подпоследовательности* (*longest common subsequence* – *LCS*). Вначале мы рассмотрим, как он работает применительно к строкам, а затем расширим приобретенные представления на текстовые файлы.

Уверен, что все мы играли с детскими головоломками, в которых нужно было преобразовать одно слово в другое, изменяя по одной букве. Все промежуточные варианты должны были быть также осмысленными словами. Так, преобразуя слово CAT в слово DOG, можно было бы выполнить следующие преобразования: CAT, COT, COG, DOG.

Смысл этих игр со словами заключается в простом удалении на каждом шаге одной буквы и вставке новой. Если бы не ограничения, накладываемые правилами игры, можно было бы наверняка преобразовать одно слово в другое, просто удалив все старые символы и вставив вместо них новые. Такой метод решения задачи можно сравнить с применением кувалды, нам же весьма желательно найти несколько более тонкий подход.

Предположим, что наша цель заключается в отыскании наименьшего количества изменений, требуемых для преобразования одного слова в другое. Для примера преобразуем слово BEGIN в слово FINISH. Мы видим, что нужно удалить буквы B, E и G, а затем вставить букву F перед оставшимися буквами и буквы I, S и H после них. Как же реализовать эти действия в виде алгоритма?

Один из возможных способов предполагает просмотр подпоследовательностей букв каждого слова и выяснение наличия в них общих последовательностей. *Подпоследовательность (subsequence)* строки образуется за счет удаления из нее одного или более символов. Оставшиеся символы не должны переставляться. Например, четырехбуквенными подпоследовательностями для строки BEGIN являются EGIN, BGIN, BEIN и BEGI. Как видите, они образуются путем поочередного отбрасывания одного из символов. Трехбуквенными подпоследовательностями являются BEG, BEI, BEN, BGI, BGN, BIN, EGI, EGN, EIN и GIN. Для данного слова существует 10 двухбуквенных подпоследовательностей и пять однобуквенных. Таким образом, для пятибуквенного слова существует всего 30 возможных подпоследовательностей, а в общем случае можно было бы показать, что для n -буквенной последовательности существует около 2^n подпоследовательностей. Пока что примите это утверждение на веру.

Алгоритм с применением "грубой силы", если его можно так назвать, заключается в просмотре двух слов BEGIN и FINISH и просмотре их пятибуквенных подпоследовательностей на предмет наличия каких-либо совпадений. Такие совпадения отсутствуют, поэтому для каждого слова то же самое нужно сделать, используя четырехбуквенные подпоследовательности. Как и в предыдущем случае, ни одна из подпоследовательностей не совпадает, поэтому мы переходим к рассмотрению трехбуквенных последовательностей. Результат снова отрицателен, поэтому мы переходим к сравнению двухбуквенных подпоследовательностей. Самой длинной общей подпоследовательностью этих двух слов является IN. Исходя из этого, можно определить, какие буквы необходимо удалить, а какие вставить.

Для коротких слов, подобных приведенному примеру, описанный подход не так уж плох. Но представим, что требуется просмотреть все подпоследовательности 100-символьной строки. Как уже упоминалось, их количество составляет 2^{100} . Алгоритм с применением "грубой силы" является экспоненциальным. Количество выполняемых операций пропорционально $O(2^n)$. Даже для строк средней длины поле поиска увеличивается чрезвычайно быстро. А это влечет за собой радикальное увеличение времени, требуемого для отыскания решения. Чтобы сказанное

было нагляднее, представим следующую ситуацию: предположим, что можно генерировать около биллиона подпоследовательностей в секунду (т.е. $2^{40} \approx 1\,099\,511\,627\,776$, или тысячу подпоследовательностей за один такт работы процессора ПК, тактовая частота которого равна 1 ГГц). Год содержит около 2^{25} секунд. Следовательно, для генерации всего набора подпоследовательностей для 100-символьного слова потребовалось бы 2^{35} (34 359 738 368) лет – 11-значное число. А теперь вспомните, что 100-символьная строка – всего лишь простенький пример того, что необходимо сделать: например, найти различие между двумя вариантами 600-строчного исходного файла.

Однако идея применения подпоследовательностей обладает своими достоинствами. Просто нужно подойти к ней с другой стороны. Вместо перечисления и сравнения всех подпоследовательностей в двух словах посмотрим, нельзя ли применить пошаговый подход.

Для начала предположим, что нам удалось найти наиболее длинную общую подпоследовательность двух слов (далее для ее обозначения мы будем использовать аббревиатуру "LCS"). В этом случае можно было бы соединить линиями буквы в LCS первого слова с буквами LCS второго слова. Эти линии не будут пересекаться. (Это обусловлено тем, что подпоследовательности определены так, что перестановки букв не допускаются. Поэтому буквы в LCS в обоих словах будут располагаться в одинаковом порядке.) LCS для слов "banana" и "abracadabra" (т.е. b, a, a, a) и линии, соединяющие совпадающие в них буквы, показаны на рис. 12.1. Обратите внимание, что для этой пары слов существует несколько возможных LCS. На рисунке показана лишь первая из них (занимающая самую левую позицию).

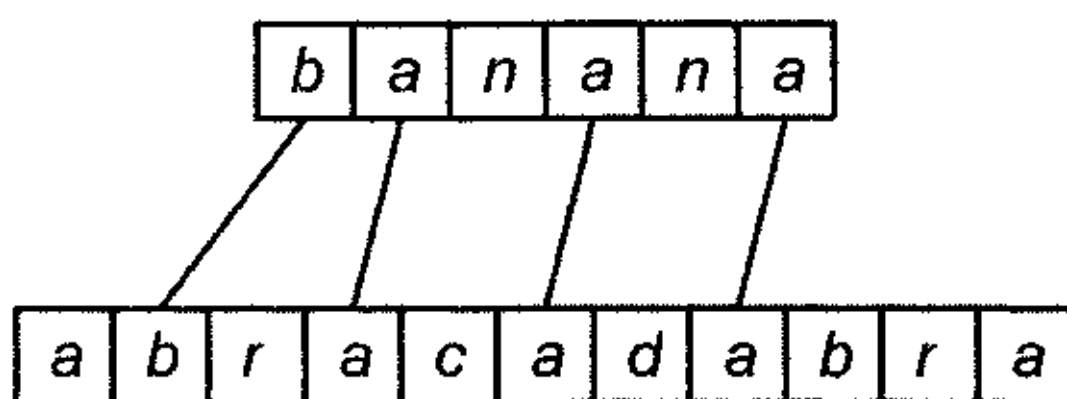


Рисунок 12.1. LCS для слов "banana" и "abracadabra"

Итак, тем или иным способом мы определили LCS двух слов. Предположим, что длина этой подпоследовательности равна x . Взгляните на последние буквы обоих слов. Если ни одна из них не является частью соединительной линии, и при этом они являются одной и той же буквой, то эта буква *должна* быть последней буквой LCS и между ними должна была бы существовать соединительная линия. (Если эта буква не является последней буквой подпоследовательности, ее можно было бы добавить, удлинив LCS на одну букву, что противоречило бы сделанному предположению о том, что первая подпоследовательность является самой длинной.) Удалим эту последнюю букву из обоих слов и из подпоследовательности.

Полученная сокращенная подпоследовательность длиной $x - 1$ представляет собой LCS двух сокращенных слов. (Если бы это было не так, для двух сокращенных слов должна была бы существовать общая подпоследовательность длиной x или больше. Добавление заключительных букв привело бы к увеличению длины новой общей подпоследовательности на единицу, а, значит, для двух полных слов должна была бы существовать общая подпоследовательность, содержащая $x+1$ или более букв. Это противоречит предположению о том, что мы определили LCS.)

Теперь предположим, что последняя буква в LCS не совпадает с последней буквой первого слова. Это означало бы, что LCS двух полных слов была бы также LCS первого слова без последней буквы и второго слова (если бы это было не так, можно было бы снова добавить последнюю букву к первому слову и найти более длинную LCS двух слов). Эти же рассуждения применимы и к случаю, когда последняя буква второго слова не совпадает с последней буквой LCS.

Все это замечательно, но о чем же оно свидетельствует? LCS содержит в себе LCS усеченных частей обоих слов. Для отыскания LCS строк X и Y мы разбиваем задачу на более мелкие задачи. Если бы последние символы слов X и Y совпадали, нам пришлось бы найти LCS для строк X и Y без их последних букв, а затем добавить эту общую букву. Если нет, нужно было бы найти LCS для строки X без последней буквы и строки Y , а также LCS строки X и строки Y без ее последней буквы, а затем выбрать более длинную из них. Мы получаем простой рекурсивный алгоритм.

Однако во избежание проблемы, которая может быть порождена простым решением, вначале необходимо описать алгоритм несколько подробнее.

Мы пытаемся вычислить LCS двух строк X и Y . Вначале мы определяем, что строка X содержит n символов, а строка Y — m . Обозначим строку, образованную первыми i символами строки X , как X_i . i может принимать также нулевое значение, что означает пустую строку (это соглашение упростит понимание алгоритма). В таком случае X_n соответствует всей строке. С применением этой формы записи алгоритм сводится к следующему: если последние два символа строк X_n и Y_m совпадают, самая длинная общая последовательность равна LCS X_{n-1} и Y_{m-1} с добавлением этого последнего символа. Если они не совпадают, LCS равна более длинной из LCS строк X_{n-1} и Y_m и LCS строк X_n и Y_{m-1} . Для вычисления этих "меньших" LCS мы рекурсивно вызываем одну и ту же подпрограмму.

Тем не менее, обратите внимание, что для вычисления LCS строк X_{n-1} и Y_m может потребоваться вычислить LCS строк X_{n-2} и Y_{m-1} , LCS строк X_{n-1} и Y_{m-1} и LCS строк X_{n-2} и Y_m . Вторую из этих подпоследовательностей можно уже вычислить. При недостаточной внимательности можно было бы вычислять одни и те же LCS снова и снова. В идеале во избежание этих повторных вычислений нужно было бы кешировать ранее вычисленные результаты. Поскольку мы располагаем двумя индексами для строк X и Y , имеет смысл воспользоваться матрицей.

Что необходимо хранить в каждом из элементов этого матричного кеша? Очевидный ответ – саму строку LCS. Однако, это не слишком целесообразно – да, это упростит вычисление LCS, но не поможет определить, какие символы нужно удалить из строки X , а какие новые символы вставить с целью получения строки Y . Лучше в каждом элементе хранить достаточный объем информации, чтобы можно было генерировать LCS за счет применения алгоритма типа $O(1)$, а также достаточный объем информации для определения команд редактирования, обеспечивающих переход от строки X к строке Y .

Один из информационных элементов, в котором мы действительно нуждаемся, – это длина LCS на каждом этапе. Используя упомянутое значение, с помощью рекурсивного алгоритма можно легко выяснить длину LCS для двух полных строк. Чтобы можно было сгенерировать саму строку LCS, необходимо знать путь, пройденный по матричному кешу. Для этого в каждом элементе потребуется сохранять указатель на предыдущий элемент, который был использован для построения LCS для данного элемента.

Однако прежде чем приступить к рассмотрению просмотра матрицы LCS, необходимо ее построить. Пока же будем считать, что в каждом элементе матрицы будут храниться два информационных фрагмента: длина LCS на данном этапе и позиция предыдущего элемента матрицы, образующего предшественницу этой LCS. Для последнего значения существует только три возможных ячейки: непосредственно над ним (к северу), слева (к западу) и выше и левее (к северо-западу). Поэтому для их обозначения вполне можно было бы использовать перечислимый тип.

Давайте вручную вычислим LCS для случая строк BEGIN/FINISH. Мы получим матрицу 6×7 (мы будем учитывать пустые подстроки, поэтому индексация должна начинаться с 0). Вместо того, чтобы рекурсивно заполнять матрицу (все эти рекурсивные вызовы трудно поддерживать в упорядоченном виде), итеративно вычислим все ячейки слева направо и сверху вниз. Вычисление ячеек первой строки и первого столбца не представляет сложности: они все являются нулями. Почему? Да потому, что наиболее длинная общая последовательность пустой и любой другой строки равна нулевой строке. С этого момента можно начать определение LCS для ячейки (1,1) или двух строк B и F. Два последних символа этих односимвольных строк не совпадают. Следовательно, длина LCS равна максимальной из предшествующих ячеек, расположенных к северу и к западу от данной. Обе эти ячейки нулевые, поэтому их максимальное значение и, следовательно, значение этой ячейки равно нулю. Ячейка (1,2) соответствует строкам B и FI. Ее значение также равно нулю. Ячейка (2,1) соответствует строкам BE и F: длина LCS снова равна 0. Продолжая подобные вычисления, можно заполнить все 42 ячейки матрицы. Обратите внимание на ячейки, соответствующие совпадающим символам: именно в них длина LCS возрастает. Конечный результат показан в таблице 12.1.

Таблица 12.1. Матрица LCS для строк BEGIN и FINISH

		F	I	N	I	S	H
	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0
I	0	0	1	1	1	1	1
N	0	0	1	2	2	2	2

Записать этот процесс выполнения действий вручную в виде кода не особенно трудно. Чтобы облегчить задачу начинающим программистам, я решил вначале создать класс матричного кеша. Внутри этого класса матрица хранится в объекте **TList** из **TLists**, причем ведущий объект **TList** представляет строки в матрице, а ведомый **TLists** – ячейки в столбцах отдельной строки. Кроме того, класс матрицы специфичен для решаемой задачи. Было бы излишним разрабатывать, кодировать и использовать общий класс матрицы. Код реализации класса матрицы показан в листинге 12.22.

Листинг 12.22. Класс матрицы для реализации алгоритма определения LCS

```

type
  TtdLCSDir = (ldNorth, ldNorthWest, ldWest);
  PtdLCSData = ^TtdLCSData;
  TtdLCSData = packed record
    ldLen : integer;
    ldPrev : TtdLCSDir;
  end;
type
  TtdLCSMatrix = class
    private
      FCols : integer;
      FMatrix : TList;
      FRows : integer;
    protected
      function mxGetItem(aRow, aCol : integer) : PtdLCSData;
      procedure mxSetItem(aRow, aCol : integer; aValue : PtdLCSData);
    public
      constructor Create(aRowCount, aColCount : integer);
      destructor Destroy; override;
      procedure Clear;
      property Items[aRow, aCol : integer] : PtdLCSData
        read mxGetItem write mxSetItem; default;
      property RowCount : integer read FRows;
      property ColCount : integer read FCols;
  end;

```

```

constructor TtdLCSMatrix.Create(aRowCount, aColCount : integer);
var
  Row : integer;
  ColList : TList;
begin
  {создать производный объект}
  inherited Create;
  {выполнить простую проверку}
  Assert((aRowCount > 0) and (aColCount > 0),
    'TtdLCSMatrix.Create: Invalid Row or column count');
  FRows := aRowCount;
  FCols := aColCount;
  {создать матрицу: она будет матрицей TList матриц TLists, упорядоченных по
    строкам}
  FMatrix := TList.Create;
  FMatrix.Count := aRowCount;
  for Row := 0 to pred(aRowCount) do begin
    ColList := TList.Create;
    ColList.Count := aColCount;
    TList(FMatrix.List^[Row]) := ColList;
  end;
end;
destructor TtdLCSMatrix.Destroy;
var
  Row : integer;
begin
  {уничтожить матрицу}
  if (FMatrix <> nil) then begin
    Clear;
    for Row := 0 to pred(FRows) do
      TList(FMatrix.List^[Row]).Free;
    FMatrix.Free;
  end;
  {уничтожить производный объект}
  inherited Destroy;
end;
procedure TtdLCSMatrix.Clear;
var
  Row, Col : integer;
  ColList : TList;
begin
  for Row := 0 to pred(FRows) do begin
    ColList := TList(FMatrix.List^[Row]);
    if (ColList <> nil) then
      for Col := 0 to pred(FCols) do begin
        if (ColList.List^[Col] <> nil) then
          Dispose(PtdLCSDData(ColList.List^[Col]));
          ColList.List^[Col] := nil;
        end;
      end;
    end;
  end;
end;

```

```

function TtdLCSMatrix.mxGetItem(aRow, aCol : integer) : PtdLCSData;
begin
  if not ((0 <= aRow) and (aRow < RowCount) and
    (0 <= aCol) and (aCol < ColCount)) then
    raise Exception.Create(
      'TtdLCSMatrix.mxGetItem: Row or column index out of bounds');
  Result := PtdLCSData(TList(FMatrix.List^[aRow]).List^[aCol]);
end;
procedure TtdLCSMatrix.mxSetItem(aRow, aCol : integer; aValue : PtdLCSData);
begin
  if not ((0 <= aRow) and (aRow < RowCount) and
    (0 <= aCol) and (aCol < ColCount)) then
    raise Exception.Create(
      'TtdLCSMatrix.mxSetItem: Row or column index out of bounds');
  TList(FMatrix.List^[aRow]).List^[aCol] := aValue;
end;

```

Следующий шаг заключается в создании класса, который реализует алгоритм вычисления LCS для строк. Код интерфейса и выполнения служебных функций класса **TtdStringLCS** приведен в листинге 12.23.

Листинг 12.23. Класс TtdStringLCS

```

type
  TtdStringLCS = class
    private
      FFromStr : string;
      FMatrix : TtdLCSMatrix;
      FToStr : string;
    protected
      procedure slFillMatrix;
      function slGetCell(aFromInx, aToInx : integer) : integer;
      procedure slWriteChange(var F : System.Text; aFromInx, aToInx : integer);
    public
      constructor Create(const aFromStr, aToStr : string);
      destructor Destroy; override;
      procedure WriteChanges(const aFileName : string);
  end;
constructor TtdStringLCS.Create(const aFromStr, aToStr : string);
begin
  {создать производный объект}
  inherited Create;
  {сохранить строки}
  FFromStr := aFromStr;
  FToStr := aToStr;
  {создать матрицу}
  FMatrix := TtdLCSMatrix.Create(succ(length(aFromStr)),
    succ(length(aToStr)));
  {заполнить матрицу}
  slFillMatrix;
end;

```

```

destructor TtdStringLCS.Destroy;
begin
  {уничтожить матрицу}
  FMatrix.Free;
  {уничтожить производный объект}
  inherited Destroy;
end;

```

При первой реализации алгоритма вычисления LCS я столкнулся с дилеммой: придерживаться ли ранее описанного рекурсивного алгоритма или же только что описанного процесса вычисления LCS вручную? Чтобы получить ответ на ряд вопросов (какой из методов проще, какой требует использования меньшего объема памяти, какой работает быстрее), я реализовал оба подхода, причем начал с реализации итеративного метода. Это итеративное решение приведено в листинге 12.24.

Листинг 12.24. Итеративное вычисление LCS

```

procedure TtdStringLCS.slFillMatrix;
var
  FromInx : integer;
  ToInx : integer;
  NorthLen: integer;
  WestLen : integer;
  LCSData : PtdLCSData;
begin
  {создать пустые элементы, располагающиеся вдоль верхней и левой сторон матрицы}
  for ToInx := 0 to length(FToStr) do begin
    New(LCSData);
    LCSData^.ldLen := 0;
    LCSData^.ldPrev := ldWest;
    FMatrix[0, ToInx] := LCSData;
  end;
  for FromInx := 1 to length(FFromStr) do begin
    New(LCSData);
    LCSData^.ldLen := 0;
    LCSData^.ldPrev := ldNorth;
    FMatrix[FromInx, 0] := LCSData;
  end;
  {построчное, слева направо, заполнение матрицы}
  for FromInx := 1 to length(FFromStr) do begin
    for ToInx := 1 to length(FToStr) do begin
      {создать новый элемент}
      New(LCSData);
      {если два текущих символа совпадают, необходимо увеличить значение
      счетчика элемента, расположенного к северо-западу, т.е. предыдущего
      элемента}
    end;
  end;

```



```

if (FFromStr[FromInx] = FToStr[ToInx]) then begin
  LCSData^.ldPrev := ldNorthWest;
  LCSData^.ldLen := succ(FMatrix[FromInx-1, ToInx-1]^ldLen);
end
{ в противном случае текущие символы различны: необходимо использовать
  максимальный из элементов, расположенных к северу или к западу от
  текущего (к западу предпочтительнее) }
else begin
  NorthLen := FMatrix[FromInx-1, ToInx]^ldLen;
  WestLen := FMatrix[FromInx, ToInx-1]^ldLen;
  if (NorthLen > WestLen) then begin
    LCSData^.ldPrev := ldNorth;
    LCSData^.ldLen := NorthLen;
  end
  else begin
    LCSData^.ldPrev := ldWest;
    LCSData^.ldLen := WestLen;
  end;
end;
{ установить элемент в матрице }
FMatrix[FromInx, ToInx] := LCSData;
end;
end;
{ на этом этапе длина элемента, расположенного в нижнем правом углу, равна
  LCS, и вычисление завершено }
end;

```

Мы начинаем с заполнения верхней строки и левого столбца матрицы нулевыми ячейками. Длина LCS в этих ячейках равна нулю (вспомните, что они описывают LCS пустой и какой-либо другой строки), и мы всего лишь устанавливаем флаг направления, дабы он указывал на предшествующую ячейку, ближайшую к ячейке (0,0). Затем следует вложенный цикл (цикл по столбцам внутри цикла по строкам). Для каждой строки мы вычисляем LCS для каждой из ячеек, просматривая их слева направо. Эти вычисления выполняются для всех строк сверху вниз. Вначале мы проверяем, совпадают ли два символа, на которые ссылается ячейка. (Ячейка матрицы представляет собой переход от символа строки **From** (Из) к символу строки **To** (В).) Если они совпадают, длина LCS в этой ячейке равна длине LCS ячейки, расположенной к северо-западу от данной, плюс единица. Обратите внимание, что способ вычисления ячеек предполагает, что ячейка, на которую осуществляется ссылка, уже вычислена (именно поэтому мы заранее вычислили значения ячеек, расположенных вдоль верхней и левой сторон матрицы). Если два символа не совпадают, необходимо просмотреть ячейки, расположенные к северу и к западу от текущей. Мы выбираем ту, которая содержит наиболее длинную LCS, и используем это значение в качестве значения данной ячейки. Если две длины равны, можно выбрать любую из них. Однако мы будем придерживаться пра-

вила, что предпочтительнее выбирать LCS, соответствующую ячейке, которая расположена слева. Этот выбор обусловлен тем, что как только путь через матрицу, обеспечивающий определение LCS обеих строк, вычислен, удаления из первой строки выполняются раньше вставок во вторую строку.

Обратите внимание, что приведенный в листинге 12.24 метод требует постоянного времени для обработки двух строк, независимо от степени их совпадения или несовпадения. Если длина строк равна, соответственно, n и m , то время, требуемое для выполнения основного цикла, будет пропорционально произведению $n \cdot m$, поскольку таковым является количество ячеек, значения которых нужно вычислить. (Помните, что ячейка, для которой действительно нужно получить ответ — последняя, значение которой должно вычисляться; она расположена в нижнем правом углу матрицы).

Алгоритм, реализованный с применением рекурсивного метода, приведен в листинге 12.25. Рекурсивная подпрограмма кодируется в виде функции, которая возвращает длину LCS для конкретной ячейки, заданной индексом строки и столбца (которые, в конечном счете, представляют собой индексы, указывающие на строки **From** и **To**).

Листинг 12.25. Рекурсивное вычисление LCS

```
function TtdStringLCS.slGetCell(aFromInx, aToInx : integer) : integer;
var
    LCSData : PtdLCSDData;
    NorthLen: integer;
    WestLen : integer;
begin
    if (aFromInx = 0) or (aToInx = 0) then
        Result := 0
    else begin
        LCSData := FMatrix[aFromInx, aToInx];
        if (LCSData <> nil) then
            Result := LCSData^.ldLen
        else begin
            {создать новый элемент}
            New(LCSData);
            {если два символа совпадают, необходимо увеличить значение счетчика
             относительно элемента, расположенного к северо-западу от данного, т.е.
             предшествующего элемента}
            if (FFromStr[aFromInx] = FToStr[aToInx]) then begin
                LCSData^.ldPrev := ldNorthWest;
                LCSData^.ldLen := slGetCell(aFromInx-1, aToInx-1) + 1;
            end
            {в противном случае текущие символы различаются: необходимо использовать
             максимальный из элементов, расположенных к северу и западу (выбор
             элемента расположенного к западу предпочтительнее)}
```

```
else begin
    NorthLen := slGetCell(aFromInx-1, aToInx);
    WestLen := slGetCell(aFromInx, aToInx-1);
    if (NorthLen > WestLen) then begin
        LCSData^.ldPrev := ldNorth;
        LCSData^.ldLen := NorthLen;
    end
    else begin
        LCSData^.ldPrev := ldWest;
        LCSData^.ldLen := WestLen;
    end;
end;
{установить значение элемента матрицы}
FMatrix[aFromInx, aToInx] := LCSData;
{вернуть длину данной LCS}
Result := LCSData^.ldLen;
end;
end;
end;
```

Первое существенное различие состоит в том, что не нужно генерировать нулевые значения для ячеек, расположенных вдоль верхней и правой сторон матрицы. Теперь эту задачу выполняет простой оператор **if**. (Честно говоря, в итеративном варианте вычисления LCS можно было бы обойтись без вычисления этих значений, но в этом случае внутренний код цикла оказался бы значительно сложнее для понимания и поддержки. Поэтому для простоты мы заранее вычисляем значения этих ячеек.) Если значение ячейки уже вычислено, мы просто возвращаем ее длину LCS. Если нет, необходимо выполнить ту же проверку, что и в предыдущем случае: совпадают ли два символа? Если да, то необходимо добавить единицу к значению LCS ячейки, расположенной к северо-западу от данной. Если нет, необходимо использовать большее из значений длины LCS ячеек, расположенных к северу и к западу от текущей. Естественно, эти значения LCS вычисляются в результате рекурсивных вызовов этой подпрограммы.

Применив обе версии (итеративную и рекурсивную), я сгенерировал матрицу для вычисления LCS слов "illiteracy" и "innumeracy". (Длина LCS этих слов равна 6 и выглядит как "ierasy".) Результаты этих немалых трудов приведены в таблицах 12.2 и 12.3. При использовании рекурсивной версии многие ячейки вообще не вычисляются (они помечены знаком вопроса). Эти ячейки образуют часть заключительной LCS.

Таблица 12.2. Итеративная матрица LCS слов "illiteracy" и "innumeracy".

		l	n	n	u	m	e	r	a	s	y
	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0
i	0	\ 1	-1	-1	-1	-1	-1	-1	-1	-1	-1
l	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
l	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i	0	\ 1	-1	-1	-1	-1	-1	-1	-1	-1	-1
t	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
e	0	1	-1	-1	-1	-1	\ 2	-2	-2	-2	-2
r	0	1	-1	-1	-1	-1	2	\ 3	-3	-3	-3
a	0	1	-1	-1	-1	-1	2	3	\ 4	-4	-4
s	0	1	-1	-1	-1	-1	2	3	4	\ 5	-5
y	0	1	-1	-1	-1	-1	2	3	4	5	\ 6

Таблица 12.3. Рекурсивная матрица LCS слов "illiteracy" и "innumeracy".

		i	n	n	u	m	e	r	a	s	y
	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0
i	? 0	\ 1	-1	-1	-1	-1	? 0	? 0	? 0	? 0	? 0
l	? 0	1	-1	-1	-1	-1	? 0	? 0	? 0	? 0	? 0
l	? 0	1	-1	-1	-1	-1	? 0	? 0	? 0	? 0	? 0
i	? 0	\ 1	-1	-1	-1	-1	? 0	? 0	? 0	? 0	? 0
t	? 0	1	-1	-1	-1	-1	? 0	? 0	? 0	? 0	? 0
e	? 0	? 0	? 0	? 0	? 0	? 0	\ 2	? 0	? 0	? 0	? 0
r	? 0	? 0	? 0	? 0	? 0	? 0	? 0	\ 3	? 0	? 0	? 0
a	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	\ 4	? 0	? 0
s	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	\ 5	? 0
y	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	? 0	\ 6

Итак, мы получили матрицу, которая определяет наиболее длинную общую подпоследовательность. Как ее можно использовать? Одна возможность связана с реализацией подпрограммы, которая создает текстовый файл, описывающий изменения, называемые *последовательностью редактирования* (*edit sequence*). Это может упростить создание аналогичной подпрограммы для текстового файла – что, собственно, является конечной целью данного раздела.

Код реализации простой технологии обхода, которая может быть приведена в соответствие с нашими потребностями, показан в листинге 12.26. Подпрограмма содержит два метода: первый вызывается пользователем с указанием имени файла, а второй представляет собой рекурсивную подпрограмму, которая записывает данные в файл. Весь основной объем работы выполняется во второй подпрограмме. Поскольку в матрице путь LCS кодируется в обратном направлении (т.е. для

определения пути необходимо начать с конца и продвигаться к началу матрицы), мы создаем метод, который вначале вызывает сам себя, а затем записывает данные, соответствующие текущей позиции. Необходимо обеспечить прерывание выполнения рекурсивной подпрограммы. Это соответствует случаю, когда подпрограмма вызывается для ячейки (0,0). В этом случае никакие данные не записываются в файл. Если индекс строки **To** равен нулю, мы выполняем рекурсивный вызов, перемещаясь вверх по матрице (индекс строки **From** уменьшается), и предпринимаемым действием должно быть удаление символа из строки **From**. Если индекс строки **From** равен нулю, мы выполняем рекурсивный вызов, перемещаясь по матрице влево, и тогда действием является ставка текущего символа в строку **To**. И, наконец, если оба индекса не равны нулю, мы находим соответствующую ячейку в матрице, выполняем рекурсивный вызов и записываем действие в файл. Перемещению вниз соответствует удаление, перемещению вправо – вставка, перемещению по диагонали – ни одно из упомянутых действий (символ "переносится" из одной строки в другую). Для обозначения удаления мы будем использовать стрелку, указывающую вправо (->), а для обозначения вставки – стрелку, указывающую влево (<-). Перенос символа не обозначается.

Листинг 12.26. Вывод последовательности редактирования

```

procedure TtdStringLCS.slWriteChange(var F : System.Text;
                                     aFromInx, aToInx : integer);

var
    Cell : PtdLCSDData;
begin
    {если оба индекса равны нулю, данная ячейка является первой ячейкой матрицы
     LCS, поэтому подпрограмма просто выполняет выход}
    if (aFromInx = 0) and (aToInx = 0) then
        Exit;
    {если индекс строки From равен нулю, ячейка расположена в левом столбце
     матрицы, поэтому необходимо переместиться вверх; этому будет соответство-
     вать удаление}
    if (aFromInx = 0) then begin
        slWriteChange(F, aFromInx, aToInx-1);
        writeln(F, '-> ', FToStr[aToInx]);
    end
    {если индекс строки To равен нулю, ячейка расположена в верхней строке
     матрицы, поэтому необходимо переместиться влево; этому будет соответство-
     вать вставка}
    else if (aToInx = 0) then begin
        slWriteChange(F, aFromInx-1, aToInx);
        writeln(F, '<- ', FFromStr[aFromInx]);
    end
    {в противном случае необходимо выполнить действия, указанные ячейкой}
    else begin
        Cell := FMatrix[aFromInx, aToInx];

```

```

case Cell^.ldPrev of
  ldNorth :
    begin
      slWriteChange(F, aFromInx-1, aToInx);
      writeln(F, '<- ', FFromStr[aFromInx]);
    end;
  ldNorthWest :
    begin
      slWriteChange(F, aFromInx-1, aToInx-1);
      writeln(F, ' ', FFromStr[aFromInx]);
    end;
  ldWest :
    begin
      slWriteChange(F, aFromInx, aToInx-1);
      writeln(F, '-> ', FToStr[aToInx]);
    end;
end;
end;
end;
procedure TtdStringLCS.WriteChanges(const aFileName : string);
var
  F : System.Text;
begin
  System.Assign(F, aFileName);
  System.Rewrite(F);
  try
    slWriteChange(F, length(FFromStr), length(FToStr));
  finally
    System.Close(F);
  end;
end;

```

Ниже показан текстовый файл, который был сгенерирован для преобразования слова "illiteracy" в слово "innumeracy".

```

<- i
<- l
<- l
  i
<- t
-> n
-> n
-> u
-> m
  e
  r
  a
  c
  y

```


Это представление действий по редактированию легко доступно для понимания, но при необходимости его можно развернуть. Как видите, наиболее длинная общая подпоследовательностью является (**i, e, r, a, c, y**), а определение удалений и вставок не представляет сложности.

Памятуя о том, что примененный метод является рекурсивным, следует подумать о требуемой для его реализации глубине стека. Если бы строки вообще не имели общих символов, последовательность редактирования сводилась бы к удалению всех символов первой строки и вставке всех символов второй строки. Если первая строка содержит n символов, а вторая m , глубина стека должна быть пропорциональной сумме $n + m$.

Вычисление LCS двух файлов

После того, как мы ознакомились с решением для двух строк, его можно модифицировать для вычисления LCS и генерации команд редактирования для двух текстовых файлов. Дабы упростить себе задачу, выполним считывание обоих файлов в объект **TStringsLists**. Понятно, что теперь одновременно выполняется сравнение целых текстовых строк, а не символов, тем не менее, в основном, реализация остается практически той же самой. Код интерфейса и вспомогательных методов приведен в листинге 12.27.

Листинг 12.27. Класс **TtdFileLCS**

```

type
  TtdFileLCS = class
  private
    FFromFile : TStringList;
    FMatrix : TtdLCSMatrix;
    FToFile : TStringList;
  protected
    function slGetCell(aFromInx, aToInx : integer) : integer;
    procedure slWriteChange(var F : System.Text;
                           aFromInx, aToInx : integer);
  public
    constructor Create(const aFromFile, aToFile : string);
    destructor Destroy; override;
    procedure WriteChanges(const aFileName : string);
  end;
constructor TtdFileLCS.Create(const aFromFile, aToFile : string);
begin
  {создать производный объект}
  inherited Create;
  {выполнить считывание файлов}
  FFromFile := TStringList.Create;
  FFromFile.LoadFromFile(aFromFile);
  FToFile := TStringList.Create;
  FToFile.LoadFromFile(aToFile);

```

```

    {создать матрицу}
    FMatrix := TtdLCSMatrix.Create(FFromFile.Count, FToFile.Count);
    {заполнить матрицу}
    slGetCell(pred(FFromFile.Count), pred(FToFile.Count));
end;
destructor TtdFileLCS.Destroy;
begin
    {уничтожить матрицу}
    FMatrix.Free;
    {освободить списки строк}
    FFromFile.Free;
    FToFile.Free;
    {уничтожить производный объект}
    inherited Destroy;
end;

```

Однако нужно решить одну проблему: при работе со строками отсчет символов начинается с 1, а при работе со списком строк отсчет строк (строк в исходном файле) начинается с 0. Поэтому необходимо внести ряд изменений.

Первое изменение заключается в простом кодировании рекурсивного метода. Если помните, итеративный метод требовал предварительного выделения ячеек, расположенных вдоль верхней и левой сторон матрицы, и установки их значений равными 0, в то время как в рекурсивном методе для выполнения этой задачи использовался оператор **if**. Потенциально это позволяет сэкономить достаточно большой объем памяти (в общем случае текстовые файлы могут содержать несколько сотен или даже тысяч строк).

Второе изменение, как уже отмечалось, – отсчет строк с 0. Рекурсивная подпрограмма автоматически решает эту задачу.

Код реализации рекурсивного метода генерирования LCS для двух файлов приведен в листинге 12.28.

Листинг 12.28. Генерация LCS для пары файлов

```

function TtdFileLCS.slGetCell(aFromInx, aToInx : integer) : integer;
var
    LCSData : PtdLCSData;
    NorthLen: integer;
    WestLen : integer;
begin
    if (aFromInx = -1) or (aToInx = -1) then
        Result := 0
    else begin
        LCSData := FMatrix[aFromInx, aToInx];
        if (LCSData <> nil) then
            Result := LCSData^.ldLen
        else begin
            {создать новый элемент}
            New(LCSData);

```

```

{если две текущие строки совпадают, необходимо увеличить значение
счетчика относительно элемента, расположенного к северо-западу от теку-
щего, т.е. предшествующего элемента}
if (FFromFile[aFromInx] = FToFile[aToInx]) then begin
    LCSData^.ldPrev := ldNorthWest;
    LCSData^.ldLen := slGetCell(aFromInx-1, aToInx-1) + 1;
end
{в противном случае текущие строки различны: необходимо использовать
максимальный из элементов, расположенных к северу или западу
(использование элемента, расположенного к западу, предпочтительнее)}
else begin
    NorthLen := slGetCell(aFromInx-1, aToInx);
    WestLen := slGetCell(aFromInx, aToInx-1);
    if (NorthLen > WestLen) then begin
        LCSData^.ldPrev := ldNorth;
        LCSData^.ldLen := NorthLen;
    end
    else begin
        LCSData^.ldPrev := ldWest;
        LCSData^.ldLen := WestLen;
    end;
end;
{установить элемент в матрице}
FMatrix[aFromInx, aToInx] := LCSData;
{вернуть длину данного LCS}
Result := LCSData^.ldLen;
end;
end;
end;

```

Метод записи последовательности редактирования, которая обеспечивает преобразование первого файла во второй, не особенно изменился по сравнению с рассмотренным ранее, за исключением того, что выполняется запись строк, а не символов. Эта подпрограмма приведена в листинге 12.29.

Листинг 12.29. Запись последовательности редактирования для пары файлов

```

procedure TtdFileLCS.slWriteChange(var F : System.Text;
                                   aFromInx, aToInx : integer);

var
    Cell : PtdLCSData;
begin
    {если оба индекса меньше нуля, данная ячейка является первой ячейкой матрицы
    LCS, поэтому подпрограмма просто выполняет выход}
    if (aFromInx = -1) and (aToInx = -1) then
        Exit;
    {если индекс строки From меньше нуля, ячейка расположена в левом столбце
    матрицы, поэтому необходимо переместиться вверх; этому будет
    соответствовать удаление}

```

```

if (aFromInx = -1) then begin
  slWriteChange(F, aFromInx, aToInx-1);
  writeln(F, '-> ', FToFile[aToInx]);
end
{если индекс строки To меньше нуля, ячейка расположена в верхней строке
 матрицы, поэтому необходимо переместиться влево; этому будет
 соответствовать вставка}
else if (aToInx = -1) then begin
  slWriteChange(F, aFromInx-1, aToInx);
  writeln(F, '<- ', FFromFile[aFromInx]);
end
{в противном случае необходимо выполнить действия, указанные ячейкой}
else begin
  Cell := FMatrix[aFromInx, aToInx];
  case Cell^.ldPrev of
    ldNorth :
      begin
        slWriteChange(F, aFromInx-1, aToInx);
        writeln(F, '<- ', FFromFile[aFromInx]);
      end;
    ldNorthWest :
      begin
        slWriteChange(F, aFromInx-1, aToInx-1);
        writeln(F, ' ', FFromFile[aFromInx]);
      end;
    ldWest :
      begin
        slWriteChange(F, aFromInx, aToInx-1);
        writeln(F, '-> ', FToFile[aToInx]);
      end;
    end;
  end;
end;
procedure TtdFileLCS.WriteChanges(const aFileName : string);
var
  F : System.Text;
begin
  System.Assign(F, aFileName);
  System.Rewrite(F);
  try
    slWriteChange(F, pred(FFromFile.Count), pred(FToFile.Count));
  finally
    System.Close(F);
  end;
end;

```

Резюме

В этой главе были рассмотрены три дополнительных алгоритма. Первые два предназначены для работы с многопоточными приложениями. Третий представляет собой весьма значимый, однако малоизвестный алгоритм поиска различий между двумя версиями файла.

Применительно к многопоточным приложениям, мы рассмотрели решение проблемы синхронизации потоков считывания-записи – алгоритма, который играет большую роль во множестве программ подобного рода. Применительно к проблеме использования потоков производителей-потребителей, мы рассмотрели алгоритм, который может применяться во многих ситуациях, когда большие объемы данных должны одновременно обрабатываться, причем различным образом.

Алгоритм определения наиболее длинной общей подпоследовательности (LCS) является более специализированным, но находит применение в системах управления исходным кодом, а также в качестве программ наподобие **diff**.

Эпилог

Если быть кратким, написание этой книги явилось интересным опытом (а также работой, попортившей немало кровушки).

В течение долгих лет я считал, что Delphi, Visual Basic, а теперь и Kylix, порождали, порождают и будут порождать программистов, которые не имеют ни малейшего представления об этом занятии. Да, они могут создавать приложения простым перетаскиванием, с использованием небольшого объема связующего кода и нескольких обработчиков событий. Тем не менее, любое приложение, достойное того, чтобы его создавать, требует определенного мастерства, опыта и теоретической подготовки, которые могут быть предоставлены традиционными компьютерными науками и программированием. Конечно, при создании программы можно немало напутать, и, тем не менее, программа таки будет работать. Однако различие будет столь же разительным, как и различие между яйцом, сваренным вкрутую, и яйцом Фаберже.

Должен признать, что вся моя теоретическая подготовка в компьютерной области была получена в результате самообразования. Я получил ученую степень по математике в Королевском колледже при Лондонском университете. Во время учебы мне довелось прослушать единственный курс по программированию – программированию на языке FORTRAN, программы которого, хранящиеся в виде колод перфокарт, были предвестниками сегодняшнего расцвета компьютерных технологий – но насколько я помню, не предпринималось никаких реальных попыток обучения студентов строгим компьютерным наукам. (В те времена не приходилось говорить и о немедленном получении результатов, столь привычном для современного программирования на ПК). Мне пришлось полюбить просматривать длинные листинги программ на языке, который не поддерживал ни локальных переменных,

ни указателей. Тем не менее, это меня не остановило. Я начал исследовать и изучать все эти премудрости. Долгими часами я мучился, пытаюсь усвоить язык MIX, разработанный Кнuthом (Knuth), C и иже с ними. Я пытался извлечь практическую пользу из учебников, которые оставляли реализацию операции **Delete** в качестве упражнения 4.25. Смею заверить, что все это – совершенно чудесный способ изучения языка.

Готов поспорить, что если вам стали известны доступные возможности языка, с которым вы более всего знакомы, то вы будете знать, следует ли при решении следующей задачи использовать хеш-таблицу или отложить клавиатуру и вычертить блок-схему конечного автомата, или, быть может, создать еще один экземпляр объекта списка **TList**. Именно в этом и состоит главная цель данной книги – показать читателям, чего можно добиться, если известны доступные возможности. Основное назначение кода, приведенного в книге – его непосредственное использование. (Вам требуется вычислитель регулярных выражений? Тогда воспользуйтесь кодом, разработанным нами в главе 10. Добавьте модуль в список **Uses** и – "лэте гоу", то бишь, за дело.)

Предупреждаю, что эта книга далеко не исчерпывающая. Когда я планировал ее написание, мне пришлось опустить больше материала, чем я смог осветить ("А чего ж ты не рассмотрел В-деревья, Юлиан?"). Угу... Так уж получилось... Прочтите книгу, а затем двигайтесь дальше, и постарайтесь выяснить, что еще написано по той или иной теме.

Список литературы

Ниже приведен список литературы, которая использовалась при написании этой книги. Некоторые из указанных работ имеют исключительно большое значение – без них я не смог бы разобраться в некоторых алгоритмах и пояснить их читателям применительно к Delphi. Другие содержат лишь материалы по менее серьезным темам, освещенным в других книгах. Тем не менее, это сделано так, что, на мой взгляд, вопрос становится более понятным.

1. Abramowitz, Milton, and Irene A. Stegun. *Handbook of Mathematical Functions*. Dover Publications, Inc., 1964.
2. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. Beck, Kent. *Extreme Programming Explained*. Addison-Wesley, 2000.
4. Binstock, Andrew, and John Rex. *Practical Algorithms for Programmers*. Addison-Wesley, 1995.
5. Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
6. Folk, Michael J., and Bill Zoellick. *File Structures*. 2nd Ed. Addison-Wesley, 1992.
7. Guibas L.J., and R. Sedgewick. "A dichromatic framework for balanced trees." *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, 1978.

8. Jones, Douglas W. "Application of Splay Trees to Data Compression." *Communications of the ACM*, Vol. 31 (1988), pp. 996-1007.
9. Kane, Thomas S. *The New Oxford Guide to Writing*. Oxford University Press, 1988.
10. King, Stephen. *On Writing*. Scribner, 2000.
11. Knuth, Donald E. *The Art of Computer Programming: Fundamental Algorithms*. 3rd Ed. Addison-Wesley, 1997.
12. Knuth, Donald E. *The Art of Computer Programming: Seminumerical Algorithms*. 3rd Ed. Addison-Wesley, 1998.
13. Knuth, Donald E. *The Art of Computer Programming: Sorting and Searching*. 2nd Ed. Addison-Wesley, 1998.
14. L'Ecuyer, Pierre. "Efficient and Portable Combined Random Number Generators." *Communications of the ACM*, Vol. 31 (1988), pp. 742-749, 774.
15. Nelson, Mark. *The Data Compression Book*. M&T Publishing, 1991.
16. Park, S.K., and K.W. Miller. "Random Number Generators: Good Ones are Hard to Find." *Communications of the ACM*, vol. 31 (1988), pp. 1192-1201.
17. Pham, Thuan Q. and Pankaj K. Garg. *Multithreaded Programming with Win32*. Prentice Hall, 1999.
18. Pugh, William. "Skip Lists: A Probabilistic Alternative to Balanced Trees." *Communications of the ACM*, Vol. 33 (1990), pp. 668-676.
19. Robbins, John. *Debugging Applications*. Microsoft Press, 2000.
20. Sedgewick, Robert. *Algorithms*. 2nd Ed. Addison-Wesley, 1988.
21. Sedgewick, Robert. *Algorithms in C*. 3rd Ed. Addison-Wesley, 1998.
22. Sleator, D.D., and R.E. Tarjan. "Self-adjusting binary search trees." *Journal of the ACM* (1985).
23. Thorpe, Danny. *Delphi Component Design*. Addison-Wesley Developers Press, 1996.
24. Wood, Derick. *Data Structures, Algorithms, and Performance*. Addison-Wesley, 1993.
25. Sedgewick, Robert. *Algorithms in C++. Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. 3rd Ed. Addison-Wesley, 1999.
26. Sedgewick, Robert. *Algorithms in C++. Parts 5: Graph Algorithms*. 3rd Ed. Addison-Wesley, 2002.
27. Роберт Седжвик. *Фундаментальные алгоритмы на C++. Части 1-4: Анализ/Структуры данных/Сортировка/Поиск*. – К.: Издательство «ДиаСофт», 2001.
28. Роберт Седжвик. *Фундаментальные алгоритмы на C++. Часть 5: Алгоритмы на графах*. – К.: Издательство «ДиаСофт», 2002.
29. Роберт Седжвик. *Фундаментальные алгоритмы на C. Части 1-4: Анализ/Структуры данных/Сортировка/Поиск*. – К.: Издательство «ДиаСофт», 2003.
30. Роберт Седжвик. *Фундаментальные алгоритмы на C. Часть 5: Алгоритмы на графах*. – К.: Издательство «ДиаСофт», 2003.
31. Джон Макгрегор, Девид Сайкс. *Тестирование объектно-ориентированного программного обеспечения. Практическое пособие*. – К.: Издательство «ДиаСофт», 2002.

Предметный указатель

А

Автомат 398

конечный 398

Алгоритм

20, 173, 205, 255, 367, 388, 449, 532

LRU 31

LZ77 489

анализ алгоритмов 22

восстановления скошенного дерева 476

двухпутевого слияния 173

кодирования лексем 444

кодирования Хаффмана 449, 454

кодирования Шеннона-Фано 449

красно-черный 340

наиболее длинной общей

подпоследовательности (LCS) 532

обхода дерева 308

Парка и Миллера 215

пирамидальной сортировки 376

поиска в дереве бинарного поиска 323

производителей-потребителей 512

просачивания вниз 368

пузырькового подъема 367

рандомизированный 205

с отходом 399

сжатия без потерь 444

сжатия с использованием скошенного дерева 470

сжатия с применением скошенного дерева 449

синтаксического анализа 388

схемы с открытой адресацией 255

считывания-записи 502

Флойда 374

Анализ покрытия 42

Б

Бинарный поиск 145

в отсортированном массиве TList 146

в отсортированном однонаправленном
связном списке 149

В

Выравнивание данных 31

Выражение

регулярное 410

Г

Генератор псевдослучайных чисел 207

Генератор случайных чисел 207, 213

аддитивный 226

комбинирование генераторов 224

минимальный стандартный 213

мультипликативный линейный конгруэнтный
213

тасующий 228

Группа 284

переполнения 284

Группирование в блоки 284

Д

Данные

сжатие 443

Двухсвязные списки 105

Декодирование 443

Делегирование 119

Дерево 303, 366, 453

бинарного поиска 322

бинарное 304

красно-черное 340

многопутевое 303

обход в глубину 308

обход в ширину 308

обход по уровням 315

префиксное 453

симметричный обход 308

скошенное 335

сортирующее 366. *См. также* Куча

косвенное 380

Хаффмана 455

Шеннона-Фано 452

Дескриптор 380

Деструктор 55

Диспетчер кучи 92

Диспетчер распределения узлов 93

З

Зондирование 255

квадратичное 270

линейное 255

псевдослучайное 270

И

Избыточность 443

Инвариант 40

К

Каталог 288

Класс

TFileStream 290

TList 61

TQueuedBuffers 516, 527

TtdArrayQueue 132

TtdArrayStack 121

TtdBinaryTree 322

- TtdDoubleLinkList 109
- TtdFileLCS 547
- TtdHashDirectory 290
- TtdHashTableChained 274
- TtdHashTableExtendible 293, 302
- TtdHashTableLinear 262, 269
- TtdInputBitStream 445
- TtdNodeManager 93, 96
- TtdOutputBitStream 445
- TtdPriorityQueue 369
- TtdPriorityQueueEx 381
- TtdProduceManyConsumeSync 522
- TtdQueue 127
- TtdReadWriteSync 507
- TtdRecordFile 290
- TtdRecordList 423
- TtdRecordStream 290
- TtdRedBlack 356
- TtdRedBlackTree 357
- TtdRegexEngine 441
- TtdRegexParser 417
- TtdSimplePriQueue1 363
- TtdSingleLinkList 97, 105
- TtdSplayTree 336
- TtdStack 119
- TtdStringLCS 539
- Ключ 322
- Кодирование 449
 - с использованием скошенного дерева 469
 - с минимальной избыточностью 444
 - Хаффмана 449, 454
 - Шеннона-Фано 449
- Комментарии 41
- Конечный автомат 388-389
 - детерминированный 398
 - недетерминированный 398
 - свободные переходы 398
- Конфликт (коллизия) 250
- Копирование 520
 - многопоточное 520
- Коэффициент сжатия 443
- Критерий хи-квадрат 207
- Куча 366
- Кэш процессора 31
- Кэш-память 31
- Л**
- Лексема 444
- Линейный конгруэнтный метод 212
- Локальность ссылок 30
- М**
- Массив 47, 139
 - динамический 48, 52, 60
 - расширение массива 58
 - стандартный 48
 - указателей 61
- Метасимвол 413
- Многопоточное копирование 520
- Н**
- Нотация 25
 - О- 25
- О**
- О-нотация 25
- Объект 505
 - примитивный 505
 - синхронизации 505
- Односвязные списки 84
- Операция
 - " " 439
 - "\$" 439
 - "*" 430
 - "+" 430
 - "?" 430
 - [] 60
 - "|" 428
 - Pop 121
 - Push 121
- Отладка 37, 44
- Отладчик TurboPower Sleuth QA Suite 45
- Отношение упорядочения набора объектов 360
- Очередь 84, 126, 360
 - кольцевая (круговая) 130
 - на основе массива 130
 - на основе односвязного списка 126
 - по приоритету 360
 - с двусторонним доступом 433
- П**
- Память 28
 - виртуальная 28
 - кэш- 31
 - страничная организация 28
- Пирамидальное свойство 366
- Подкачка 29
- Подпоследовательность 533
- Подпрограмма IsRed 347
- Поиск 21, 136, 139
 - бинарный 21, 145
 - последовательный 21, 139
 - в однонаправленном связном списке 144
- Постусловие 40
- Поток битов 445
- Потребитель (consumers) 512
- Предусловие 40
- Преобразование Бокса-Мюллера 231

Преобразование целочисленного значения в строку 124

Пробуксовка 29

Программирование 42

экстремальное 42

Производитель (producers) 512

Протоколирование 41

Профилировщик 22

Процедуры сравнения 136

Р

Разрешение конфликтов 250

посредством линейного зондирования 255

Разрыв грамматического правила 419

Распределение 231

нормальное (гауссово) 231

равномерное 231

хи-квадрат 208

экспоненциальное 232

Регулярные выражения 410

компиляция 420

С

Свободный список 92

Свопинг 29. *См. также* память: подкачка

Связный список 84, 233, 303

голова списка 86

двухсвязный 105

односвязный 84

прохождение связного списка 88

с пропусками 233

фиктивный начальный узел 90

Связывание 272

Семафор 505

Сжатие данных 443

без потерь 443

коэффициент сжатия 443

с использованием словаря 479

с минимальной избыточностью 449

с потерями 443

с применением скошенного дерева 449

с применением словаря 444

Сигнальный (sentinel) элемент 51

Сигнатура 491

Синтаксический анализ 388

Синтаксический анализатор 388, 413

нисходящий 413

Событие 505

Сортировка 153, 374

быстрая 182

методом вставок (простыми вставками) 153

методом выбора 162

методом прочесывания 170

методом Шелла 167

пирамидальная 374

пузырьковая 159

слиянием 173

Шейкер- 161

Список с пропусками 233

Стек 84, 118

на основе массива 121

на основе односвязного списка 118

Страница 28

Строка

длинная 35

Схемы с открытой адресацией 255

Т

Таблица

переходов 423

хеш 249, 285

Тест

на однородность 217

на пропусках 218

"покер" 220

"сбор купонов" 221

Тестирование 37, 217

модулей 42

регрессионное 43

Трассировка 41

У

Узел 84

диспетчер распределения узлов 93

фиктивный начальный 90

Указатели 61

Утверждение 38

инвариант 40

постусловие 40

предусловие 40

Ф

Файл индексов 285

Функция хеширования 249, 251, 253, 272

Х

Хеширование 249

двойное 271

расширяемое 286

Ч

Числа 206

взаимно простые 212

случайные 206

распределенные по экспоненциальному

закону 233

с нормальным распределением 232

Э

Экстремальное программирование 42

Элемент 51

сигнальный 51

Научное издание
Джулиан М. Бакнелл
**ФУНДАМЕНТАЛЬНЫЕ АЛГОРИТМЫ
И СТРУКТУРЫ ДАННЫХ В DELPHI**

Заведующий редакцией *С.Н.Козлов*
Научный редактор *Ю.Н.Артеменко*
Верстка *Т.Н.Артеменко*
Главный дизайнер *О.А.Шадрин*
Н/К

ООО «ДиаСофтЮП», 196105, Санкт-Петербург, пр. Ю.Гагарина, д. 1, ком. 108.
Лицензия №000328 от 9 декабря 1999 г.

Сдано в набор 10.02.2003. Подписано в печать 24.03.2003. Формат 70×100/16.
Бумага типографская. Гарнитура Таймс. Печать офсетная. Печ.л. 35.
Тираж 3000 экз. Заказ № 138

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29.

