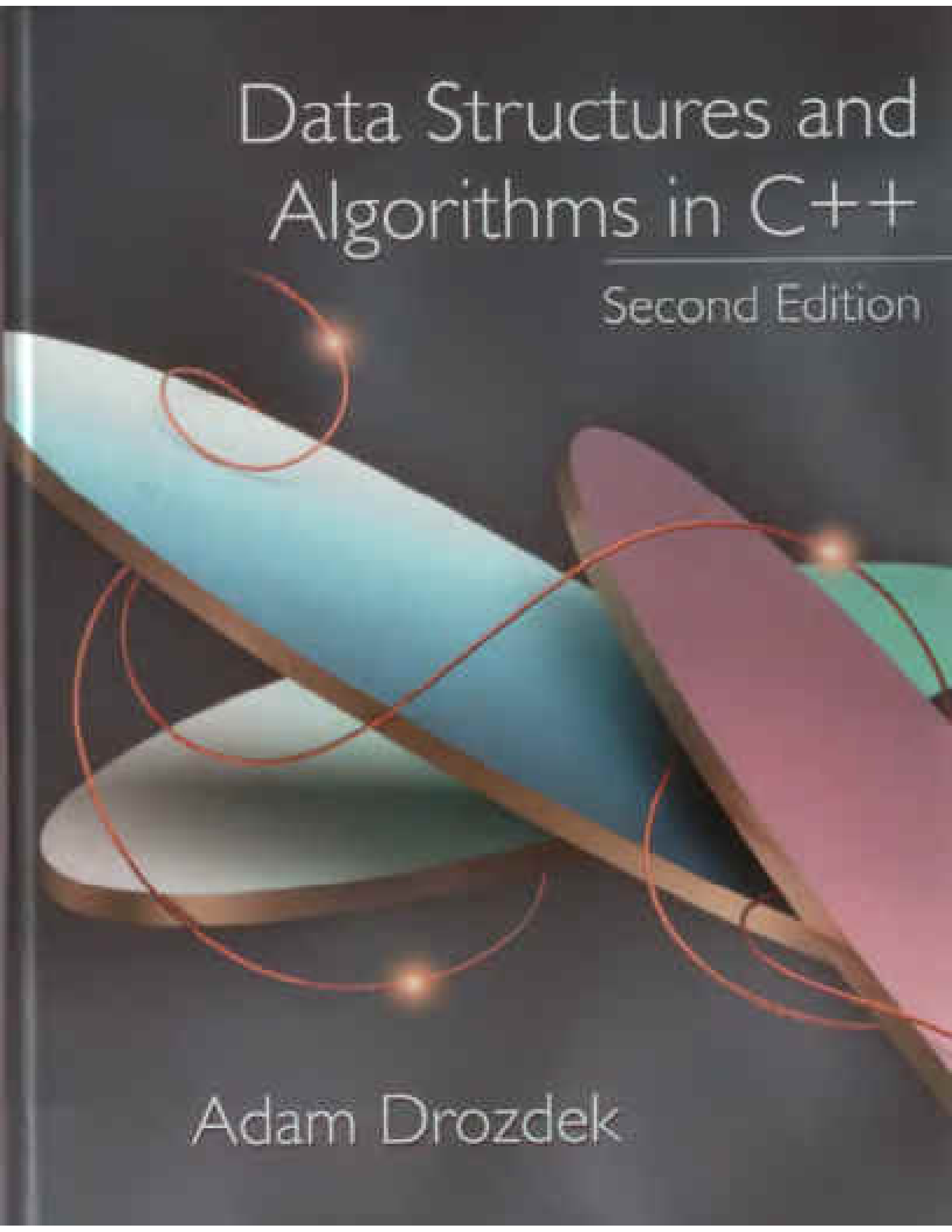
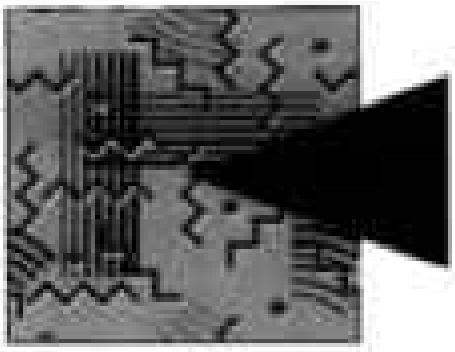
Data Structures and Algorithms in C++

Second Edition



Adam Drozdek

Data Structures and Algorithms in C++



Data Structures and Algorithms in C++

SECOND EDITION

Adam Drozdek

Brooks/Cole
Thomson Learning

Australia • Canada • Mexico • Singapore • Spain • United Kingdom • United States

Sponsoring Editor: Acquisitions Editor
Marketing Manager: Marketing Coordinator
Project Editor: Project Editor
Technical Reviewer: Technical Reviewer
Production Service: Production Service
Manuscript Editor: Manuscript Editor
Proofreader: Proofreader
Indexer: Indexer
Cover Designer: Cover Designer
Cover Photo: Cover Photo
Interior Illustrator: Interior Illustrator
Photo Researcher: Photo Researcher
Typesetting: Typesetting
Cover Printing: Cover Printing
Printing and Binding: Printing and Binding

COPYRIGHT © 2010 by Pearson Education
Publishing as Microsoft Press
One Microsoft Way
Redmond, Washington 98052-6399

A CIP catalog record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Microsoft, Microsoft Press, Access, Active Directory, Excel, Internet Explorer, Outlook, PowerPoint, SharePoint, Windows, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 10 11

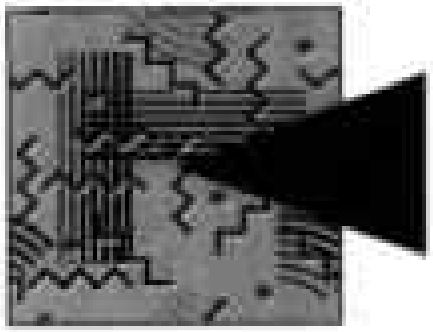
Library of Congress Cataloging-in-Publication Data

Microsoft Windows Server 2008 Active Directory, Second Edition / The Microsoft Windows Server Team.
p. cm.
Includes index.
ISBN 978-0-7356-2627-1
1. Microsoft Windows server. 2. Operating systems (Computers) 3. Client/server computing. I. Microsoft Windows Server Team.
QA76.76.O63M5135 2009
005.4'476--dc22

2009018760

To my wife, Bogna

Contents



1 Object-Oriented Programming Using C++ 1

1.1 Abstract Data Types 1

1.2 Encapsulation 2

1.3 Inheritance 6

1.4 Pointers 9

1.4.1 Pointers and Arrays 12

1.4.2 Pointers and Copy Constructors 14

1.4.3 Pointers and Destructors 16

1.4.4 Pointers and Reference Variables 17

1.4.5 Pointers to Functions 19

1.5 Polymorphism 20

1.6 C++ and Object-Oriented Programming 22

1.7 The Standard Template Library 23

1.7.1 Containers 23

1.7.2 Iterators 24

1.7.3 Algorithms 25

1.7.4 Function Objects 25

1.8 Vectors in the Standard Template Library 27

1.9 Data Structures and Object-Oriented Programming 35

1.10 Case Study: Random Access File 35

1.11 Exercises 48
1.12 Programming Assignments 50

2 Complexity Analysis 53
2.1 Computational and Asymptotic Complexity 55
2.2 Big-O Notation 54
2.3 Properties of Big-O Notation 57
2.4 Ω and Θ Notations 58
2.5 Possible Problems 59
2.6 Examples of Complexities 60
2.7 Finding Asymptotic Complexity: Examples 61
2.8 The Best, Average, and Worst Cases 63
2.9 Amortized Complexity 66
2.10 Exercises 71

3 Linked Lists 75
3.1 Singly Linked Lists 75
3.1.1 Insertion 81
3.1.2 Deletion 84
3.1.3 Search 91
3.2 Doubly Linked Lists 90
3.3 Circular Lists 95
3.4 Skip Lists 97
3.5 Self-Organizing Lists 102
3.6 Sparse Tables 106
3.7 Lists in the Standard Template Library 110
3.8 Deques in the Standard Template Library 114
3.9 Concluding Remarks 119
3.10 Case Study: A Library 120
3.11 Exercises 125
3.12 Programming Assignments 133

4 Stacks and Queues 137
4.1 Stacks 137
4.2 Queues 145
4.3 Priority Queues 151
4.4 Stacks in the Standard Template Library 155
4.5 Queues in the Standard Template Library 156
4.6 Priority Queues in the Standard Template Library 156
4.7 Case Study: Exiting a Maze 160
4.8 Exercises 166
4.9 Programming Assignments 169

5 Recursion 171
5.1 Recursive Definitions 171
5.2 Function Calls and Recursion Implementation 174
5.3 Anatomy of a Recursive Call 176
5.4 Tail Recursion 179
5.5 Nontail Recursion 181
5.6 Indirect Recursion 186
5.7 Nested Recursion 186
5.8 Excessive Recursion 189
5.9 Backtracking 192
5.10 Concluding Remarks 200
5.11 Case Study: A Recursive Descent Interpreter 201
5.12 Exercises 209
5.13 Programming Assignments 213

6 Binary Trees 216
6.1 Trees, Binary Trees, and Binary Search Trees 216
6.2 Implementing Binary Trees 221
6.3 Searching a Binary Search Tree 224
6.4 Tree Traversal 226
6.4.1 Breadth-First Traversal 227

6.4.3 Depth-First Traversal 227
6.4.4 Stackless Depth-First Traversal 234
6.5 Insertion 242
6.6 Deletion 245
6.6.1 Deletion by Merging 246
6.6.2 Deletion by Copying 249
6.7 Balancing a Tree 252
6.7.1 The DSW Algorithm 253
6.7.2 AVL Trees 258
6.8 Self-Adjusting Trees 264
6.8.1 Self-Restructuring Trees 264
6.8.2 Splaying 265
6.9 Heaps 271
6.9.1 Heaps as Priority Queues 272
6.9.2 Organizing Arrays as Heaps 275
6.10 Polish Notation and Expression Trees 279
6.10.1 Operations on Expression Trees 281
6.11 Case Study: Computing Word Frequencies 284
6.12 Exercises 291
6.13 Programming Assignments 295

7 Multiway Trees 302
7.1 The Family of B-Trees 307
7.1.1 B-Trees 312
7.1.2 B+-Trees 314
7.1.3 B*-Trees 316
7.1.4 Prefix B+-Trees 319
7.1.5 Bit-Trees 323
7.1.6 R-Trees 329
7.1.7 2-3 Trees 334
7.1.8 Sets and Multisets in the Standard Template Library 337
7.1.9 Maps and Multimaps in the Standard Template Library 339
7.2 Tries 345
7.3 Concluding Remarks 357
7.4 Case Study: Spell Checker 358
7.5 Exercises 364
7.6 Programming Assignments 366

8 GRAPHS 371

8.1 Graph Representation 373
8.2 Graph Traversals 373
8.3 Shortest Paths 378
8.3.1 A Fast All-Pairs Shortest Path Algorithm 383
8.4 Cycle Detection 388
8.4.1 Union-Find Problem 391
8.5 Spanning Trees 391
8.5.1 Borůvka's Algorithm 392
8.5.2 Kruskal's Algorithm 397
8.5.3 Jarník-Prim's Algorithm 399
8.5.4 Dijkstra's Method 398
8.6 Connectivity 396
8.6.1 Connectivity in Undirected Graphs 397
8.6.2 Connectivity in Directed Graphs 401
8.7 Topological Sort 402
8.8 Networks 404
8.8.1 Maximum Flows 404
8.8.2 Maximum Flows and Minimum Cuts 414
8.9 Matching 417
8.9.1 Assignment Problem 421
8.9.2 Matching in Nonbipartite Graphs 426
8.10 Eulerian and Hamiltonian Graphs 428
8.10.1 Eulerian Graphs 428
8.10.2 Hamiltonian Graphs 429
8.11 Case Study: Distinct Representatives 432
8.12 Exercises 444
8.13 Programming Assignments 448

9 SORTING 452

9.1 Elementary Sorting Algorithms 453
9.1.1 Insertion Sort 454
9.1.2 Selection Sort 456
9.1.3 Tri Pile Sort 460
9.2 Decision Trees 460

9.5 Efficient Sorting Algorithms 463
9.5.1 Shell Sort 463
9.5.2 Heap Sort 468
9.5.3 Quicksort 471
9.5.4 Mergesort 479
9.5.5 Radix Sort 488
9.4 Sorting in the Standard Template Library 488
9.5 Concluding Remarks 489
9.6 Case Study: Adding Polynomials 491
9.7 Exercises 498
9.8 Programming Assignments 500

10 Hashing 504
10.1 Hash Functions 505
10.1.1 Division 505
10.1.2 Folding 505
10.1.3 Mid-Square Function 506
10.1.4 Extraction 506
10.1.5 Radix Transformation 507
10.2 Collision Resolution 507
10.2.1 Open Addressing 508
10.2.2 Chaining 511
10.2.3 Bucket Addressing 514
10.3 Deletion 517
10.4 Perfect Hash Functions 518
10.4.1 Cichelli's Method 519
10.4.2 The FHCD Algorithm 521
10.5 Hash Functions for Extendible Files 524
10.5.1 Extendible Hashing 524
10.5.2 Linear Hashing 527
10.6 Case Study: Hashing with Buckets 530
10.7 Exercises 539
10.8 Programming Assignments 540

11 Data Compression 543
11.1 Conditions for Data Compression 545

11.2 Huffman Coding 545
11.2.1 Adaptive Huffman Coding 556
11.3 Shannon-Fano Code 559
11.4 Run-Length Encoding 561
11.5 Ziv-Lempel Code 562
11.6 Case Study: Huffman Method with Run-Length Encoding 565
11.7 Exercises 577
11.8 Programming Assignments 578

12 MEMORY MANAGEMENT 581
12.1 The Sequential-Fit Methods 582
12.2 The Nonsequential-Fit Methods 584
12.2.1 Buddy Systems 585
12.3 Garbage Collection 592
12.3.1 Mark-and-Sweep 595
12.3.2 Copying Methods 601
12.3.3 Incremental Garbage Collection 603
12.4 Concluding Remarks 610
12.5 Case Study: An In-Place Garbage Collector 611
12.6 Exercises 619
12.7 Programming Assignments 621

APPENDIXES
A Computing Big-O 627
A.1 Harmonic Series 627
A.2 Approximation of the Factorial Function 627
A.3 Big-O for Average Case of Quicksort 628
A.4 Average Path Length in a Random Binary Tree 631
B Algorithms in the Standard Template Library 633
B.1 Standard Algorithms 635

Name Index 643
Subject Index 647

Preface

The study of data structures is fundamental to computer science. Data structures serve as the foundation upon which many of the computer science fields are built. Some knowledge of data structures is a must for students who wish to do work in design, implementation, testing, or maintenance of virtually any software system. This text is presented in terms of the abstract data structures and algorithms in C++ that provides students with the necessary knowledge to perform such work.

This book highlights three important aspects of data structures. First, a very strong emphasis is placed on the connection between data structures and their applications, including applets and algorithms of complexity beyond data structures are presented in the object-oriented setting in accordance with the current design and implementation paradigm. In particular, the information hiding principle receives considerable attention throughout the text. Finally, a complete implementation of the book's data structures is provided. Specifically, the C++ language is used as the programming language.

The language C++ can be viewed as an extension of C, a widespread industry and academic standard. C++ supports many important object-oriented features, although it is not a pure object-oriented language. In addition, Pascal has been used to teach data structures, although Modula-2 and Ada have also been used. However, because of the widespread use of C++ in industry and the object-oriented characteristics of the language, using C++ to teach data structures and algorithms is a move that is well justified.

This book provides the material for two courses. It includes the topics tested on the CS1 and CS2 of the old ACM curriculum. It also covers the topics for most of the courses, e.g., CS7, e.g., CS8, and e.g., CS9 of the new ACM curriculum.

Most chapters include a case study that illustrates a complete solution which centers on a particular data structure or algorithm. These case studies are chosen from challenging problems in various areas such as computer graphics, scientific computing,

and file processing to indicate the wide range of applications to which topics under discussion may apply.

Brief examples of C++ code are included throughout the book to illustrate the practical importance of data structures. However, theoretical analysis is equally important, so presentations of algorithms are integrated with analyses of efficiency.

Great care is taken in the presentation of recursion because even advanced students have problems with it. Our experience has shown that recursion can be explained best if the run-time stack is taken into consideration. Changes to the stack are shown when tracing a recursive function not only in the chapter on recursion but in other chapters as well. For example, a surprisingly short function for tree traversal may conceal a trajectory of work done by the system on the run-time stack is not included in the explanation. Standing aloof from the system and retaining only a purely theoretical perspective when discussing data structures and algorithms are not necessarily helpful.

The thrust of this book is data structures, and other topics are treated here only as much as necessary to create a proper understanding of this subject. Algorithms are discussed from the perspective of data structures so that the reader will not find a comprehensive discussion of different kinds of algorithms and all the facets that a full presentation of algorithms requires. However, as mentioned, recursion is covered in depth. In addition, complexity analysis of algorithms is presented in some detail.

Chapters 1–8 present a number of different data structures and the algorithms that operate on them. The efficiency of each algorithm is analyzed, and improvements to the algorithms are suggested.

- Chapter 1 presents the basic principles of object-oriented programming, an introduction to dynamic memory allocation and the use of pointers, and a rudimentary presentation of the Standard Template Library (STL).
- Chapter 2 describes some methods used to assess the efficiency of algorithms.
- Chapter 3 presents different types of linked lists with an emphasis on their implementation with pointers.
- Chapter 4 presents stacks and queues and their applications.
- Chapter 5 contains a detailed discussion of recursion. Different types of recursion are discussed and a recursive call is dissected.
- Chapter 6 discusses binary trees, including implementation, traversal, and search. Balanced trees are also included in this chapter.
- Chapter 7 details more generalized trees such as tries, 2–4 trees, and B-trees.
- Chapter 8 presents graphs.

Chapters 9–12 show different applications of data structures introduced in the previous chapters. They emphasize the data structure aspects of each topic under consideration.

- Chapter 9 analyzes sorting in detail, and several elementary and nonelementary methods are presented.
- Chapter 10 discusses hashing, one of the most important areas in searching. Various techniques are presented with an emphasis on the utilization of data structures.

- Chapter [illegible] discusses data compression algorithms and data structures.
- Chapter [illegible] presents various techniques used to design external data structures for secondary memory.
- Appendix A discusses in greater detail Big-Oh notation, introduced in Chapter 2.
- Appendix B discusses standard aspects of the Standard Template Library.

Each chapter contains a discussion of the topics, illustrated with appropriate diagrams and tables. Except for Chapter 2, all chapters include complete code implementations. Chapter 2 includes a case study, whereas other chapters include code fragments that are more representative of real-world code. Exercises of varying degrees of difficulty follow each chapter. Chapters 3 and 4 provide pseudocode implementations of algorithms described in the text. Chapters 5 and 6 provide C++ implementations that are fairly complete. Chapters 7 and 8 describe algorithms at a high level that can be implemented in a variety of ways. Chapter 9 provides detailed implementations for some of the algorithms described in Chapter 8. Chapter 10 describes external data structures and algorithms and gives complete implementations.

Chapters 1–6 (excluding Sections 3.6, 3.7, 6.4.2, 6.7, and 6.8) contain the core material that should be the basis of any data structures course. These chapters should be studied in sequence. The remaining six chapters can be covered in any order. A one-semester course could include Chapters 1–6, Section 7.4, and Sections 10.1 and 10.2. The entire book could also be used in a two-semester sequence.

The source code for the complete programs in this book is available via the World Wide Web at the URL http://www.cs.fiu.edu/~weiss/dsaa_c++/code.gz

Changes in the Second Edition

The most significant development in recent years in C++ is the addition of the Standard Template Library (STL). The STL is a powerful collection of reusable components in the form of algorithms, containers, and iterators. It greatly simplifies programming by relieving the programmer from rewriting the most commonly used data structures. However, the STL components can be used properly only after a fair amount of background and well understood, slightly inefficient implementations of the same data structures. The role of this book is to supply the reader with such knowledge. For this reason, many data structures are rewritten in the book with no reliance on the STL. However, because the STL is now part of the C++ programming environment, the second edition of this book makes as much use of the STL as possible, in the sense that we describe how it can be used, and then we explain how the data structure works or how the STL implementation provided in the STL.

Furthermore, a new section of this book is devoted entirely to the STL. It is the focus of

- A gentle introduction to the STL in Section 1.7.
- Introductions to data structures and algorithms related components in the STL in sections 3.6, 3.7, 6.4.2, 6.7, 6.8, 7.7.6, 7.8, and 10.4.
- Appendix B.
- The use of the STL in case studies.

Other new material added to the book includes a discussion of amortized analysis in Section 2.9. The discussions of matching in Section 11.9 and Eulerian and Hamiltonian graphs in Sections 11.9 and 11.10 are also new.

The discussion of object oriented features of C++ is written anew in Chapter 1. The presentation of pointers in Section 1.4 and their application in implementing linked lists in Sections 3.1–3.2 is also rewritten. Chapter 4, presenting stacks and queues, is significantly simplified and includes a new case study. There are also numerous smaller modifications all through the book.

ACKNOWLEDGMENTS

I would like to thank the following reviewers, whose comments and advice helped me to improve this book:

J. David Bezek, University of Evansville

Paul Chu, Hawkeye State University

Jonathan Graham, Norfolk State University

Lonnie Harrison, Norfolk State University

Tai Chi Lee, Saginaw Valley State University

Martin Marshall, College Institute of Technology

Wilson Powell, Austin College

Thomas Wahl, University of Pittsburgh

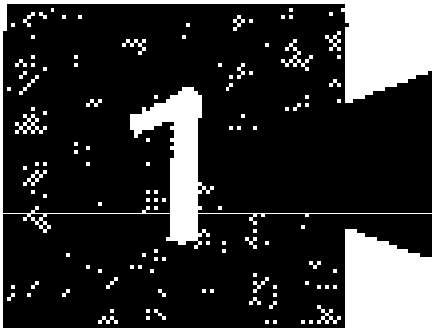
Xuecheng Tu, Southern Illinois University Edwardsville

However, the ultimate content is my responsibility, and I would appreciate hearing from readers about any shortcomings or strengths. My e-mail address is drozdek@duq.edu.

Adam Drozdek

Data Structures and Algorithms in C++

Object-Oriented Programming Using C++



1.1 Abstract Data Types

Before a program is written, we should have a fairly good idea how to accomplish the task being implemented by this program. Hence, an outline of the program containing its requirements should precede the coding process. The larger and more complex the project, the more detailed the outline phase should be. The implementation details should be delayed to the later stages of the project. In particular, the details of the particular data structures to be used in the implementation should not be specified at the beginning.

From the start, it is important to specify each task in terms of input and output. At the beginning stages, we should be more concerned with what the program should do, but how it should or could be done. Behavior of the program is more important than the gears of the mechanism accomplishing it. For example, a certain item is needed to accomplish some tasks. The item is specified in terms of operations performed on it rather than in terms of its inner structure. These operations may act upon this item, for example, modifying it, searching for some details in it, or storing something in it. After these operations are precisely specified, the implementation of the program may start. The implementation decides which data structure should be used to make execution most efficient in terms of time and space. An item specified in terms of operations is called an abstract data type. An abstract data type is not a part of a program. Solve a program written in a programming language requires the definition of a data structure, not just the operations on the data structure. However, an object-oriented language (OOL), such as C++, has a direct link to abstract data types by implementing them as a class.

1.2 Encapsulation

Object-oriented programming (OOP) revolves around the concept of an object. Objects, however, are created using a class definition. A class is a template in accordance to which objects are created. A class is a piece of software that includes data specification and functions operating on these data and possibly on the data belonging to other class instances. Functions defined in a class are called *methods*, member functions, or functions members, and variables used in a class are called data members or properties; they should be called instance members. This combining of the data and related operations is called data encapsulation. An object is an instance of a class, an entity created using a class definition.

In contradistinction to functional languages that are not object oriented, objects make the connection between data and member functions much tighter and more meaningful. In languages that are not object oriented, declarations of data and definitions of functions can be interspersed throughout the entire program, and only the program documentation indicates that there is a connection between them. In OOP, a connection is established right at the outset in that the program is based on this connection. An object is defined by related data and operations, and because there may be many objects used in the same program, the objects communicate by exchanging messages which reveal to each other as little as possible about their internal structure as necessary for adequate communication. Structuring programs in terms of objects allows us to accomplish several goals.

First, this strong coupling of data and operations can be used much better in modeling a fragment of the world which is emphasized especially by software engineering. Not surprisingly, OOP has its roots in simulation, that is, in modeling real world events. The first OOL was called Simula and it was developed in the 1960s in Norway.

Second, objects allow for easier error finding because operations are localized to the confines of their objects. Even if side effects occur, they are easier to track.

Third, objects allow us to conceal certain details of their operations from other objects so that these operations may not be adversely affected by other objects. This is known as the *information hiding principle* in languages that are not object oriented. This principle can be found to some extent in the case of local variables or even in local functions or procedures, which can only be used and accessed by the function defining them. This is, however, a very right holding or no holding at all. Sometimes we may need access, as in Pascal, a function *f2* defined in *f1* outside of *f1*, but we cannot. Sometimes we may need to access some data local in *f1* without exactly knowing the structure of these data, but we cannot. Hence, some modification is needed and this is accomplished in OOLs.

An object in OOP is like a watch. As users, we are interested in what the hands show, but not in the inner workings of the watch. We are aware that there are gears and springs inside the watch. But because we usually know very little about why all these parts are in a particular configuration, we should not have access to this mechanism so that we do not damage it, inadvertently or on purpose. This mechanism is

In short, encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically only the object itself can directly inspect or manipulate its fields.

An object is like a clock. Clocks have well-defined behaviors that can be manipulated by users knowing almost nothing about how they work internally. For example, you know that you can set a clock to a specific time by telling it the hour and minute. Internally, this might involve setting certain values in memory or sending certain signals to circuits, but you don't know or care about the internal mechanisms. Similarly, you know that you can ask a clock what time it is, but you don't know or care how it computes that answer.

Encapsulation is a powerful principle because it allows the implementation of an object to be changed without affecting the users of that object. For example, the internal representation of a clock could be changed from a digital display to an analog display without changing the way that users interact with the clock. Encapsulation is also important because it prevents users from tampering with the internal workings of an object, possibly leaving it in an inconsistent or erroneous state.

Encapsulation is implemented in C++ using the `class` keyword. A class is a user-defined type that can contain both data members and function members. The data members of a class are variables that hold the data for each object of the class. The function members, also called *methods*, implement the behavior of objects of the class. Collectively, the data members and function members of a class are referred to as *class members*. The function members of a class are the only functions that have direct access to the internal representation of objects of the class. Other functions can access the internal representation of an object only indirectly, by calling function members that do have direct access.

Information hiding is the principle that segregates the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. Typically, each data structure is hidden in exactly one module, and that module provides a limited interface to other modules that need indirect access to the information represented by the data structure. Modules are free to make whatever changes they like to the data structure, as long as the interface to other modules remains unchanged.

Encapsulation and information hiding are closely related concepts. Encapsulation is usually enforced by the language, and information hiding is a principle that programmers follow to make programs more maintainable. Encapsulation is a tool that supports information hiding. Information hiding is the principle that motivates encapsulation.

Encapsulation and information hiding are important principles in the design of large programs. Encapsulation helps to control the complexity of large programs by preventing programmers from relying on implementation details that may change. Information hiding helps to localize the effects of changes to the representation of data structures. Both principles help to make programs more robust and more maintainable.

```
class C {
public:
    C(char *s = "", int i = 0, double d = 0) {
        strcpy(dataMember1, s);
        dataMember2 = i;
        dataMember3 = d;
    }
    void memberFunction1() {
        cout << dataMember1 << " " << dataMember2 << " "
             << dataMember3 << endl;
    }
    void memberFunction2(int i, char *s = "Unknown") {
        dataMember2 = i;
```

```
        cout << "\n" << " received from " << a << endl;
    }
protected:
    char dataMember [10];
    int dataNumber;
    double dataProtect;
};

C:\object1.160.200C1. object2('object2'); object1
```

Message passing terminology is not universally used in object-oriented languages. However, to stress the fact that in C++ the terms object or class are not to be taken literally, this term is used. For example, the statement ... public.memberFunction(); ... object1.

```
object1.memberFunction();
```

is read as message memberFunction() sent to object1. Upon receiving the message, the object invokes its member function and displays relevant information. Message can include parameters as in

```
object1.memberFunction2(123);
```

in the message memberFunction2(123) is the parameter 123 passed to object1.

The basic principle is that messages are sent to the main program unit from which control is transferred to a member of another object. Therefore, the receiver of the message is identifiable, but not necessarily the sender. If object1 receives the message memberFunction1(), it does not know where the message originated. It only responds to it by displaying the information memberFunction1(). Capsulation terminology memberFunction(). Therefore, the sender may perform a lookup message to obtain the object identification as follows:

```
object_name.memberFunction2( 23, "myobject.2");
```

A powerful feature of C++ is the possibility of declaring generic classes by using type parameters in the class declaration. For example, if we need to declare a class that associates integers with strings, then we may declare the class as

```
class intClass {
    int associate[50];
    ..................
};
```

However, in this way, we limit the usability of the class to integers only. If we need a class that performs the same operations as intClass except that it operates on float numbers, then a new declaration is needed, such as

```
class floatClass {
    float storage[5];
    ..................
};
```

If a storage is to hold structures, or pointers to characters, then two more classes must be declared. It is much better to declare a generic class and decide to which type of items the object is referring only when creating the object. Fortunately C++ allows us to declare a class in this way and the declaration for the example is

```
template<class genType>
class genClass {
    genType *storage[50];
    ...................
};
```

Later we make the decision about how to initialize genType.

```
genClass<int> intObject;
genClass<Date> DateObject;
```

This generic class manifests itself in different forms according to the specific declaration. One generic declaration suffices for creating as many different forms.

We can go even further than that by not committing ourselves to 50 cells in a storage and by delaying that decision until the object definition is encountered. In most cases we only need to define values that be declared in two ways:

```
template<class genType, int size = 50>
class genClass {
    genType storage[size];
 ...................
};
```

The object definitions now:

```
genClass<int> intObject1;  // uses the default size;
genClass<int,100> intObject2;
genClass<Date,128> DateObject;
```

This method of using generic types is not limited to classes only, we can use them in function declarations as well. For example, the standard operation for swapping two values can be defined by the function

```
template<class genType>
void swap(genType &i1, genType &i2) {
    genType tmp = i1; i1 = i2; i2 = tmp;
}
```

This example also indicates the need for a specific function to operate on specific situations. If genType is a number, a character, or a structure, then the assignment operations is fine. But if genType is an array, then we can encounter a problem in assigning. The problem can be resolved by overloading the swap() operation by adding to it the function only required by a specific data type.

After a generic function has been declared, a program can use it as many times as it wants for as many different types as it wants. For example, if the compiler sees two calls

```
    swap(n,m); // swap two integers
    swap(x,y); // swap two floats
```

It is not necessary to explicitly overload the swap() function for each data type.

1.3 INHERITANCE

OOP is the process of creating new classes, called derived classes, from existing classes or abstract classes. The derived class inherits all of the capabilities of the existing class but can add refinements and new capabilities of its own. The following program illustrates inheritance:

```
// mar Reset mar [
public:
    BaseClass() { }
    void fighter fc = "baseevil" {
        cout << "Function f() in BaseClass called from " << x << endl;
        f();
    }
protected:
    void global fc = "baseevil" {
        cout << "Function g() in BaseClass called from " << x << endl;
    }
private:
    void h() {
        cout << "Function h() in BaseClass\n",
    }
};
// mar Derived Level  : public virtual BaseClass {
public:
    void fighter fc = "evilmoves" {
        cout << "Function f() in DerivedLevel1 called from " << x << endl;
        g("Derived Level 1");
        h("Derived Level 1");
    }
    void fighter fc = "evilmoves" {
        cout << "Function h() in DerivedLevel1 called from " << x << endl ;
    }
};
// mar Derived2Level  : pub ic virtual BaseClass {
public:
    void fighter fc = "evilmoves" {
        cout << "Function f() in DerivedLevel1 called from " << x << endl ;
        g("Derived2Level1");
//      h();  // error: BaseClass::h() is not accessible
    }
};
```

```
class DerivedLevel_2 : public DerivedLevel1, public DerivedLevel1 {
public:
    void f(char c = 'unknown') {
        cout << "Function f() in DerivedLevel2 called from " << a << endl;
        g("DerivedLevel_2");
        DerivedLevel_1::f("DerivedLevel2");
        BaseClass::f("DerivedLevel2");
    }
};
```

Example program:

```
int main() {
    BaseClass bc;
    DerivedLevel_1 d1;
    DerivedLevel1 d11;
    DerivedLevel2 d12;
    bc.f("main()");
//  bc.g(); // error: BaseClass::g() is not accessible
//  bc.h(); // error: BaseClass::h() is not accessible
    d1.f("main()");
//  d1.g(); // error: BaseClass::g() is not accessible
    d11.f("main()");
    d12.f("main()");
//  d12.g(); // error: BaseClass::g() is not accessible
//  d12.h(); // error: BaseClass::h() is not accessible
    d_2.f("main()");
//  d12.g();  // error: BaseClass::g() is not accessible
    d_2.h();
}
```

Program output:

```
Function f() in BaseClass called from main()
Function h() in BaseClass
Function f() in DerivedLevel_1 called from main()
Function g() in BaseClass called from DerivedLevel
Function h() in DerivedLevel_1 called from DerivedLevel1
Function f() in DerivedLevel_1 called from main()
Function f() in DerivedLevel_1 called from main()
Function g() in BaseC ass called from DerivedLevel_
Function f() in DerivedLevel2 called from main()
Function g() in BaseC ass called from DerivedLevel_2
Function g() in DerivedLevel1 called from DerivedLevel2
Function h() in BaseC ass called from DerivedLevel_2
Function h() in BaseC ass
Function g() in DerivedLevel1 called from main
```

In this example, the f() function is a member function of the base class called BaseClass. The derived classes DerivedLevel_1, DerivedLevel1, and DerivedLevel_2 inherit

their own base class members and member functions specified as baseClass::protected accessible. They inherit all these members from their base class so that they do not have to repeat the same definitions. However, a derived class can override the definition of a member function by redefining its own definition. In this way, both the base class and the derived class have some measure of control over their member functions.

The base class can decide which member functions and data members can be accessed by derived classes so that the principle of information hiding holds not only with respect to the rest of the base class but also to the derived classes. Moreover, the derived class can decide which parts of the public and protected member functions and data members to reveal, use, and which to redefine. For example, both `DerivedLevel1` and `DerivedLevel2` have their own versions of `Display`. However, the access to the member function with the same name in any of the classes higher up in the hierarchy is still possible by specifying the function with the name of the class and the scope resolution operator (::) as in `BaseClass::Display` in `DerivedLevel2`.

A derived class can also be used as a base class. A derived class can become a base class for further derived classes. In such inheritance, each derived class can be deliberately extended. For example, `DerivedLevel1` is derived from `BaseClass`, but then `DerivedLevel2` is derived from `DerivedLevel1`.

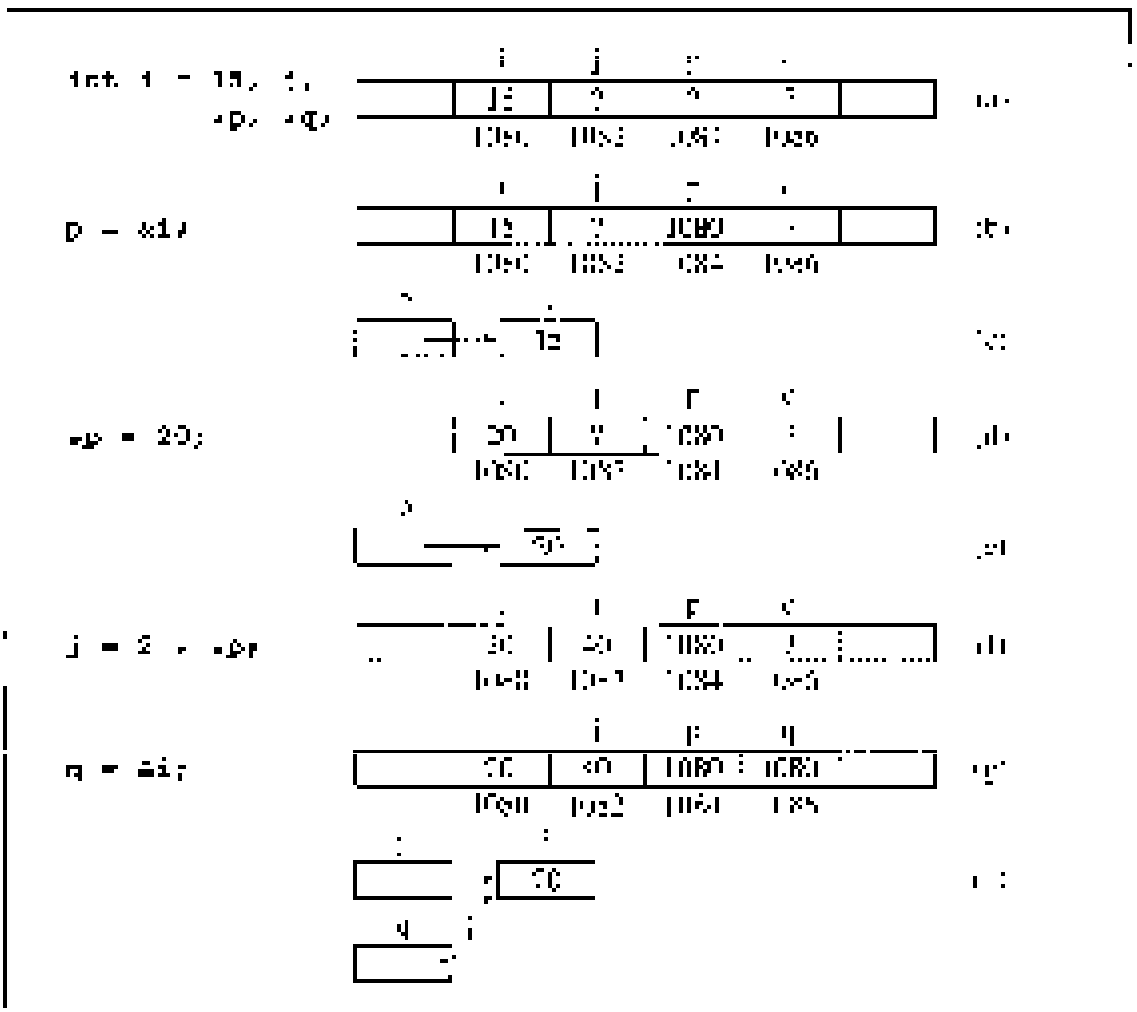
Inheritance in our examples is specified as public by using the reserved word `public` after the variable name. In general, the choice of a derived class' inheritance mode is `public`, `protected`, or `private`. In public inheritance, public members of the base class are also public in the derived class, and protected members are also protected. In this way, protected inheritance works the same as public inheritance, except that both public and protected members of the base class become protected in the derived class. Finally, in private inheritance, both public and protected members of the base class become private in the derived class. In all types of inheritance, private members of the base class are inaccessible to any of the derived classes. For example, an attempt to call `App` from `App` in `DerivedLevel1` causes a compilation error because `BaseClass::App` is not accessible. However, nothing from `App` in `DerivedLevel1` prevents a problem because it may call `BaseClass::App` defined in `DerivedLevel1`.

Protected members of the base class are accessible only to derived classes and not to nonderived classes. For this reason, both `DerivedLevel1` and `DerivedLevel2` can call `BaseClass`'s protected member function `App`, but a call to this member function using `pBaseClass` does not.

A derived class does not have to inherit from one base class only. It can be derived from more than one base class. For example, `DerivedLevel2` is derived as a class derived from both `BaseClass1` and `DerivedLevel1`, inheriting in this way all the member functions of `DerivedLevel1` and `BaseClass1`. However, `DerivedLevel2` inherits the same member functions from `BaseClass` twice because both classes used in the definition of `DerivedLevel2` are derived from `BaseClass`. This is redundant at best and at worst can cause a compilation error. The member function `BaseClass::App()` and `DerivedLevel1::App()` to prevent this from happening, the definition of this own class includes the modifier `virtual`, which means that `DerivedLevel2`

1.4 Pointers

Figure 1.7 Changes to variables via assignments are made using pointer variables. Note that p1 and p2 show the same situation and so do q1 and q2. Also, p1->i and (*p1).i are identical and refer to the same memory location.



In fact, pointers—like all variables—give you easy attribute encapsulation and data hiding. Data hiding means that only other variables in the same scope as a pointer can access the pointer.

Like any other data type, pointer variables can be assigned to pointers. Multiple pointers can refer to the same location that are accessible under different scopes and even different names. Pointers can also be used by their own name. These pointers must be set to point to memory that is allocated dynamically during program execution. This is unlike variables that are declared at program start-up and are allocated memory automatically.

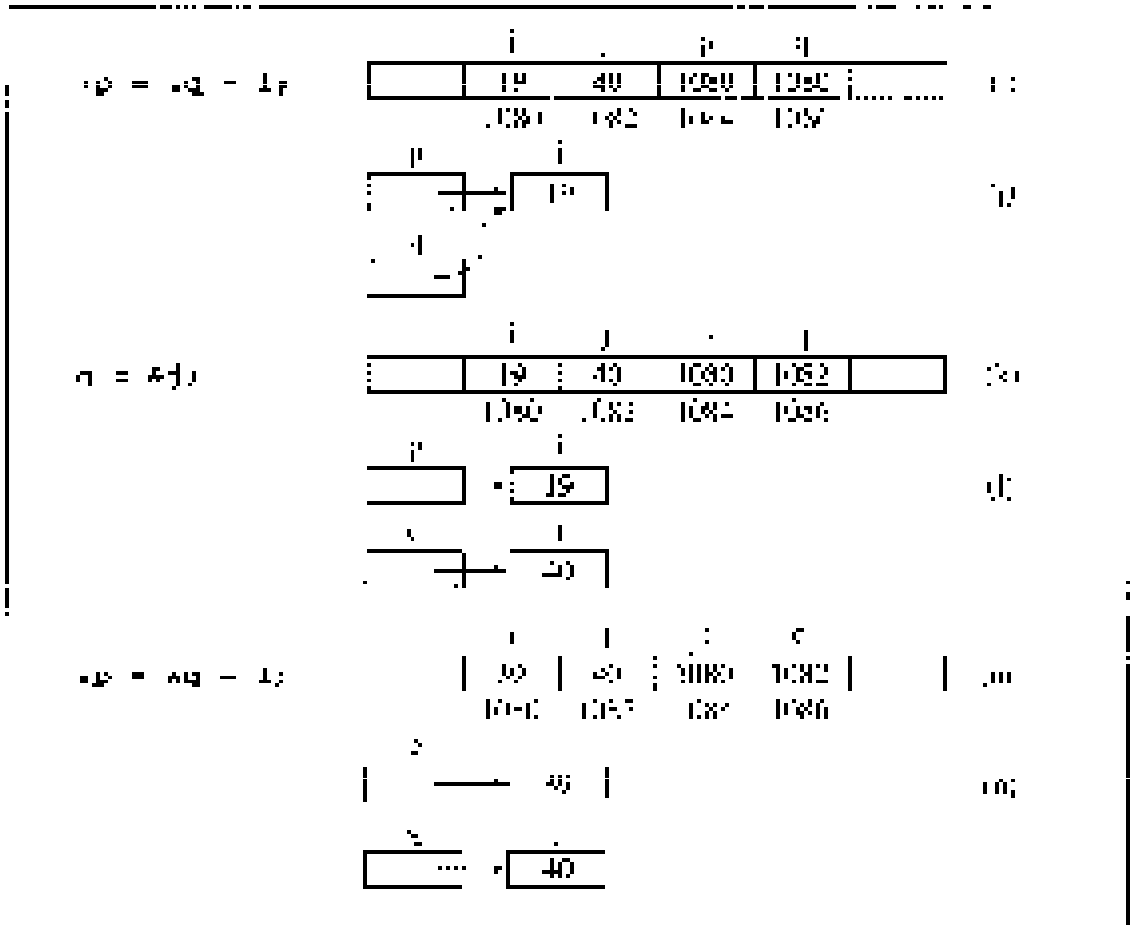
To dynamically allocate and deallocate memory, two C++ functions are used. These functions, new and delete, are operators and not standard functions as they are in C. The following new function, with the syntax form

```
p = new int;
```

tells the operating system to allocate memory to a pointer variable p. The new operator determines how much memory is required to hold the specified data type, finds a piece of memory of the correct size, and returns a pointer to that memory. Subsequently, the pointer p can be used to assign a value to the memory location:

```
*p = 5;
```

Figure 14.1 (continued)



to the memory block pointed to by p by dereferencing the pointer, either to get the value stored there or to store a value. When memory is allocated by calling malloc, the memory block is not initialized.

If the memory block is no longer needed, it can be returned to the operating system by passing the pointer to free. Memory that is not explicitly freed will be returned to the operating system when the program terminates.

```
free(p);
```

The call to free passes the pointer p to the operating system, indicating that the memory block it points to is no longer needed. The program can still access the memory, but it is illegal to read or write to memory that has been freed. Calling free has no effect on the pointer variable p; it still points to the memory block. In fact, the memory block is not changed in any way. It's the responsibility of the program to avoid using p unless it's assigned the address of a new memory block. Calling free causes the memory block to be recycled, so that it can be used by malloc to allocate a different memory block. If free is called more than once with the same pointer, the program may crash or behave erratically. Calling free with a null pointer is harmless.

problem. To avoid this problem, an address has to be assigned to a pointer. It cannot be the address of any variable; it should be a null address, which is usually 0. After execution of the assignment

```
p = 0;
```

we may not say that p points to null, but that it contains null or 0 is not.

1.4.1 Pointers and Arrays

In the last example, the parameter grade was a block of memory that holds one integer. A more interesting scenario is when a pointer refers to a data structure that is created and manipulated dynamically. This is a situation we would like to have to overcome the weaknesses imposed by arrays. Arrays in C++, and in other programming languages, have to be declared in advance. Their size must have to be known before the program executes. It is common that the programmer needs a fair knowledge of the problem to be programmed in order to choose the right size for the array. If the size is too big, then the array unnecessarily occupies many spaces in memory that basically wasted. If the size is too small, then we may run into problems. The alternative will be, however, to choose the size of the array to be very large, therefore, the decision is delayed until run time, when the right size may still be wasted or not.

The problem is solved with the use of pointers. Consider the following segment of code, which is legal in the IBM PC Windows environment under the IBM PC Windows environment, and which is illegal in the UNIX environment. The example is to access the value at a certain address (which is supplied by the user) and to add the size of an integer variable to the address. It should give us the address of the next integer. And this is essentially the way of traversing arrays.

Consider the following declarations:

```
int *p, *q;
```

The code segment specifies that p is a variable which is a pointer to an integer. The value of p is the address of the memory location where an integer is stored. The pointer is fixed; that is, it cannot be changed to point to another location. Suppose we execute the statement

```
p = q;
```

and

```
q++;
```

Now, the value of q is the address of the next integer. Pointer notation can be used to access cells of the array. For example, the array notation used in the loop which sums the numbers is

```
for (int i = 0; i < n; i++)
        sum += a[i];
```

can be replaced by pointer notation

```
for (int *p = a; p < a + n; p++)
        sum += *p;
```

array

```
[illegible]
```

Note that [illegible] a function of the [illegible] so that [illegible] supplied. First, [illegible] 1828, [illegible] 1021 but 1822 because [illegible] depends on [illegible] for example, [illegible]

```
[illegible]
[illegible]
```

and [illegible] 1858 [illegible] 1853 because [illegible] 1851 because [illegible] characters [illegible] 1070 [illegible] by [illegible] for [illegible]. The [illegible] [illegible] [illegible]

some where in memory. Assume that the function call is executed. The result is that copying is performed in that area outside of which, eventually, store will the program to crash.

1.4.2 Pointers and Copy Constructors

Some problems arise when pointers to class members are not handled properly when copy operations for objects are performed. Consider the following definitions:

```
class Node {
    char *name;
    int age;
    Node(char *n = "", int a = 0) {
      name = new char[strlen(n)+1];
      strcpy(name,n);
      age = a;
    }
};
```

The declaration of the declarations

```
Node node1("Roger",20); Node node2 = node1; // or node2 = node1;
```

is to create object node1, assign values to the two data members in node1, and then create object node2 and initialize its data members to the same values as in node1. These actions are to include allocating storage and assigning values to them. However, the assignment

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

the printing statement

```
cout<<node1.name<<" "<<node1.age<<" "<<node2.name<<" "<<node2.age;
```

generates the output

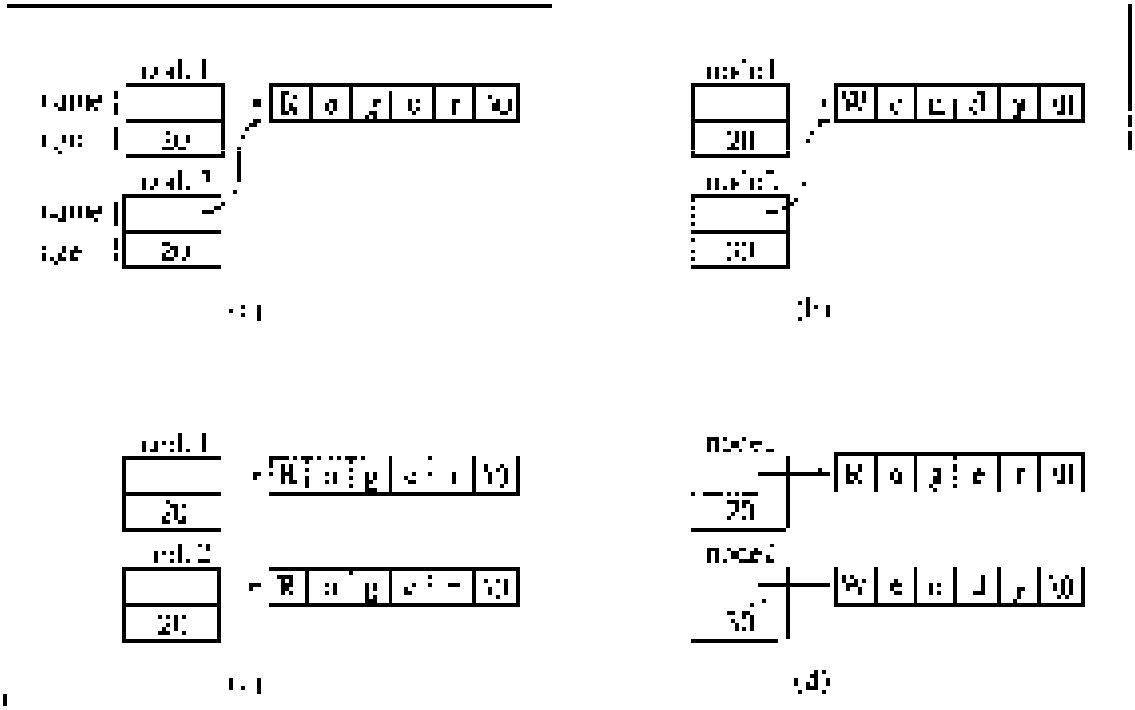
```
Wendy 30 Wendy 30
```

The problem is that the default version of Node does not provide a copy constructor

```
Node(const Node&);
```

which is necessary to execute the declaration node2 = node1 or node2(node1). If a user copy constructor is not provided, the compiler generates one automatically by the compiler. But the compiler-generated copy constructor performs member-by-member copying. Because name is a pointer, the copy constructor copies only the string address node1.name to node2.name, not the string contents. Thus, only one copy of the string exists. The declaration is as in Figure 1.4a. Now the assignments

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```



The memberwise assignment node2 = node1; leaves node2 with a bitwise copy of node1. Now both nodes point to the same memory location for their name strings. This is rarely what is intended. In this case, when node2 goes out of scope, the destructor will delete the memory pointed to by node2.name, leaving node1.name pointing to memory that is no longer allocated. This leads to run-time errors that are very difficult to find. To prevent this from happening, the user must define a copy constructor, such as

```
struct int_Node {
    char* name;
    int age;
    int_Node(char* n = 0, int a = 0) {
        name = new char[strlen(n)+1];
        strcpy(name, n);
        age = a;
    }
    int_Node(const int_Node& n) { // copy initialization:
        name = new char[strlen(n.name)+1];
        strcpy(name, n.name);
        age = n.age;
    }
};
```

With the new constructor, the declaration node2(node1) generates another copy of "Roger" pointed to by node2.name (Figure 1.2c), and the assignments to data members in one object have no effect on members in the other object, so the effect of the assignments

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

the object node1 remains unchanged, as illustrated in Figure 1.2c.

Note that a similar problem is raised by the assignment operator. If a definition of the assignment operator is not provided by the user, an assignment

```
node1 = node2;
```

performs member by member copying, which leads to the same problem as in Figure 1.2b. To avoid the problem, the assignment operator has to be overloaded by the user. For node, the overloading is accomplished by

```
Node& Node::operator=(const Node& n) {
    if (this == &n) return *this; // no need to assign to itself
        if (name != 0)
            delete [] name;
        name = new char[strlen(n.name)+1];
        strcpy(name,n.name);
        age = n.age;
    }
    return *this;
}
```

In this code, a special pointer this is used. Each object can access its own address through the pointer this so that *this is the object itself.

1.4.3 Pointers and Destructors

What happens to locally defined objects of type Node? Like all local items, they are destroyed in the sense that they become unavailable outside the block in which they are defined, and memory occupied by them is also released. But although memory occupied by an object of type node is released, not all the memory related to this object becomes available. One of the data members of this object is a pointer to a string; therefore, memory occupied by the pointer data member is released, but memory taken by the string is not. After the object is destroyed, the string previously available from its data member name becomes inaccessible (if not assigned to name of some other object or to a string variable) and memory occupied by this string can no longer be released. This is a problem with objects that have data members pointing to dynamically allocated locations. To avoid the problem, the class definition should include a definition of a destructor. A destructor is a function that is automatically invoked when an object is destroyed, which takes place upon exit from the block in which the object is defined or upon the call of delete. Destructors take no

arguments and return values is that there can be only one destructor per class and a destructor can be defined as

```
~Vector() {
    if (count == 0)
        delete [] items;
}
```

1.4.4 Pointers and Reference Variables

Consider the following declarations:

```
int i = 5, &ri = i, *pi = &i;
```

Variable `ri` is declared as being of type reference to int, and is initialized to refer to variable `i`. Variable `pi` is declared as being of type pointer to int, and is initialized to point to the address of variable `i`. A reference variable must be initialized to refer to a variable of the appropriate type when it is declared. It is not possible to change the variable to which a reference refers once it has been declared. This is in contrast to pointer variables, which can be made to point to different variables at any time. Thus, `ri` will always refer to `i`, but `pi` can be made to point to any variable of type pointer to int.

After the above declarations, the statement

```
cout << i << ri << *pi << pi << &ri << &pi << endl;
```

outputs `5 5 5 0x8f8 0x8f8 0x8f4`. After the assignment

```
ri = 7;
```

the same statement outputs `7 7 7`. After the assignment

```
*pi = 9;
```

the statement outputs `9 9 9`, and the assignment

```
i = 11;
```

leads to the output `11 11 11`. These statements indicate that an access of or assignment to a reference variable is equivalent to an access of or assignment to the variable to which the reference variable refers. Thus, it is not an accident that `ri` is always equal to `i`. Reference variables are usually declared as local variables or function parameters. They are rarely declared as class members. Pointer variables are declared as follows:

```
int *pi = 0;
```

Pointer variables are usually initialized to point to a variable of the appropriate type when they are declared. For example,

```
int *pi = &i;
```

where `i` is a variable of type int. The operator `&` is the address-of operator. It returns the address of its operand. Pointer variables can be made to point to different variables at different times. For example,

```
pi = &j;
```

where `j` is another int variable. Pointer variables can also be set to 0 or NULL, which points to no variable. For example,

```
pi = 0;
```

sets `pi` to point to no variable. Pointer variables are usually declared as local variables or function parameters. They are rarely declared as class members. However, pointer variables are frequently used in class member functions to point to variables passed to the function as parameters. For example,

```
void Vector::Copy(int *pi)
{
    for (int i = 0; i < count; i++)
        pi[i] = items[i];
}
```

copies the items in the vector to the array pointed to by `pi`. The statement

```
pi[i] = items[i];
```

is equivalent to

```
*(pi + i) = items[i];
```

```
int i = 1;
```

is acceptable if n is not a constant integer.

It is important to note the difference between the type `int&` (reference to int) and the type `const int` . The latter is a type of pointer to a constant integer:

```
const int *ip = &n;
```

after which the assignment

```
*ip = 5;
```

where n is an integer (whether constant or not) is impossible, but the assignment

```
ip = 2;
```

is permissible, even if n is not a constant.

Reference variables are used to pass up arguments by reference to functions (call-by-reference). Passing by reference is not limited to formal arguments that are pointers (which is the only way to pass arguments by reference in C). Any type of variable can be passed by reference, for example, a float declared as a reference type:

```
void f(int& i, float& x, int& k) {
    i = 1;
    x = 2;
    k = 3;
}
```

The values of the variables

```
int n1 = 0, n2 = 0, n3 = 0;
```

after execution of the call

```
f(n1, n2, n3);
```

are n1 = 1, n2 = 2, n3 = 3.

Reference types are also used in overloaded operator and type-conversion functions. For example, having defined the function

```
int& f(int a[], int n) {
    return a[n];
}
```

and the array

```
int a[] = {1, 2, 3, 4, 5};
```

we can use `f()` on the left side of the assignment operator. For instance, on the right-hand side,

```
n = f(a, 2);
```

or even on the left-hand side,

```
*(p+2) = 6;
```

will assign 6 to a[2] so that a = {1,2,6,4}. Note that we can accomplish the same with pointers, but dereferencing has to be used explicitly:

```
int* f(int a[], int i) {
    return &a[i];
}
```

and then

```
*f(a,3) = 6;
```

1.4.3 Pointers to Functions

As indicated in Section 1.4.1, one of the attributes of a variable is its address indicating its position in computer memory. The same is true for functions. One of the attributes of a function is the address indicating the location of the body of the function in memory. Upon function call, the system transfers control to this location to execute the function. For this reason, it is possible to use pointers to functions. These pointers are very useful in implementing functionals (that is, functions that use functions as arguments), such as integral.

Consider a simple function

```
double f(double x) {
    return 2*x;
}
```

With the definition, f is the pointer to the function f(); *f is the function itself, and (*f)(1); is a call to the function.

Consider now writing a function that computes the following sum:

$$\sum_{i=1}^{n} f(i)$$

To compute the sum, we have to supply not only limits n and m, but also a function f. Therefore, the desired implementation should allow not only for passing numbers as arguments, but also functions. This is done in C++ in the following fashion:

```
double sum(double (*f)(double), int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f(i);
    return result;
}
```

In this definition of sum(), the declaration of the first formal argument

```
double (*f)(double)
```

means that f is a pointer to a function with one double argument and a double return value. Note the need for parentheses around *f. Because parentheses have precedence over the dereference operator *, the expression

```
double *f(double)
```

represents a function that returns a double value.

The function sum1 can be called now with any built-in or user-defined double function that takes one double argument, such as

```
cout << sum(f,1,2) << endl;
cout << sum(sin,2,7) << endl;
```

Another example is a function that finds a root of a continuous function in an interval. The root is found by repetitively bisecting an interval and finding a midpoint of the current interval. If the midpoint is zero or the interval is smaller than some small value, the midpoint is returned. If the values of the function on the left limit of the current interval and on the midpoint have opposite signs, the search continues in the left half of the current interval; otherwise, the current interval becomes its right half. Here is an implementation of this algorithm:

```
double root(double (*f)(double), double a, double b, double epsilon) {
    double middle = (a + b) / 2;
    while (fmiddle) > 0 && fabs(b - a) > epsilon) {
        if (f(a) * f(middle) < 0)   // if f(a) and f(middle) have
            b = middle;             // opposite signs;
        else a = middle;
        middle = (a + b) / 2;
    }
    return middle;
}
```

1.5 POLYMORPHISM

Polymorphism refers to the ability of assuming many forms. In the context of OOP this means that the same function name denotes many functions that are members of different classes. Consider the following example:

```
class Class1 {
public:
    virtual void f() {
        cout << "Function f() in Class1\n";
    }
    void g() {
        cout << "Function g() in Class1\n";
    }
};
class Class2 {
public:
    virtual void f() {
```

```
        cout << "Function f() in Class2\n";
    }
    void g() {
        cout << "Function g() in Class2\n";
    }
};
class Class3 {
public :
    virtual void f() {
        cout << "Enter one of g() in Class3\n";
    }
};
void main() {
   Class1 object1, *p;
   Class2 object2;
   Class3 object3;
   p = &object1;
   p->f();
   p->g();
   p = (Class1*) &object2;
   p->f();
   p->g();
   p = (Class1*) &object3;
// p->f();  // Abnormal program terminate occurs;
   p->g();
// p->h();  // h() is not a member of Class3
}
```

The output of this program is as follows:

```
Function f() in Class1
Function g() in Class1
Function f() in Class2
Function g() in Class1
Function g() in Class1
```

As such, it can be explained that when pointer variable p points to object, of class type, it can call the virtual member functions that are defined in Class1. But after p becomes to point to object2 of class type Class2, then p->f() can call the function f() defined in Class2, whereas p->g() activates function defined in Class1. Here is the point of the difference between the normal with virtual member functions, the latter can be controlled.

In the case of the specified class hierarchy, the appropriate member function is executed depending on the type of the object to which the pointer variable points. Thus, the decision is delayed until the execution time. In this way, it is virtual function that is called, then the function corresponding to the object type pointed to by the pointer variable is

declaration. So at the type of value the pointer can only hold. Pointers to objects of different classes are not type compatible. Therefore, if p points to an object of class A, then p++ is illegal. There is no problem in the program when the actual function to be called is always resolved at compile time. This is known as static binding. Binding refers to the process used to convert identifiers (such as variable and function names) into machine addresses. Each time the identifier is encountered during compilation, its address is obtained from the symbol table. Binding is the process of converting identifier names into machine addresses. Static binding occurs when the address of an identifier is known before the program starts executing. Static binding results in efficient executable code. This is usually the case in procedural languages like C. However, in object-oriented programming, it is sometimes not possible to know the function to be called at compile time. This is known as late binding or dynamic binding. In late binding, the function name is associated with the machine address only at execution time. This is usually done with the help of pointers. Late binding requires some overhead to be incurred at execution time. However, late binding is a powerful concept in object-oriented programming. It allows the programmer to decide at execution time which function is to be invoked based on the type of object. Late binding is also known as dynamic binding. In C++, late binding is implemented using virtual functions. Static binding is the default binding used in C++. To declare a function as virtual, the keyword virtual is used in the function declaration in the base class. The resolution of a call to a virtual function is done at run time. At run time, the type of object pointed to by the pointer is determined and the appropriate function is called. Late binding requires some overhead in terms of memory and execution time. However, it provides flexibility in the design of programs. Late binding is usually implemented using a table of pointers to functions. This table is commonly referred to as the virtual table (vtable). Each class containing virtual functions has its own vtable. Each object created by such a class contains a pointer to the vtable for that class. Whenever a virtual function call is made, the vtable is used to resolve to the function address. This is much faster than searching a symbol table for the function name. Late binding ensures that the function code used is always the one defined in the class of the object pointed to by the pointer, irrespective of the type of the pointer. For example, if p is a pointer to objects of class Base, then p++ will always make p point to the next Base object in memory, irrespective of whether p actually points to an object of Base or of a class derived from Base. Late binding is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time. Late binding is more flexible than early binding. Late binding code usually runs slower than early binding code because of the overhead incurred. However, late binding is usually more powerful than early binding. Late binding is usually implemented using a table of addresses of functions. This table is usually

C++ uses a technical encapsulation facility which allows for well controlled information hiding. There is however a relaxation of this rule in the use of so called friend functions. The problem is that private information of a certain class cannot be accessed by anyone, not even public information is accessible to every user. But sometimes we would like to allow only some users to have access to the private portion of information. This can be accomplished if the class lists the user functions as its friends. For example, if the definition is

```
class C {
      int n;
      friend int f();
} ;
```

the function f() has direct access to variable n belonging to the class C, as in

```
int f ()
{   return 10 * sizeof( n ); }
```

This could be considered a violation of the information hiding principle; however, the class itself grants the right to make public in some cases what is private and inaccessible to others. Thus, since the user has control over what to consider a friend function, the friend function mechanism can be considered an extension of the information-hiding principle. This mechanism, which itself is used to facilitate programming and special operations since overriding does without using friend functions can be a major problem. Such a relaxation of some rules is, by the way, not uncommon in computer science, we can mention the existence of loops in functional languages such as LISP or sorting some information in the implementation of data files or relations of the relational database models as in dBaseIII+.

1.7 The Standard Template Library

C++ is an object-oriented language, but some extensions to the language bring C++ a higher level. The most significant extension to the language is the Standard Template Library (STL) developed primarily by Alexander Stepanov and Meng Lee. The library includes three types of objects: utilities, containers, iterators, and algorithms. Algorithms are frequently used functions that can be applied to different data structures. The application is mediated through the iterators that determine which algorithms can be applied to which types of objects. The STL receives requirements from within the container classes and provides a basis of functionality. Instead, they can use prepackaged, generic implementations made available by the implementation.

1.7.1 Containers

A container is a data structure that holds some objects, which are usually of the same type. Different types of containers organize the objects within them differently. Although the number of different containers is theoretically unlimited, only a small

number of them has practical significance and the most frequently used organizations are incorporated in the STL. The STL includes the following container classes: deque, list, map, multimap, set, multiset, stack, queue, priority_queue, and vector.

The STL containers are implemented as template classes that include member functions specifying what operations can be performed on the elements stored in the data structure specified by the container or on the data structure itself. Some operations can be found in all containers, although they may be implemented differently. The member functions common to all containers include the default constructor, copy constructor, destructor, empty(), max_size(), size(), swap(), operator=, and, except in priority queues, the overloaded relational operator functions (operator== etc.). Moreover, the member functions common to all containers except stacks, queues, and priority queues include the functions begin(), end(), rbegin(), rend(), erase(), and clear().

Elements stored in containers can be of any type, but they have to supply at least a default constructor, a destructor, and an assignment operator. This is particularly important for user-defined types. Some compilers may also require some relational operators to be overloaded (e.g., operator== and/or operator<) as well, even though the program does not use them. Also, a copy constructor and the assignment operator are specially provided if the members are pointers because insertion operations use a copy of an element being inserted, not the element itself.

1.7.2 Iterators

An iterator is a class used to traverse an element stored in a container. It is a generalization of the pointer. An iterator allows for accessing information included in a container so that desired operations can be performed on those elements.

As a generalization of pointers, iterators retain the same dereferencing notation. For example, *i1 gives the element referenced by iterator i1. Also, iterator arithmetic is similar to pointer arithmetic. Although all operations on iterators are not allowed in all containers.

No iterators are supported for the stack, queue, and priority_queue container classes. Iterator operations for classes list, map, multimap, set, and multiset are available. If i1 and i2 are iterators, it is acceptable to write

```
i1++, ++i1, i1--, --i1
i1 = i2
i1 == i2, i1 != i2
*i1
```

In addition to these operations, iterator operations for classes deque and vector are as follows:

```
i1 < i2, i1 <= i2, i1 > i2, i1 >= i2
i1 = a, i1 = b
i1 -= n, i1 += n
i1[n]
```

1.7.3 Algorithms

The STL provides some 70 generic functions, called algorithms, that can be applied to the STL containers and to arrays. A list of all the algorithms is in Appendix E. These algorithms are implementing operations that are very frequently used in most programs, such as locating an element in a container, inserting an element into a sequence of elements, removing an element from a sequence, modifying elements, comparing elements, finding a value based on a sequence of elements, sorting the sequence of elements, and so on. Most of the algorithms work on a half-open range of elements specified by two operands. The first operand refers to the first element of the range, and the second operand refers to an element after the last element of the range. This convention makes it easy to specify a range that includes all the elements in a container by using the functions `begin()` and `end()`. For example:

```
vector<int> v;
// Insert elements into v
sort(v.begin(), v.end());
```

sorts the elements in the container `v`. The first argument to `sort` specifies the beginning of the range, and the second argument specifies the end of the range. Note that `v.end()` refers to the position after the last element in the container. The function call

```
sort(v.begin(), v.end(), greater<int>());
```

sorts the elements in descending order. The third argument to `sort` is a function object that specifies the sorting criterion. The standard header file `<functional>` defines the function object `greater<int>`, which is used to compare two integers.

Algorithms usually work on iterators rather than containers, so they can be applied to any container that supports the required iterator operations. For example, the `sort` algorithm requires random access iterators, which are provided by vectors and deques. Algorithms that do not require random access iterators, such as `find`, can be applied to lists as well as vectors and deques.

1.7.4 Function Objects

In C++, a function object is an object that can be called as if it were an ordinary function. This is accomplished by defining a class that overloads the function call operator, `operator()`. For example, the following class overloads `operator()` to compute the cube of a `double` value:

```
class cube {
public:
```

```
        class2() {
        }
        double operator()(double x) {
            return 2*x;
        }
    };
```

and defining

```
    double sum2(class2 f, int n, int m) {
        double result = 0;
        for (int i = n; i <= m; i++)
            result += f(i);
        return result;
    }
```

which differs from sum() only in the first parameter which is a function object not a function (or, otherwise, in the name). The new function can now be called as in

```
    class2 c2;
    cout << sum2(c2,2,5) << endl;
```

or simply

```
    cout << sum2(class2(),2,5) << endl;
```

The latter way of calling requires a definition of constructor class2() (even if it is a no-body) to create a object of type class2; when sum2() is called.

The same can be accomplished without overloading the function call operator as exemplified in these two definitions:

```
    class class2 {
    public:
        class2() {
        }
        double run(double x) {
            return 2*x;
        }
    };
    double sum3(class2 f, int n, int m) {
        double result = 0;
        for (int i = n; i <= m; i++)
            result += f.run(i);
        return result;
    }
```

and then

```
    cout << sum3(class2(),2,5) << endl;
```

The STL relies very heavily on functor objects. The most common form of function pointer is instantiated for built-in operations. How could we pass sum() a pointer to

sum(1) The syntax sum(,2.5) is illegal. To circumvent the problem, the STL defines numerous <functional> function objects for the common C++ operators. For example, multiplies is defined as:

```
template<class T>
struct multiplies : public binary_function<T,T,T> {
    T operator()(const T& x, const T&
        return x*y;
    }
};
```

Now, after redefining function sum, you can benchmark a generic function:

```
template<class T>
double sum(T a, int n, int s) {
    double result = 0;
    for (int i = n-1; i >= 0; i--)
        result += a[i*s];
    return result;
}
```

This function can then be called with the previous function object.
sum(mega,vectorsize,stride,multiplies<double>());

1.8 VECTORS IN THE STANDARD TEMPLATE LIBRARY

The single most important container in the vector, which is a dynamic array that contiguous blocks of memory. Because memory locations are sequential, elements can be randomly accessed as fast as the access time of any element of the vector is constant. Storage is managed automatically so that as elements are inserted and the vector grows, a larger memory block is allocated for the vector, the elements are copied to the new block, and the old block is released. A vector is thus a flexible array that can grow and shrink as needed.

Figure 1.4 lists the header file of the vector template class. Most of the functions are self-explanatory and are described briefly in the comments. Figure 1.5 illustrates a few of the vector functions. Note that the program in Figure 1.5 only uses the integer functions; to use the program with other types, only a recompile is necessary.

To use the class vector, the program must include the header file vector:

```
#include <vector>
```

The class vector must be parameterized with the desired element type:

```
vector<int> v1(5);
```

Here the vector v1 is initialized to 5 elements:

```
vector<int> v2(3,1);
```

Figure 1.3 An alphabetical list of member functions for the class vector.

Member Function	Operation
`void assign(first, last)`	removes the nodes in the vector and inserts in their place the elements from the range denoted by iterators first and last.
`void assign(n, val = T())`	removes the nodes in the vector and inserts in their place n copies of val.
`T& at(n)`	returns the element in position n of the vector.
`value_t& at(n) const`	returns the element in position n of the vector.
`T& back()`	returns the last element of the vector.
`const T& back() const`	returns the last element of the vector.
`iterator begin()`	returns an iterator that references the first element of the vector.
`const_iterator begin() const`	returns a const iterator that references the first element of the vector.
`size_type capacity() const`	returns the number of elements that can be stored in the vector.
`void clear()`	removes all the elements in the vector.
`bool empty() const`	returns true if the vector includes no elements and false otherwise.
`iterator end()`	returns an iterator that references the last element of the vector.
`const_iterator end() const`	returns a const iterator that references the last element of the vector.
`iterator erase(i)`	removes the element in position i.
`iterator erase(first, last)`	removes the elements in the range indicated by iterators first and last.
`T& front()`	returns the first element of the vector.
`const T& front() const`	returns the first element of the vector.
`iterator insert(i, val = T())`	inserts a value into the element to be accessed by iterator i and returns an iterator referencing the newly inserted element.
`void insert(i, n, val)`	inserts n copies of val before the element referenced by iterator i.

but for vectors, the operation which is illustrated by `push_back()` is potentially a very costly operation.

Vectors grow automatically as new elements are inserted with, for example, `push_back()`. Adding a new element to a vector is usually fast unless the vector is full and it has to be copied to a new block. This situation occurs if the size of the vector equals its capacity. So, if the vector has some unused cells, `push_back()` can simply store a new

PROGRAM 1.3 (continued)

Member function	Operation
`void insert(i, Ertt, last)`	inserts elements from range [first,last) beginning at index i and just before the node referenced by iterator
`size_type max_size() const`	returns the maximum number of elements for the vector
`T& operator[]`	overloads operator []
`const T& operator[] const`	overloads operator []
`void pop_back()`	removes the last element of the vector
`void push_back(T&)`	inserts a new element at the end of the vector
`reverse_iterator rbegin()`	returns a reverse iterator that points to the last element of the vector
`const_reverse_iterator rbegin() const`	returns a const iterator that points to the last element of the vector
`reverse_iterator rend()`	returns a reverse iterator that points to the first element of the vector
`const_reverse_iterator rend() const`	returns a const iterator that points to before the first element of the vector
`void reserve(n)`	reserves enough room for the vector to hold n items if its capacity is not already that large
`void resize(n, T&)`	if n is smaller than the current size, the last size() - n elements will be discarded; otherwise, n - size() elements will be added to the end of the vector
`size_type size() const`	returns the number of elements in the vector
`void swap(v)`	swaps the current vector with the vector v
`vector()`	constructs an empty vector
`vector(... el = T())`	constructs a vector with a copy of el for all the elements (el defaults to the default constructor for type T)
`vector(iter, last)`	constructs a vector with the elements from the range specified by iterators first and last
`vector(v)`	copy constructor

constant immediately before the semicolon. The current values of the parameters can be tested with the functions size() that returns the number of elements currently in the vector and the function capacity() that returns the number of available cells in the vector. If necessary, the capacity can be changed with the function reserve(). For example, after executing

```
v.reserve(100);
```

Program 1.4 A program demonstrates implementation of generic vector functions

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // greater<T>

using namespace std;

template<class T>
void printVector(char *s, const vector<T>& v) {
    cout << s << " = ";
    if (v.size() == 0) {
        cout << "[]\n";
        return;
    }
    vector<T>::const_iterator i = v.begin();
    for(; i < v.begin()+v.size()-1; i++)
        cout << *i << ' ';
    cout << *i << "]\n";
}

bool f(int n) {
    return n > 4;
}

void main() {
    int a[] = {1,2,3,4,5};
    vector<int> v1;          // v1 is empty, size = 0, capacity = 0
    printVector("v1",v1);
    for (int j = 1; j <= 5; j++)
        v1.push_back(j);  // v1 = [1 2 3 4 5], size = 5, capacity = 8
    vector<int> v2(a,a+3); // v2 = [1 2 3]
```

vector v3 = [1 2] retains the same elements and the same size = 2, but its capacity changes from 2 to 0. The function resize() affects only the capacity of the vector, not its content. The function resize() affects content and possibly the capacity. For example, vector v4 = [1 2 3 4 5] of size = capacity = 5 changes after execution of

```
v4.resize(7);
```

to v4 = [1 2 3 4 5 0 0], size = 7, capacity = 10, and after another call to resize(),

```
v4.resize(2);
```

Figure 1.4 (continued)

```
    vector<int> v1(vec3.size() + 1, vec1.back()+1);
    vector<int> v2(v1.begin()+1, v1.end()-1);   // v2 = {2, 3, 4, 5, 6}
    cout << "v4 = " << v4 << endl;              // v4 = {1, 2, 3, 4, 5}, size = 5, capacity = 5
    cout << "v3 = " << v3 << endl;              // v3 = {2, 4, 6, 8, 10}
    cout << "v3[1] = " << v3[2] << endl;        // v3 = {2, 4, 6, 8, 10}
    cout << "v3.front() = " << v3.front() << endl;   // v3 = {2, 4, 6, 8, 10}
    cout << "v4.back() = " << v4.back() << endl;     // v4 = {1, 2, 3, 4, 5}, size = 5, capacity = 10
    cout << "v4.at(2) = " << v4.at(2) << endl;       // v4 = {1, 2, 3, 4, 5}, size = 5, capacity = 10
    cout << "v4.empty() = " << v4.empty() << endl;   // v4 is empty, size = 0, capacity = 10 (1)
    v4.insert(v4.end(), v3.begin(), v3.end());       // v4 = {3}
    v4.insert(v4.end(), v1.begin()+1, v1.end()-1);   // v4 = {3, 3, 4}
    v4.insert(v4.end(), 2, 4);                       // v4 = {3, 3, 4, 4, 4}
    v4.insert(v4.end(), v2.begin(), v2.end());       // v4 = {3, 3, 4, 4, 2, 3, 4}
    v4.erase(v4.end()-2);                            // v4 = {3, 3, 4, 4, 2, 4}
    v4.erase(v4.begin(), v4.begin()+4);              // v4 = {2, 4}
    v4.assign(3, 6);                                 // v4 = {6, 6, 6}
    v4.assign(vec3.rbegin(), vec3.rend());           // v4 = {10, 8, 6}
    vector<int>::reverse_iterator iter = v4.rbegin();
    for ( ; iter != v4.rend(); iter++)
        cout << *iter << " ";                        // prints: 2 4 1
    cout << endl;

//  elsewhere

    v1[0] = 3;                                       // v5 = {3, 9, 6, 9, 3}
    copy(vec_int.begin(), vec_int.end(), v5.begin()+1);   // v5 = {3, 6, 7, 8, 7}
    v1[0] = 3; v1[2] = v5[4] - 2;                    // v5 = {3, 6, 6, 8, 3}
    copy(deque1.begin(), deque1.end(), v5.begin());  // v5 = {3, 6, 7, 8, 7}
    sort(v5.begin(), v5.end());                      // v5 = {3, 7, 7, 8, 8}
    sort(v5.begin(), v5.end(), greater<int>());      // v5 = {8, 8, 7, 7, 3}
    v5.front() = 2;                                  // v5 = {2, 8, 7, 7, 3}
}
```

level = 1, size = 3, capacity = 10. This indicates that new space is allocated for vec_int, but it is not used.

Note that there is a member function pop_front(). This deletes the first element in a vector. In general, if you have a sequence container, you should use pop_front() instead of pop_back() if you want to process the elements in the order in which they were inserted. If you use pop_front(), all of the elements are moved by one position to make room for the new element. It is thus a time-consuming operation and can be a considerable waste. The function insert() (Table 1.4) is also a function which automatically allocates more memory for the vector if necessary. Other functions that increase the size of a vector are the constructor, function reserve(), and operator=.

Vector elements can be accessed with the subscript operator, as used for arrays, as in

```
v[2] = n;
```

or with iterators with dereferencing operators, used for pointers, as in

```
vector<int>::iterator is = v.begin();
*it = n;
```

Note that some member functions have no return type, that is, a reference type. For example, for an integer vector, the signature of member function `front()` is

```
int& front();
```

This means that, for example, `front()` can be used both on the left side and the right side of the assignment operator:

```
v.front() = 2;
v[1] = v.front();
```

Note that `front()` is an example of an overloaded member function because it can return a reference to a value or a constant reference to a value. To see the difference, consider these two assignments:

```
int& r1 = v.front();          // uses int& front();
const int& r2 = v.front();    // uses const int& front() const
```

Suppose that `v.front()` returns a value, say, a reference to a value, not with the declaration of `r1` but with the declaration of `r2`. To see the difference, consider the two assignments:

```
int r1 = v.front();
```

After the point, the value of `r1` is the same as the value of

```
v[0] = 1, r2 = v[0] = 2
```

But after the assignment

```
v.front() = 3;
```

the value of the first vector element is

```
v[0] = 3, r2 = v[0] = 3
```

All the STL algorithms can be applied to vectors. For example, the statement

```
replace(v.begin(), v.end(), 3, 5);
```

replaces all the 3s in vector `v` with 5s, so that the vector `v = {5, 2, 5}` becomes `v = {5, 2, 5}`, and the call

```
sort(v.begin(), v.end());
```

sorts the vector in ascending order, so that `v = {2, 5, 5}`. Using function pointers, but not objects, the programmer can define the comparison function.

```
bool f(int n) {
    return n < 4;
}
```

then the call

```
replace_if(v.begin(), v.end(), f, 7);
```

replaces by 7 every element of v that is less than 4. If the sequence v = {3, 9, 0, 5, 11, 4} is sorted, the way to accomplish the same result without the need for a predicate definition is to use

```
vec.erase(v.begin(), lower_bound(v.begin(), v.end(), 4));
```

In this expression, lower_bound(v.begin(), v.end(), 4) returns an iterator that indicates the first element of v that is greater than or equal to 4. Since the beginning of the vector is indicated by v.begin(), the expression v.erase(v.begin(), lower_bound(v.begin(), v.end(), 4)) erases all elements of v that are less than 4.

The sorting algorithm sort() can be used to sort the elements of a vector. For example, the vector v = {3, 9, 0, 5, 11, 4} is sorted in ascending order. Here is one way to sort it in descending order. One way is to sort it in ascending order and then reverse it. Another way is to first use sort() to sort it in ascending order and then reverse it. Yet another way is to first use sort() to sort it in ascending order and then use the following statement

```
sort(v.begin(), v.end(), greater<int>());
```

or more directly, use

```
sort(v.begin(), v.end(), f2);
```

where f2 is defined as

```
bool f2(int m, int n) {
    return m > n;
}
```

In this section we gave only a brief introduction to the STL. Element of objects are defined as template structures, which can be used to generate specific types of objects. For example, greater<int>() is the built-in template structure that can be used to generate a simple type of comparison function.

The version of the algorithm sort() we used takes two iterators to specify the range of elements to be sorted. We can also use sort() to sort objects of user-defined types. For example, to sort a vector of student objects in descending order of their ages, we can first define the student type as follows:

```
struct Person {
public:
    Person(char *n = "", int a = 0) {
        name = new char[strlen(n) + 1];
        strcpy(name, n);
        age = a;
    }
```

```
        bool operator==(const Person& p) const {
            return name.compare(p.name) == 0 && age == p.age;
        }
        bool operator<(const Person& p) const {
            return name.compare(p.name) < 0;
        }
        bool operator>(const Person& p) const {
            return name.compare(p.name) > 0;
        }
        private:
           char *name;
           int age;
           friend bool lessAge(const Person&, const Person&);
    };
```

Now, with this definition

```
    vector<Person> vp(1,Person("Gregg",55));
```

adding two more objects

```
    vp.push_back(Person("Ann",32));
    vp.push_back(Person("Bill",40));
```

and executing

```
    sort(vp.begin(),vp.end());
```

produces the error message "no match for 'operator<' in '__first.__gnu_cxx::__normal_iterator<_Iter, _Container>::operator*() < __last.__gnu_cxx::__normal_iterator<_Iter, _Container>::operator*()'" because the version of sort() called by this statement requires that the < operator be overloaded in class Person. The statement

```
    sort(vp.begin(),vp.end(),greaterPerson());
```

changes the order to "Ann", 32; "Bill", 40; "Gregg", 55 because the version of sort() called by this statement requires that the > operator be overloaded in class Person. What if we need to sort the objects by age? In this case, a function must be defined, such as

```
    bool lessAge(const Person& p1, const Person& p2) {
        return p1.age < p2.age;
    }
```

and then used as a third argument in the call to sort(), e.g.,

```
    sort(vp.begin(),vp.end(),lessAge);
```

which causes vp = {"Gregg", 55; "Bill", 40; "Ann", 32} to change to vp = {"Ann", 32; "Bill", 40; "Gregg", 55}.

1.9 Data Structures and Object-Oriented Programming

Although a template approach to data structures usually leads to a more efficient program, it does not lend itself to object-oriented programming style. In this style of programming, one designs a program around the data structures that are to be used. Operations that can be performed on these structures are then associated with each data type. A data type uses attributes as its building blocks rather than using data types as the building blocks. Such data types are already quite complex, since they encapsulate data structures and algorithms. Now data types are to be encapsulated into even more complex data types. The task given to the designer is to provide a mechanism for creating such new structures and investigate their behavior in terms of time and space requirements. Unlike the object oriented approach where we start with a basic structure and then create the most suitable data type which allows for an efficient performance of desirable operations, we now start with a data type specification and a fixed data structure and then describe what it can do, how it does it and how efficiently. The data structure used is designed to be a building block to be incorporated into classes by application programmers. Although using data structures that can perform certain operations efficiently and without increasing the memory burden on computer memory. It is also interested in building classes by concentrating on the mechanics of how classes are implemented. All of the most basic operations are not visible to the user of the classes. The data structures help most notably the extensibility of these classes and its improvement by modifying the data structure inside the classes, since it has direct access to them. It sharpens tools and advises the user as to what purposes they can be applied. Because of inheritance, the user can add some more operations to these classes and create sub-classes. From this point of view, the class designer did not have the data structures in mind. Therefore, these new operations can be tested by running and not by reading and re-writing the values of the variables, unless the user has access to the source code.

The data structures take advantage of the object-oriented facilities in this way it can build upon without the original. Here tools will be incorporated, reused in the application. By encapsulating the data structures it limits access to only making public only what is necessary for proper use of the class. The designer has tool to develop code whose functions are not compromised by unnecessary interference.

1.10 Case Study: Random Access File

Processes usually are designed to fulfill one of the two general categories of operation. The STL will be applied in these as it was in earlier chapters.

From the perspective of the operating system, files are collection of bytes, regardless of their contents. From the user's perspective, files are collections of records, numbers, and sequences, records and so on. The user wants to access the field of each

in a text file, a searching procedure goes sequentially through the file starting at position 0 and checks all of the bytes along the way. It counts the number of sequences of blank characters, and after it sees four such sequences (or five if a sequence of blanks begins the file), it knows because it encountered the beginning of the fifth nonblank sequence or the fifth word. This would enable it to begin at any position of the file. It is impossible to go to a particular position of any text file and be certain that this is a starting position of the fifth word of the file. Ideally, we want to go directly to a certain position of the file and be sure that the fifth word begins there. The problem is caused by the length of the preceding words and sequences of blanks. If we know that each word occupies the same amount of space, then it is possible to go directly to the fifth word by going to the position 4 * length-of-words. But because words are of various lengths, this can be accomplished and be assured only if the same number of bytes in each word is used. If a word is shorter, some padding characters are added to fill up the remaining space. If it is larger, then it would be truncated. In this way, a new organization is imposed on the file. The file is now thought of not merely as a collection of bytes, but as a collection of records, where each record consists of several words. If a request comes to access the fifth word, it would be directly accessed by looking at the preceding words, which have been previously converted into random access files.

A random access file allows for direct access of each record. The records usually include more than one word. The preceding paragraph suggests one way of creating a random access file, namely by using padding characters. Our task in this case study is to write a general program that will permit access and random access files for any type of records. The workings of the program are illustrated for a file containing personal records, each record consisting of five data members (social security number, name, city, year of birth, and salary), and for a random file that stores student records. The latter records have the same data members as personal records plus information about academic performance. Finally, we will create an inventory.

In this case study, a generic update record file must permit inserting a new record into a file, finding a record in the file, and modifying a record. The name of the file has to be supplied by the user, and if the file is not found, it is created; otherwise, it is opened for reading and writing. The program is shown in Figure 1.6.

The function findp determines whether a record is in the file. It performs the search register by comparing each stored record key to the sought record key and overloaded equality operator == . The function moves a current pointer in the file randomly by scanning up a record (or record, or byte by byte. To be sure, the records are built out of bytes and all manipulations happen on a particular record basis to be read, but only the basic read and write operations participating in the conversion.

The function modify updates information stored in a particular record. The record is first retrieved from the file, also using sequential search, and the new information is read from the keyboard and overloaded input operator >> . To store the updated record back in the file, modify positions the file pointer to the beginning of the record that has just been read; otherwise, the next record following that key in the file would be overwritten. The starting position of that can be determined unambiguously because each record occupies the same number of bytes; therefore, it is enough to jump back the number of bytes occupied by one record.

Figure 1.5 Listing of program to manage airline reservations.

```
// ********************** personal.h ***********************

#ifndef PERSONAL
#define PERSONAL

#include <stdlib.h>
#include <string.h>

class Personal {
public:
    Personal();
    Personal(char *, char *, int, int);
    void writeToFile(FILE *) const;
    void readFromFile(FILE *);
    void readKey();
    int size() const {
        return 0 + sizeof(name) + sizeof(phone) + sizeof(age) + sizeof(sex);
    }
    bool operator==(const Personal &) const {
        return strcmp(getKey(), "") == 0;
    }
protected:
    const int maxName, maxPhone;
    char *name, *phone, sex;
    int age;
    long lastKey;
    static void setKey(long);
    friend ostream &operator<<(ostream &, const Personal &);
    friend istream &operator>>(istream &, Personal &);
};

#endif

// ********************** personal.cpp ***********************

#include "personal.h"
```

Program 1.5 (continued)

```
Personal::Personal() : name(10), city(10) {
    name = new char[nameLen];
    city = new char[cityLen];
}

Personal::Personal(char *pSSN, char *n, char *c, int y, long s) :
        name(10), city(10) {
    name = new char[nameLen];
    city = new char[cityLen];
    strcpy(SSN,pSSN);
    strcpy(name,n);
    strcpy(city,c);
    year = y;
    salary = s;
}

void Personal::writeToFile(ofstream& out) const {
    out.write(SSN,9);
    out.write(name,nameLen);
    out.write(city,cityLen);
    out.write(reinterpret_cast<const char*>(&year),sizeof(int));
    out.write(reinterpret_cast<const char*>(&salary),sizeof(long));
}

void Personal::readFromFile(ifstream& in) {
    in.read(SSN,9);
    in.read(name,nameLen);
    in.read(city,cityLen);
    in.read(reinterpret_cast<char*>(&year),sizeof(int));
    in.read(reinterpret_cast<char*>(&salary),sizeof(long));
}

void Personal::readKeyin() {
    char s[10];
    cout << "Enter SSN: ";
    cin.getline(s,10);
    strncpy(SSN,s,9);
}

ostream& Personal::writeLegibly(ostream& out) {
    SSN[9] = name[nameLen] = city[cityLen] = '\0';
    out << "SSN:    " << SSN << ", name = " << name
```

FIGURE 1.5 (continued)

```
        << ", city = " << city << ", year = " << year
        << ", salary = " << salary;
    return out;
}

istream& Personal::readFromConsole(istream& in) {
    char c[80];
    cout << "SSN: ";
    in.getline(c,80);
    strncpy(SSN,c,9);
    cout << "Name: ";
    in.getline(c,80);
    strncpy(name,c,sizeof(name));
    cout << "City: ";
    in.getline(c,80);
    strncpy(city,c,sizeof(city));
    cout << "Birthyear: ";
    in >> year;
    cout << "Salary: ";
    in >> salary;
    in.getline(c,80); // get '\n'
    return in;
}

//-------------------------- student.h ---------------------------

#include "personal.h"

class Student : public Personal {
public:
    Student();
    Student(char *ssn, char *n, int, long int);
    void setProfile(fstream&) const;
    void readFromFile(fstream&);
    int size() const {
        return Personal::size() + sizeof(int);
    }
protected:
    char major[20];
    const char *getMajor() const;
    void setMajor(const char *);
```

LISTING 1.5 (continued)

```
    friend ostream& operator<<(ostream& out, Student& sr) {
        return sr.writeRegisterly(out);
    }
    istream& readFromConsole(istream&);
    friend istream& operator>>(istream& in, Student& sr) {
        return sr.readFromConsole(in);
    }
};

//////////////////////  student.cpp  ////////////////////

#include "student.h"

Student::Student() : majorLen(30) {
    Name();
    major = new char[majorLen];
}

Student::Student(char *fnam, char mn, char *ln, int y, long s, char *m) :
        majorLen(m ? : 0) {
    Name(fnam,mn,ln,y);
    sugar = new char[majorLen];
    strcpy(major,m);
}

void Student::writeToFile(ostream& out) const {
    Person::writeToFile(out);
    out.write(major,majorLen);
}

void Student::readFromFile(istream& in) {
    Person::readFromFile(in);
    in.read(major,majorLen);
}

ostream& Student::writeRegisterly(ostream& out) {
    Person::writeRegisterly(out);
    major[majorLen-1] = '\0';
    out << ", major = " << major;
    return out;
}
```

Figure 1.5 (continued)

```
istream& Student::readFromConsole(istream& in) {
    PersonalInfo::readFromConsole(in);
    char s[30];
    cout << "Major: ";
    in.getline(s,30);
    stuMap[major].s = s;
    return in;
}

//////////////////  Database  ////////////////////

#ifndef DATABASE
#define DATABASE

template<class T>
class Database {
public:
    Database();
    void run();
private:
    fstream database;
    char fName[20];
    ostream& print(ostream&);
    void add(T);
    bool find(const T&);
    vector<long> index;
    friend ostream& operator<<(ostream& os, Database& db) {
        return db.print(os);
    }
};

template<class T>
Database<T>::Database() {
    cout << "File name: ";
    cin >> fName;
}

template<class T>
void Database<T>::add(T d) {
    database.open(fName,ios::in|ios::out|ios::binary);
    database.seekp(0,ios::end);
```

Program 1.5 continued

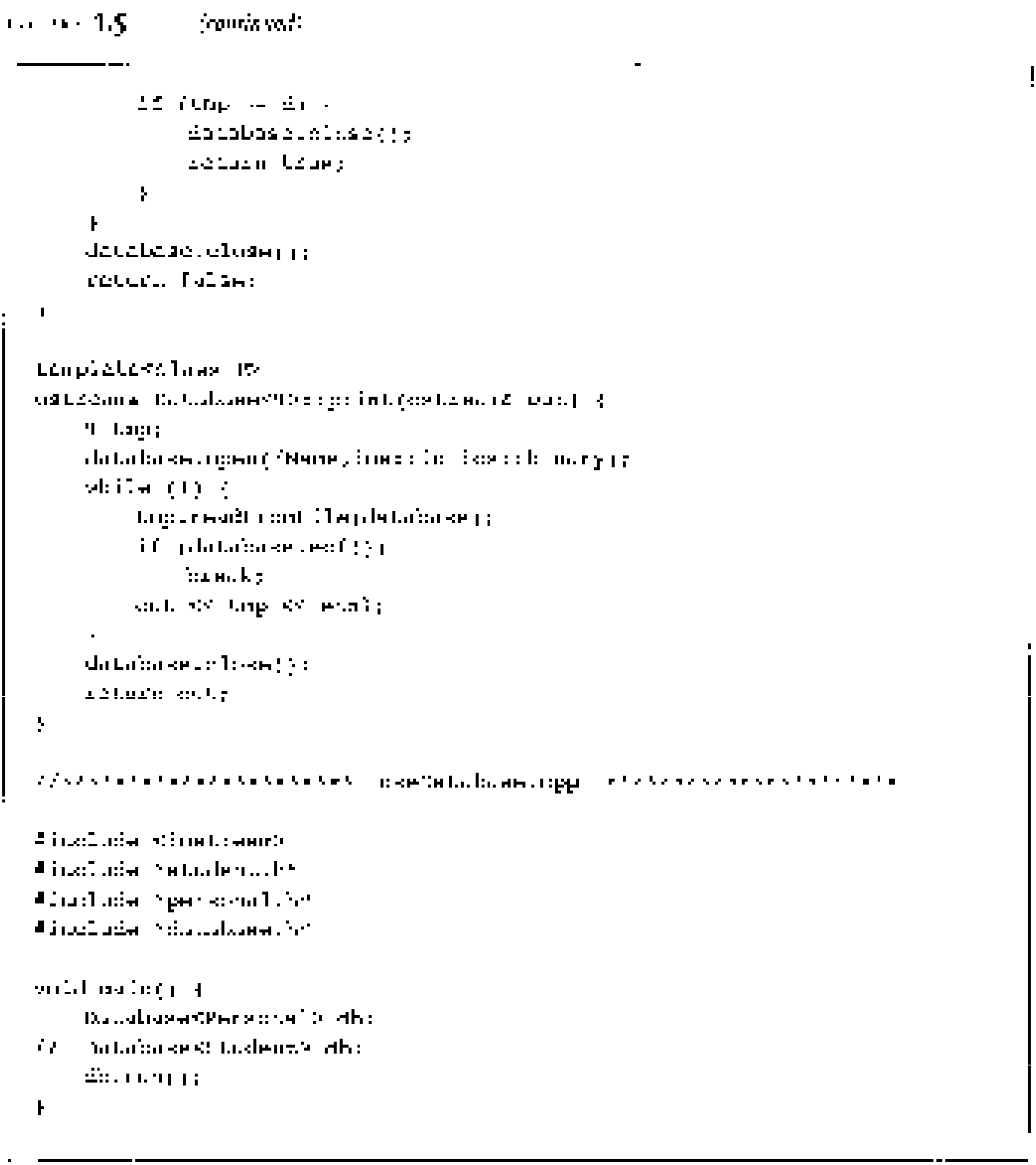
```
    database.writeToFile(database);
    database.close();
}

template <>
void Database<T> modify(const T& c) {
    T tmp;
    database.open(fName,ios::in|ios::out|ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);
        if (tmp == c) { // overloaded ==
            cin >> tmp; // overloaded >>
            database.seekp(-c.size(),ios::cur);
            tmp.writeToFile(database);
            database.close();
            return;
        }
    }
    database.close();
    cout << "The record to be modified is not in the database\n";
}

template <>
bool Database<T>::find(const T& c) {
    T tmp;
    database.open(fName,ios::in|ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);
        if (tmp == c) { // overloaded ==
            database.close();
            return true;
        }
    }
    database.close();
    return false;
}

template <>
ostream& Database<T>::print(ostream& out) {
    T tmp;
    database.open(fName, ios::in|ios::binary);
    while (1) {
```

Figure 1.5 (continued)

```
        tmp.seekg(recordNumber*sizeof(tmp));
        tmp.read(reinterpret_cast<char *>(&tmp), sizeof(tmp));
        if (tmp.getAccountNumber() != 0)
            break;
        out << tmp << endl; // overloaded <<
    }
    database.close();
    return out;
}

template<class T>
void DataBase<T>::main()
{
    char option='3';
    T rec;
    cout << "1. Add 2. Find 3. Add by a record; 4. Exit\n";
    cout << "Enter an option: ";
    cin.get(option).get(); // get option along with '\n'
    while (option.getline(option,4)) {
        if (option == '1') {
            cin >> rec;   // overloaded >>
            add(rec);
        }
        else if (option == '2') {
            rec.readAcct();
            cout << "the record is ";
            if (find(rec) == false)
                cout << "not ";
            cout << "in the database\n";
        }
        else if (option == '3') {
            rec.readAcct();
            modify(rec);
        }
        else if (option != '4')
            cout << "Wrong option\n";
        else return;
        cout << rec;   // overloaded <<
        cout << "Enter an option: ";
    }
}

#endif
```

Program 1.5 Continued

```
//------------------------ databaseDisp ------------------------

#include "database.h"

int main(void)
{
    database::displayAll();
    return 0;
}

void database::displayAll() {
    databaseLogin(fname, ios::in | ios::nocreate | ios::binary);
    databaseLoop(0, cout);
    close(fileDescriptor);
    databaseLogout();
}

void database::modify(const int id) {
    int stop;
    databaseLogin(fname, ios::in | ios::nocreate | ios::binary);
    while (databaseLoop()) {
        stop = recordExists(id);
        if (stop == id) {
            cout << stop;
            databaseLoop(-1, ios::out);
            updateRecord(id);
            databaseLogout();
            return;
        }
    }
    databaseLogout();
    cout << "The record to be modified is not in the database\n";
}

int database::findRecord(int id) {
    int stop;
    databaseLogin(fname, ios::in | ios::binary);
    while (databaseLoop()) {
        stop = recordExists(id);
```



[illegible] accomplished by writing data members of class type. Consider the following class definition:

```
class Person {
    char name[30];
    char city[10];
    int age;
    long salary;
public:
    Person();
    Person(char *, char *, int, long);
    void getData();
    void putData();
};
```

The program below creates two objects of type Person and writes the contents of the objects to a file. Then it reads the file and displays the contents of the file.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <iomanip.h>
#include <ctype.h>
#include <process.h>
#include "person.h"

int main()
{
    Person p1("Mary", "Boston", 25, 35000);
    Person p2("Sam", "Chicago", 30, 45000);
    ofstream outFile("PERSON.DAT", ios::out | ios::binary);
    outFile.write((char *) &p1, sizeof p1);
    outFile.write((char *) &p2, sizeof p2);
    outFile.close();
    ifstream inFile("PERSON.DAT", ios::in | ios::binary);
    if (!inFile) {
        cout << "File could not be opened" << endl;
        exit(1);
    }
    Person p;
    inFile.read((char *) &p, sizeof p);
    while (inFile.good()) {
        p.putData();
        inFile.read((char *) &p, sizeof p);
    }
    inFile.close();
    return 0;
}
```

The output is:

```
Name: Mary
City: Boston
Age: 25
Salary: 35000
Name: Sam
City: Chicago
Age: 30
Salary: 45000
```

Note that we cannot initialize a character array in the class definition:

```
class Person {
    char name[30] = "";   // illegal
    char city[10] = "";   // illegal
    int age;
    long salary;
public:
    Person();
    Person(char *, char *, int, long);
    void getData();
    void putData();
};
```

But this possible setback need not worry us. The default constructor in classes can be used to initialize variables.

Storing data from an object requires particular attention to the role of function `setfill()`. The SSN data member is a 9-character string. Social security number always includes nine digits. Leading zeros in the input representation should be used. However, for a single or multiple digit social security number, the output will be left-justified and padded with blanks. This is undesirable if the data file is designed for later processing. Therefore, we always want to store nine digits. To guarantee this, the function `setfill('0')` is used to set the fill character to '0'. Then, when outputting the two strings to the file, we specify a fixed number of characters (nine for the string including '\0', which is the default value of the operator `<<`).

Another problem is posed by the two numerical data members, `age` and `salary`. In particular, the alternative statement `salary = 35000;` will be treated by the compiler as an integer. Then the salary 35000 is written as a 4-byte integer, `strip`, `\x88\x00`, and the salary 45000 as a 4-byte long integer, `\x20\x02`, which is different than each record in the data file has a fixed length. One way to get around the problem is to store the numbers as strings. For example, `"35000"` is stored in the data member `salary` as a string of 32 bits `00000000000000000000000000000000`, assuming that `char` variables are stored in 8 bits. We can view this sequence of bits as representing not a long integer, but a string of five characters `00000000, 00000000, 00000000, 00000000, 00000000`. That is, the characters of the ASCII codes of the numbers 0, 0, 0, 0, and 0. In this way, regardless of the value of `salary`, the value is always stored in four bytes. This is accomplished with the statement

```
sprintf(salary, "%ld", 35000);
```

```
istream& readFromFile( istream& );
```

which first uses function `read` to read `sizeof( int )` bytes of input stream by converting the address `&accountNumber` to type `char *` and specifying the length of type `int`.

A similar approach is used to read data from a data file, which is the role of function `readFromFile`. Only the particular strings that should be stored in the record data members `lastName` and `firstName` are read from the file, so the function `readFromFile` is defined as

```
is.read( reinterpret_cast< char * >( &salary ), sizeof( double ) );
```

to get only the `salary` member. Although we know that `accountNumber` is stored first in each record, we do not read it because we want to illustrate how to seek to the `salary` member. The function `ignore` (with no arguments) is used to skip the `lastName` and `firstName` members.

This method of storing records in a data file presents a problem for input, however. If the user enters incorrect data, such as a string where a number is expected, the input stream fails. Once a stream fails, all subsequent stream operations are ignored, so it is not clear how to recover from the error. Function `fail` can be used to test for this, but it is not obvious how to position the file pointer so that the user can enter correct data for the current record.

To test the flexibility of the `Database` class, another user class is defined—class `Student`. This class also serves to show one more example of inheritance.

Class `Student` uses the same data members as class `Person` by being defined as a class derived from `Person` plus one more member, a string member `major`. Processing input and output for objects of class type `Student` is very similar to that for class `Person`, but the additional member has to be accounted for. This is done by redefining functions from the base class and reusing them. Consider the function `writeToFile` for writing a record in a data file in fixed length format:

```
void Student::writeToFile( ostream& out ) const {
    Person::writeToFile( out );
    out.write( major.c_str(), 15 );
}
```

The first parameter to `Person::writeToFile` is `out`, the file stream. Note that `Person::writeToFile` is called explicitly, even though `Student` is derived from `Person`. Next, the `write` function is called to write the string `major`. Function `write` requires the address of the first character to be written and the number of characters to write. The address of the first character in `major` is `major.c_str()`, and the number of characters is 15. Thus, the `major` string is stored in exactly fifteen bytes, padding with blanks if necessary.

1.11 Exercises

1. Identify the errors in the following declarations and initializations:

```
int p = &x;          float *q = &x;          double *r;
int *q = &x;         int **p = &q;           int *q = &r;
int p = &x;          float *q = &x;          double *r = &p;
int **p = &x;        int *q = &p;
```

2. Identify the errors in the following program segment. Assume that `x` has been declared and assigned a string:

```
class CharArray
{
    char *ch;
    int size;
public:
    CharArray(int);
    ~CharArray();
};
CharArray::CharArray(int s)
{
    size = s;
    ch = new char[size];
}
CharArray::~CharArray()
{
    delete ch;
}
```

3. Consider the following declaration:

```
int intArray[] = {1, 2, 3, 4, 5}, *p = intArray;
```

Which of the following will produce the same result as `intArray[3]` in every context?

a. `*(3);`

b. `*(p+3);`

c. `*(p+3*sizeof(int));`

4. Using only pointer notation, write a function that

a. Returns the smallest element in an integer array.

b. A function to remove all odd numbers from an ordered array. The array should remain ordered. Would it be easier to write this function if the array were unordered?

5. Using pointer only, implement the following string functions:

a. `strlen()`

b. `strcpy()`

c. `strcmp()`

d. `strcat()`

6. What is the difference between `if (p == q)` and `if (*p == *q)`?

7. Early versions of C++ did not support templates but did support classes and inheritance. Explain why templates are important. Could you implement templates without the use of such features?

8. What is the meaning of private, protected, and public parts of a class?

9. What should be the type of constructors and destructors listed in class?

10. Assume the following class declaration:

```
template<class genType>
class genClass {
   ...
   char aFunction(...);
   ... };
```

What is wrong with this function definition?

```
char genClass::aFunction(...) { ... };
```

11. Overloading is a powerful tool in C++, but there are some exceptions. What operators must not be overloaded?

12. If class B includes a private member, a protected variable, and a public member function, and class C is derived from class B, which of these variables can be used in class C? Can B become in class B private? protected? public? How about enabler and not? Does it make a difference whether the declaration of class C was private, protected, or public?

13. Transform the declaration

```
template<class genType, int size = 50>
class genClass {
    genType stack[size];
    ...................
    void someFunc() {
        ............
        if (sourVar < size) { ..... }
        ............
    }
};
```

which uses an integer variable size as a parameter to template in the declaration of genClass, which does not include size as a parameter to template, and yet allows for flexibility of the value of size. Consider a declaration of genClass's constructor. Is there any advantage of one version over another?

14. What is the difference between function members which are virtual and those which are not?

15. What happens if the declaration of genClass:

```
class baseClass {
    ...............
    virtual void process1(class1);
    virtual void process2(class2);
};
```

is followed by the declaration of derived class as:

```
class derivedClass : public baseClass {
    ...............
    void process1(int);
    int process2(char);
};
```

When member functions are invoked with the declaration of the pointers

```
baseClass *objectPtr1 = &derivedClass,
          *objectPtr2 = &derivedClass;
```

is followed by the statements as:

```
objectPtr1->process1(100);
objectPtr2->process2('A');
```

1.12 PROGRAMMING ASSIGNMENTS

1. Write a class called matrix which can be used to create two dimensional matrices of double type and perform the following operations on them: addition, subtraction, multiplication and transpose. Overload the operators +, -, * and = to perform the above operations. Write a driver program to test your class.

2. Write a class complex which can be used to perform complex number operations. Overload the operators +, -, * and = to perform the above operations. Write a driver program to test your class.

$$x = y + z^{2} = -1$$
$$\text{if } x_{ij} = \{ \} \; M = \{ y \} \; \text{and} \; \text{and} = \text{and} = y$$
$$x = \ln + \cdots + \sin(\cdots) = (y - q) \cdots = \infty;$$
$$= \text{in} = \text{by} \cdot \text{in} + \text{pow}(\cdots) \cdot \text{by} \cdot \text{pi} \cdot \text{in}$$
$$\{ \cdots \} \cdots = \{ \cdots \} \; \text{in} \cdots \; \text{by} + \text{by} \cdots =$$
$$= \{ \text{by} \cdots \text{by} \cdots \text{in} \cdots \} \{ \text{by} + \text{by} \cdots \} \cdots \text{in} \cdots$$
$$= \ln \cdots + \text{pi} \cdot \ln \cdots = \text{by} \{ \} = \ln \cdots + \text{by} \cdots + \text{by} \cdots = \text{by} \{ \}.$$

Use this equation to write a program using operation class.

3. Write a program to implement a spell checker for a file of text. The program should read in a dictionary of correctly spelled words and then read in the file to be checked. Any word in the file that is not in the dictionary should be printed, preceded by the line number on which it occurs. Words that are capitalized should be ignored. Make sure that your program is robust in the face of unexpected input.

4. Write a program that reads in a file of text and produces a concordance (table of contents) for it. A concordance is a table giving each word that occurs in the file and a list of the line numbers on which it occurs. Words that are capitalized should be ignored. Make sure that your program is robust in the face of unexpected input.

5. Write a program that reads in a file of text and produces a table of all pairs of words that occur in sequence. For example, the text

   So pure the sky, so quiet was the air!
   So like each day that was gone before,
   So like the days that were to be,
   So full of peace, we thought that care
   Was dead, and that it never would waken again.

   The 55-word concordance is as follows:

   ```
   1: a so quiet was like the
   1: air! so like each day
   1: it never would waken again.
   1: quiet was like the air!
   1: so pure the sky, so
   1: that it never would
   1: was dead, and that
   1: were to be, so full
   1: would waken again.
   2: be, so full of
   2: care was dead, and
   2: days that were to
   2: each day that was
   2: full of peace, we
   2: gone before, so like
   2: like each day that
   2: like the days that
   2: of peace, we thought
   2: pure the sky, so quiet
   2: sky, so quiet was
   2: so like the days
   2: so pure the sky,
   2: so quiet was like
   2: so like each day
   2: so like the days
   2: so pure the sky,
   2: thought that care was
   2: to be, so full
   2: was dead, and that
   2: we thought that care
   2: were to be, so
   2: would waken again.
   ```

   In this concordance, each word is followed by a list of up to five words that occurred in front of it. Each line is preceded by a count of the number of words that would be in the list if it were not limited to five. For example, the line "2: so like the days" indicates that the word "so" is followed by "like" in this poem in two cases. Note that punctuation marks are treated as part of the word.

   Write a program that reads in a concordance and a file of text and produces a version of the text in which each word is preceded by the number of times it occurs in the text. Then use this program to process the text of this exercise.

6. Modify the program from the previous exercise to produce a concordance in which up to five of the most recently encountered words are listed for each word in the text. The output should include the line number on which each word occurs. Common words such as "a", "an", "the", "and", "or", "but", "is", "are", "was", "were", "be", "been", "am", "of", "in", "on", "at", "to", "for", "with", "by", "from", "as", "that", "this", "it", "he", "she", "they", "we", "you", "I", "not", "no", "do", "does", "did", "have", "has", "had", "will", "would", "shall", "should", "can", "could", "may", "might", "must", and "ought" should be excluded from the table. Make sure that your program is robust in the face of unexpected input.

7. Write a program that reads in a file of text and produces a table of all pairs of words that occur within three words of each other in the text. Common words such as "a", "an", "the", "and", "or", "but", "is", "are", "was", "were", "be", "been", "am", "of", "in", "on", "at", "to", "for", "with", "by", "from", "as", "that", "this", "it", "he", "she", "they", "we", "you", "I", "not", "no", "do", "does", "did", "have", "has", "had", "will", "would", "shall", "should", "can", "could", "may", "might", "must", and "ought" should be excluded from the table. Make sure that your program is robust in the face of unexpected input.

8. Modify the program from the previous exercise to produce a table of all pairs of words that occur within n words of each other in the text, where n is a parameter to the program. Make sure that your program is robust in the face of unexpected input.

delete deletes one record. A null record can be assumed to begin with a null character (a character whose ASCII position is the first in order). The definition of the function remove in the database query class prototype is provided. The position of a record to be deleted and overwritten with the null record. After a session in which the database destructor should be invoked, which copies nonnull records to a new data file, deletes the old data file, and renames the new data file with the name of the old data file.

Bibliography

Lippman, Stanley B. *Inside the C++ Object Model*. Reading, MA: Addison-Wesley, 1996.

Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1998.

Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1997.

Wang, Paul S. *C++ with Object-Oriented Programming*. Boston: PWS, 1994.

Complexity Analysis

2.1 Computational and Asymptotic Complexity

The same problem can frequently be solved with algorithms that differ in efficiency. The difference between the algorithms may be one of the processing of a small number of data items, but these differences grow proportionally with the number of data. To compare the efficiency of algorithms, a measure of the dependency of the size of an algorithm called computational complexity was developed by Juris Hartmanis and Richard E. Stearns.

Computational complexity indicates how much effort is needed to apply a algorithm to a piece of data. The effort can be measured in a variety of ways and the particular context determines its meaning. This book concentrates on two of the most important measures: the time efficiency and the space efficiency. Time efficiency considerations usually focus on the amount of time elapsed when a process is applied to data. However, the most efficient algorithm run on a Cray computer can execute much slower than the most inefficient algorithm run on a PC, assuming all other system dependencies are identical. For example, to compare two algorithms, both of which would have to be run on the same machine. Furthermore, the results of such tests depend on the language in which a given algorithm is written. The tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in Turbo Pascal may run 20 times faster than the same program executed in BASIC or LISP.

To evaluate an algorithm's efficiency, real time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size of a file or an array and the amount of time required to process the data should be used. If there is a linear relationship between the size and the processing time, then doubling the size will double the time required. Conversely, if the size is cut in half, the processing time will also be cut in half. This is the ideal situation for algorithms. As the size of the computer

time by the same factor, if $x_2 = ax_1$, then $y_2 = ay_1$. Similarly, if $x_2 = \log_2 x_1$, then doubling x increases y by only one unit of time. Therefore, if $y_2 = \log_2 x_1$, then $x_2 = x_1^2$.

A benefit of expressing the relationship between x and y is that we study much more complex and calculating such a function is important only in regard to large values of data; any terms which do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, it usually is sufficiently close to the original, especially for a function that processes a large quantities of data. This measure of efficiency is called *asymptotic complexity* and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only a approximation can be found. To illustrate the first case, consider the following example:

$$f(x) = 3x^2 + 100x + \log_2 x + 1000 \tag{2.1}$$

For small values of x, the last term, 1000, is the largest. When x is 10, the second (1000) and last (1001) terms are roughly equivalent, with the other terms making a small contribution to the function's value. When x reaches the value of 100, the first and the second terms make the same contribution to the result, but when x becomes larger than 100, the contribution of the second term becomes less significant. Hence, for large values of x, due to the quadratic growth of the first term, $3x^2$, the value of the function is dependent mainly on the value of this term, as Figure 2.1 demonstrates. What terms can be disregarded in the equation?

2.2 Big-O Notation

The most commonly used notation for specifying asymptotic complexity, that is, for estimating the rate of function growth, is the *big-O* notation, introduced in 1894 by Paul Bachmann. Given two positive-valued functions f and g, consider the following definition:

Definition: $f(x)$ is $O(g(x))$ if there exist two positive numbers c and N such that $f(x) \leq cg(x)$ for all $x > N$.

This definition means that f is big-O of g if there is a positive number c such that $f(x)$ is never greater than $cg(x)$ for sufficiently large x, that is, for all x larger than some number N. The relationship between f and g can be expressed by stating either that $g(x)$ is an upper bound on the value of $f(x)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that first, technically there must exist certain c and N for which any two functions can be calculated. These constants, second, do not put any restrictions on these values and provide no guidance in situations where there are many candidates. In fact, there are usually infinitely many pairs of c and N that can be given for any two functions f and g. For example, for

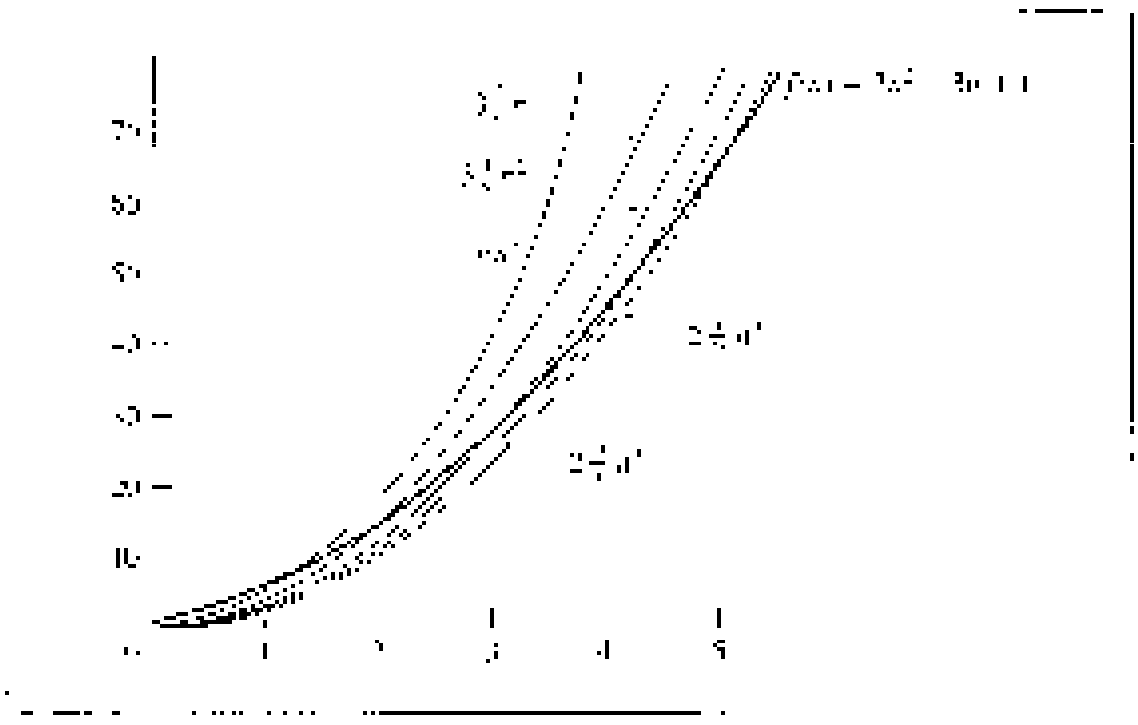
$$f(x) = 3x^2 + 2x + 1 = O(x^2) \tag{2.2}$$

we can say $g(x) = x^2$ or indeed any function of the form cx^2 where $c > 3$. Now, observe in Figure 2.2

Program 2.1 The growth rates of all common functions $f(n) = n^3 - 100n \log_{10} n - 1000$.

n	$f(n)$	n^3	$100n$		$\log_{10} n$		1000	
	Value	Value	%	Value	%	Value	%	Value	%
1	[illegible]	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Figure 2.3 Comparison of functions for different values of c and N from figure 2.2.



number of pairs of cs and Ns can be identified. The point is that f and g grow at the same rate. The definition states, however, that cg is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all n but not necessarily all n greater than or equal to some constant N. The trick of the matter is that the value of c depends on which N is chosen and vice versa. For example, if n is chosen as the value of N that is, if g is multiplied by one then $cg(n)$ will never be less than $f(n)$ for every n greater than or equal to 5. If $cg(n)$ is greater than or equal to $f(n)$ starting from $n = 2$, then it is sufficient that c is equal to 3. The constant c need not be an integer; it might be 1.5 as long as $cg(n)$ is not less than $f(n)$ starting from $n = 5$. Figure 2.3 shows the graphs of the functions f and g. The function g is positive with different coefficients c. Also N is always a point where the functions $cg(n)$ and f intersect each other.

The inherent imprecision of the big-O notation does not matter, since there can be an infinite number of functions g for a given function f. For example, the function $2n^2$ is big-O not only for n^2 but also for n^3, n^4, ..., n^k, ... for any $k \geq 2$. To avoid this embarrassment of riches, the smallest function g is chosen, n^2 in this case.

The approximation of function f can be either strengthened or weakened by the use of the notation, suppressing more or fewer terms. For example, for Equation 2.2, the contribution of the third and last terms to the value of the function can be omitted, and Equation 2.2 is

$$f(n) = n^2 + 100n + O(\log n) \tag{2.3}$$

Similarly, the function in Equation 2.2 can be approximated as

$$f(n) = 3n^2 + O(n) \tag{2.4}$$

2.3 Properties of Big-O Notation

Big-O notation has several helpful properties that are frequently used when analyzing the efficiency of algorithms.

Fact 1. (Transitivity) If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$. (This can be stated succinctly as $O(O(g(n))) = O(g(n))$.)

Proof: According to the definition, $g(n)$ is $O(h(n))$ if there are positive numbers M_1 and N_1 such that $g(n) \leq M_1 h(n)$ for all $n \geq N_1$. Similarly, $f(n)$ is $O(g(n))$ if there are positive numbers M_2 and N_2 such that $f(n) \leq M_2 g(n)$ for all $n \geq N_2$.

Observe that $f(n) \leq M_2 g(n) \leq M_2 M_1 h(n)$ for all $n \geq \max\{N_1, N_2\}$. (Here we take $c = M_2 M_1$ and $N = \max\{N_1, N_2\}$ when we use the definition of big-O.)

Fact 2. If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Proof: (Exercise 24 is equivalent to this fact. See the hint for Exercise 24.)

Fact 3. The function n^k is $O(n^m)$ if $k \leq m$.

Proof: (Exercise 25 is equivalent to this fact. See the hint for Exercise 25.)

Fact 4. The function n^k is $O(b^n)$ for any positive b.

Proof: (This is Exercise 26.)

It follows from all these facts that every polynomial $f(n)$ of degree d is $O(n^d)$. (This is Exercise 27.)

One of the most important facts about big-O notation is that it is possible to change the base of logarithms without affecting whether a function is $O(\log_b n)$. Thus, when we say that the logarithm of n is $O(\log n)$, we do not have to specify the base. Furthermore, the logarithm of n to any base is $O(n)$, even though the logarithm function grows more slowly than the identity function. We state these facts as the following theorem.

Fact 5. If $b > 1$, then $\log_b n$ is $O(\log_2 n)$.

Fact 6. For any positive $b \neq 1$, the function $\log_b n$ is $O(n)$.

The correspondence rule between logarithms to different bases is helpful to establish Fact 5. It says that $\log_b n = \log_a n / \log_a b$ for any positive $a \neq 1$. (This is Exercise 28.) Fact 6 follows from Fact 5 and the fact that $\log_2 n \leq n$ for all $n \geq 1$. (This is Exercise 29.)

Proof: By Fact 5, $\log_2 n$ is $O(\log_b n)$ and $\log_b n$ is $O(\log_2 n)$. By the definition of big-O notation, there are positive numbers M and N such that

$$\log_2 n \leq M \log_b n \quad \text{for all } n \geq N.$$

Taking the logarithm base 2 of both sides of the inequality, we obtain

$$\log_2 \log_2 n \leq \log_2 M + \log_2 \log_b n \quad \text{for all } n \geq N.$$

Thus

$$x \ln n = y \ln b,$$

$$\ln a \log_a n = \ln b \log_b n,$$

$$\log_a n = \frac{\ln b}{\ln a} \log_b n = c \log_b n$$

which proves that $\log_a n$ and $\log_b n$ are multiples of each other. So $\log_a n$ is $O(\log_b n)$ and $\log_b n$ is $O(\log_a n)$.

Because the base of the logarithm is irrelevant in the context of big-O notation, we can always use just one base and that base can be whatever we want.

Fact 2: $\log_a n$ is $O(\lg n)$ for any positive $a \neq 1$, where $\lg n = \log_2 n$.

2.4 Ω and Θ Notations

Big-O notation refers to the upper bounds of functions. There is a symmetric definition for a lower bound in the definition of big-Ω:

Definition 2: The function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

This definition reads: f is Ω (big-omega) of g if there is a positive number c such that f is at least equal to cg for almost all n. In other words, $cg(n)$ is a lower bound on the size of $f(n)$ or, in the long run, f grows at least as fast as g.

The only difference between this definition and the definition of big-O notation is the direction of the inequality: one definition can be turned into the other by replacing "≤" by "≥." There is an intimate connection between these two notations expressed by the equivalence

$$f(n) \text{ is } \Omega(g(n)) \text{ iff } g(n) \text{ is } O(f(n)).$$

Ω notation suffers from the same practical problem as does big-O notation. There is an unlimited number of choices for the constants c and N. For Equation 2.3, we are looking for such a c for which $2n^2 + 3n + 1 \geq cn^2$, which is true for any $c \leq 2$, where 2 is the limit borne in Figure 2.2. A set f is an $\Omega(g)$ set and $c \leq g$ then f is an Ω of g; that is, if for one such c and one g such that f is an Ω of g, then we can find infinitely many. For example, the function $2n^2$ is an Ω of n^2 and also of n, $n^{1.5}$, $n^{1.9}$, ... and also of $g = n \lg n$, $g = n \sqrt{n}$, ... and of many other functions. For practical purposes, only the closest Ω are the most interesting, the largest lower bounds. This restriction is made implicit such time we choose an Ω of a function f.

There are an infinite number of possible lower bounds for the function f. That is, there is an infinite set of g such that $f(n)$ is $\Omega(g(n))$, as well as an unbounded number of possible upper bounds of f. This may be somewhat disquieting, so we restrict our attention to the smallest upper bound and the largest lower bounds. Next, so, there is a common ground for big-O and Ω notations indicated by the equalities in

for definitions of these notations: Big-O is defined in terms of "≤" and Ω in terms of "≥"; "=" is included in both inequalities. This suggests a way of restricting the sets of possible lower and upper bounds. This restriction can be accomplished by the following definition of Θ (theta) notation:

Definition 3: $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge N$.

This definition reads: "For an order of magnitude g, f is on the order of g or both functions grow at the same rate in the long run. We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The only function just listed that is both big-O and Ω of the function 2.2 is n^2. However, n^2 is not the only choice and there are still an infinite number of choices, since the functions $2n^2$, $3n^2$, $4n^2$, . . . are also Θ of function 2.2. But it is rather obvious that the simplest, n^2, will be chosen.

When applying any of these notations (big-O, Ω, and Θ), do not forget that they are approximations that hide some detail which in many cases may be considered important.

2.5 Possible Problems

All the notations serve the purpose of comparing the efficiency of various algorithms designed for solving the same problem. However, if only big Os are used to represent the efficiency of algorithms, then some of them may be rejected prematurely. The problem is that in the definition of big O notation, f is considered big-O of g if the inequality $f(n) \le cg(n)$ holds in the long run for all natural numbers with a few exceptions. The number of ns violating this inequality is always finite. It is enough to meet the condition of the definition. As Figure 2.2 indicates, this number of exceptions can be reduced by choosing a sufficiently large c. However, this may be of little practical significance if the constant c in $f(n) \le cg(n)$ is prohibitively large, say 10^9, although the function g taken by itself seems to be promising.

Consider that there are two algorithms to solve a certain problem and suppose that the number of operations required by these algorithms is $10^9 n$ and $10n^2$. The first function is big-O of n and the second is big-O of n^2. Using just the big-O information, the second algorithm is rejected because it grows too fast. It is true that, again, in the long run, since for $n > 10^8$, which is 100 million, the second algorithm performs fewer operations than the first. Although 100 million is not an unheard of number of elements to be processed by an algorithm, in most cases the number is much lower, and in these cases the second algorithm is preferable.

For these reasons it may be desirable to use one more notation that includes constants which may be very large for practical reasons. Udi Manber proposes a double O (OO) notation to indicate such functions: f is $OO(g(n))$ if it is big-O of $g(n)$ and the constant is so large as to have practical significance. Thus, $10^9 n$ is $OO(n)$. However, the definition of "so large" depends on the particular application.

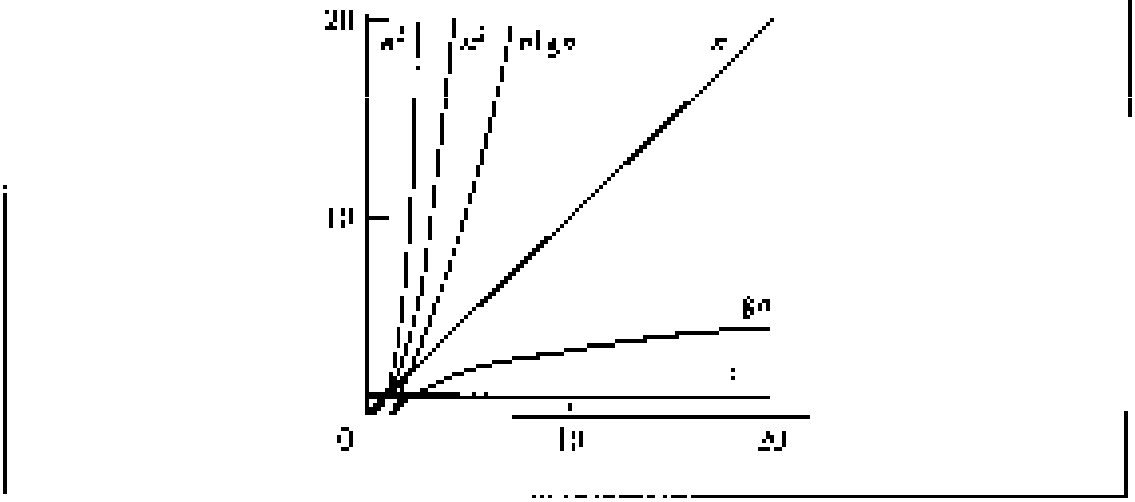
2.6 Examples of Complexities

Algorithms can be classified by their time or space complexities and in this respect several classes of such algorithms can be distinguished, as Figure 2.4 illustrates. Their growth is also displayed in Figure 2.5. For example, an algorithm is called *constant* if its execution time is the same for any number of elements; it is called *quadratic* if its execution time is $O(n^2)$. For each of these classes, a number of operations is shown, along with the real time needed for executing them on a machine able to perform 1 million operations per second or one operation per microsecond (μsec). The scale in Figure 2.4 indicates that some ill-designed algorithms, or algorithms whose complexity cannot be improved, have no practical application on available computers. To process 1 million items with a quadratic algorithm, over 11 days are needed.

Figure 2.4 Classes of algorithms and their execution times on a computer executing 1 million operations per second (1 sec = 10^6 μsec = 10^3 msec).

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\log n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \log n)$	$O(n \log n)$	33.2	33 μsec	664	664 μsec	9973	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	3.17×10^{17} yrs	10^{301}	
n		10^4		10^5		10^6	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\log n)$	13.3	13 μsec	16.6	17 μsec	19.93	20 μsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \log n)$	$O(n \log n)$	133×10^3	133 msec	166×10^4	1.6 sec	199.3×10^5	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	31.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		$10^{30,103}$		$10^{301,030}$	

FIGURE 2.5 Typical functions applied in big-O estimates.



and for a cubic algorithm, thousands of years. Even if a computer can perform one operation per nanosecond (1 billion operations per second), the quadratic algorithm finishes in only 16.7 seconds, but the cubic algorithm requires over 31 years. Even a 1000-fold improvement in execution speed has very little practical bearing for this algorithm. Analyzing the complexity of algorithms is of extreme importance and cannot be overlooked on account of fast computers. What we have termed as extremely little work a computer such as your desktop can execute millions of operations per second. The importance of analyzing the complexity of algorithms in any context but in the context of data structures in particular cannot be overestimated. The impressive speed of computers is not limited use if the programs that run on them use inefficient algorithms.

2.7 FINDING ASYMPTOTIC COMPLEXITY: EXAMPLES

Asymptotic bounds are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed. This section illustrates how this complexity can be determined.

In most cases, we are interested in time complexity, which usually measures the number of assignments and comparisons performed during the execution of a program. Chapter 9, which deals with sorting algorithms, considers both types of operations; this chapter considers only the number of assignment statements.

Begin with a simple loop to calculate the sum of numbers in an array:

```
for(i = sum = 0; i < n; i++)
    sum += a[i];
```

First, two variables are initialized. Then the `for` loop iterates n times, and during each iteration, it executes two assignments, one of which updates `sum` and the other updates `i`. Thus, there are $2 + 2n$ assignments, so the complete run of this `for` loop in its asymptotic complexity is $O(n)$.

Complexity usually grows if nested loops are used, as in the following code, which computes the sums of all the subarrays that begin with position 0:

```
for(i = 0; i < n; i++) {
    for(j = 1, sum = a[0]; j <= i; j++)
        sum += a[j];
    cout<<"sum for subarray 0 through "<< i <<" is "<<sum<<endl;
}
```

Before the loops start, `i` is initialized. The outer loop is performed n times, executing in each iteration an inner `for` loop, print statement, and assignment statements for `j`, `sum`. The inner loop is executed i times for each $i \in \{1, \ldots, n-1\}$ with two assignments in each iteration: one for `sum` and one for `j`. Therefore, there are $1 + 3n + 2\sum_{i=1}^{n-1} i = 1 - 3n + 2(1 + 2 + \cdots + n-1) = 1 + 3n + n(n-1) = O(n) + O(n^2) = O(n^2)$ assignments executed before the program is completed.

For nested loops, complexity usually grows in comparison with one loop, but it does not have to grow at all. For example, we may request printing sums of numbers in the last five cells of the subarrays starting in position 0. We adapt the foregoing code and transform it to

```
for(i = 4; i < n; i++) {
    for(j = i-3, sum = a[i-4]; j <= i; j++)
        sum += a[j];
    cout<<"sum for subarray "<<i-4<<" through "<< i <<" is "<<sum<<endl;
}
```

For each i, the inner loop is executed only four times; for each iteration of the outer loop, there are 11 assignments. This number does not depend on the size of the array. The program makes $1 + 11(n-4) = 11n - 43 = O(n)$ assignments, although the use of nested loops suggests something else.

Analyses of these two examples is relatively uncomplicated because the numbers of times the loops executed did not depend on the ordering of the arrays. Computations of asymptotic complexity is more involved if the number of iterations is not always the same. This point can be illustrated with a loop used to determine the length of the longest subarray with the numbers in increasing order. For example, in `| 1 3 2 5 9 11 12 |`, 1 is three, the length of subarray `| 1 2 5 |`. The code is

```
for(i1 = 0, longest = 1; i1 < n-1; i1++) {
    for(i2 = i1 + 1; i2 < n-1 && a[i2] < a[i2+1]; i2++);
    if(longest < i2 - i1 + 1)
      longest = i2 - i1 + 1;
}
```

Notice that if all numbers in the array are in decreasing order, the outer loop is executed $n-1$ times, but in each iteration, the inner loop executes just one time. Thus, the

algorithm is $O(n)$. The algorithm is least efficient if the numbers are in increasing order. In this case, the outer `for` loop is executed $n - 1$ times, and the inner loop is executed i times for each $i = 1, \ldots, n - 1$. Thus, the algorithm is $O(n^2)$. In most cases, the arrangement of data is less orderly, and measuring the efficiency in these cases is of great importance. However, it is far from trivial to determine the efficiency in the average cases.

A fifth example used to determine the asymptotic complexity is the *binary search* algorithm, which is used to locate an element in an ordered array. If a is an array of numbers and we try to locate number k, then the algorithm accesses the middle element of the array first. If that element is equal to k, then the algorithm returns its position; if not, the algorithm continues to the second trial, at which half of the original array is considered. The first half, if k is smaller than the middle element, and the second otherwise. Now the middle element of the chosen subarray is accessed and compared to k. If it is the same, the algorithm completes successfully. Otherwise, the subarray is divided in two halves, and if k is smaller than this middle element, the first half is discarded; otherwise, the first half is retained. This process of halving and comparing continues until k is found or the array can no longer be divided in two nonempty subarrays. This relatively simple algorithm is encoded as follows:

```
template<class genType>  // Overloaded operator< is used;
int binarySearch(const genType a[], int arrSize, const genType& key) {
    int lo = 0, mid, hi = arrSize-1;
    while (lo <= hi) {
          mid = (lo + hi)/2;
          if (key < a[mid])
             hi = mid - 1;
          else if (a[mid] < key)
             lo = mid + 1;
          else return mid;  // guaranteed: return the index of
    }                       //    the null array with key;
    return -1;              // failure: key is not in the array;
}
```

If `key` is in the middle of the array, the loop executes only one time. How many times does the loop execute in the case where `key` is not in the array? First the algorithm looks at the entire array of size n, then at one of its halves of size $\frac{n}{2}$, then at one of the halves of this half of size $\frac{n}{4}$, and so on, until the array is of size 1. Hence, we have the sequence $n, \frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^k}$, and we want to know the value of k. But the last term of this sequence is equal to 1, from which we have $n = 2^k$, so the fact that k is not in the array can be determined after $\lg n$ iterations of the loop.

2.8 The Best, Average, and Worst Cases

The last two examples in the preceding section indicate the need for distinguishing at least three cases for which the efficiency of algorithms has to be determined. The worst case is when an algorithm requires a maximum number of steps, and the first loop is

when the number of steps is the smallest. The average case falls between these extremes. In simple cases, the average complexity is established by considering possible inputs to an algorithm, determining the number of steps performed by the algorithm for each input, adding the number of steps for all the inputs, and dividing by the number of inputs. This definition, however, assumes that the probability of occurrence of each input is the same, which is not always the case. To consider the probability explicitly, the average complexity is defined as the average over the number of steps executed when processing each input weighted by the probability of occurrence of that input, or:

$$C_{avg} = \sum_i p(input_i) * cost(input_i)$$

This is the definition of expected value which assumes that all the possibilities can be determined and that the probability distribution is known, which simply determines the probability of occurrence of each input, $p(input_i)$. The probability function p satisfies two conditions: it is never negative, $p(input_i) \geq 0$, and all probabilities add up to 1, $\sum_i p(input_i) = 1$.

As an example, consider searching sequentially an unordered array to find a number. The best case is when the number is found in the first cell. The worst case is when the number is in the last cell or is not in the array at all. In this case, all the cells are checked to determine this fact. And the average case? We may make the assumption that there is an equal chance for the number to be found in any cell of the array; that is, the probability distribution is uniform. In this case, there is a probability equal to $\frac{1}{n}$ that the number is in the first cell, a probability equal to $\frac{1}{n}$ that it is in the second cell, ..., and finally a probability equal to $\frac{1}{n}$ that it is in the last, nth cell. This means that the probability of finding the number after one try equals $\frac{1}{n}$, the probability of finding it after two tries equals $\frac{1}{n}$, ..., and the probability of finding it after n tries also equals $\frac{1}{n}$. Therefore, we can average all these possible numbers of tries over the number of possibilities and conclude that it takes on the average

$$\frac{1 + 2 + \ldots + n}{n} = \frac{n+1}{2}$$

steps to find a number. But if the probabilities differ, then the average over gives a different answer. For example, if the probability of finding a number in the first cell equals $\frac{1}{2}$, the probability of finding it in the second cell equals $\frac{1}{4}$, and the probability of finding it in any of the remaining cells is the same and equal to

$$\frac{1}{n-2} \cdot \frac{1}{4} = \frac{1}{4(n-2)}$$

then, on the average, it takes

$$\frac{1}{2} + \frac{2}{4} + \frac{3 + \ldots + n}{4(n-2)} = 1 + \frac{n(n+1) - 6}{8(n-2)} = 1 + \frac{n+3}{8}$$

steps to find a number, which is approximately four times better than $\frac{n+1}{2}$ found previously for the uniform distribution. Note that the probabilities of accessing a particular cell have no impact on the best and last cases.

The complexity for the three cases was relatively easy to determine for sequential search, but usually it is not that straightforward. Particularly, the complexity of the average case can pose difficult computational problems. If the computation is very complex, approximations are used, and that is where we find the big-O, Ω, and Θ notations most useful.

As an example, consider the average case for binary search. Assume that the size of the array is a power of 2 and that a number in it to be searched has an equal chance to be in any of the cells of the array. Binary search can locate it either after one try in the middle of the array, or after two tries in the middle of the first half of the array, or after two tries in the middle of the second half, or after three tries in the middle of the first quarter of the array, or . . . or after three tries in the middle of the fourth quarter, or after four tries in the middle of the first eighth of the array, or . . . or after four tries in the middle of the eighth eighth of the array, or . . . or after $\lg n$ tries in the first cell, or after $\lg n$ tries in the third cell, or . . . or, finally, after $\lg n$ tries in the last cell. That is, the number of all possible tries equals

$$1 + 2 \cdot 2 + 1 \cdot 3 + 3 \cdot 4 + 1 \cdot 5 + \ldots + \frac{n}{2} \lg n = \sum_{i=1}^{\lg n} i \cdot 2^{i-1}$$

which has to be divided by n to determine the average over n cells. What is this sum equal to? We know that it is between 1 (the best case result) and $\frac{1}{2} n \lg n$ (the worst case) determined in the preceding section. But is it closer to the best case—say, $\frac{1}{4} n \lg n$—or to the worst case—for instance, $\frac{1}{2} n \lg n$? The sum does not lend itself to a simple conversion into a closed form; therefore its estimation should be used. Our conjecture is that the sum is not less than the sum of powers of 2 in the specified range multiplied by a half of $\lg n$, that is,

$$s_1 = \sum_{i=0}^{\lg n - 1} 2^i (i+1) \geq \frac{\lg n}{2} \sum_{i=0}^{\lg n - 1} 2^i = s_2$$

The reason for this choice is that s_2 is a power series multiplied by a constant factor, and that can be presented in closed form very easily, namely,

$$s_2 = \frac{\lg n}{2} \sum_{i=0}^{\lg n - 1} 2^i = \frac{\lg n}{2} \cdot \frac{2^{\lg n} - 1}{2 - 1} = \frac{\lg n}{2} (n - 1)$$

which is $\Omega(n \lg n)$. Because s_2 is the lower bound for the sum s_1 under scrutiny—that is, s_1 is $\Omega(s_2)$—then so is s_1, the lower bound of the straight average case complexity t_1—that is, $t_1 = \Omega(s_1)$. Because s_1 is $\Omega(n \lg n)$, so must be t_1. Because $\lg n$ is an assessment of the complexity of the worst case, the average case's complexity equals $\Theta(\lg n)$.

There is still one unresolved problem: Is $s_1 \geq s_2$? To determine this, we conjecture that the sum of each pair of terms positioned symmetrically with respect to the center of the sum s_1 is not less than the sum of the corresponding terms of s_2. That is,

$$2^1 + 1 + 2^{\lg n - 1} \lg n \geq 2^1 \frac{\lg n}{2} + 2^{\lg n - 1} \frac{\lg n}{2}$$

$$2^1 \cdot 2 + 2^{\lg n - 2} (\lg n - 2) \geq 2^1 \frac{\lg n}{2} + 2^{\lg n - 2} \frac{\lg n}{2}$$

...

$$2^1 (j+1) + 2^{\lg n - j - 1} (\lg n - j) \geq 2^1 \frac{\lg n}{2} + 2^{\lg n - j - 1} \frac{\lg n}{2}$$

...

where $j = \frac{\lg n}{2} - 1$. The last inequality, which represents every other inequality, is transformed into

$$2^{\lg n - 1} \cdot \left(\frac{\lg n}{2} - j \right) \geq 2^1 \left(\frac{\lg n}{2} - j - 1 \right)$$

and then into

$$2^{\lg n - j - 1} \geq \frac{\lg n - j - 1}{\frac{\lg n}{2} - j} = 2 + \frac{1}{\frac{\lg n}{2} - j} \qquad (2.5)$$

All of these transformations are allowed because all the terms that moved from one side of the conjectured inequality to another are nonnegative and thus do not change the direction of inequality. Is the inequality true? Because $j = \frac{\lg n}{2} - 1$, $2^{\lg n - j - 1} \geq 2$, and the right-hand side of the inequality (2.5) is always less than 4, the conjectured inequality is true.

This concludes our investigation of the average case for binary search. The algorithm is relatively straightforward, but the process of finding the complexity for the average case is rather grueling, even for uniform probability distributions. For more complex algorithms, such calculations are significantly more challenging.

2.9 Amortized Complexity

In many situations, data structures are subject to a sequence of operations rather than one operation. In this sequence, one operation possibly performs extensive modifications that have an impact on the run time of the next operation in the sequence. One way of assessing the worst-case run time of the entire sequence is to add worst-case complexities for each operation. But this may result in an excessively large and unrealistic bound on the sequence's run time. To be more realistic, amortized analysis can be used to find the average complexity of a worst-case sequence of operations. By analyzing sequences of operations rather than isolated operations, amortized analysis takes into account interdependencies between operations and their results. For example, if an array is sorted and only a very few new elements are added, then resorting this array should be much faster than sorting it for the first time because

after the new additions, the array is nearly sorted. Thus it should be quicker to put all elements in perfect order than in a completely disorganized array. Without taking this correlation into account, the run time of the two sorting operations can be considered twice the worst-case efficiency. Amortized analysis on the other hand, discloses that the second sorting is hardly applied in the worst case situation so that the combined complexity of the two sorting operations is much less than double the worst-case complexity. Consequently, the average for the worst case sequence of sorting, a few insertions, and sorting again is lower according to amortized analysis than according to worst case analysis, which disregards the fact that the second sorting is applied to an array operated on already by a previous sorting.

It is important to stress that amortized analysis is analyzing sequences of operations, or if single operations are analyzed, it is done in view of their being part of the sequence. The cost of operations in the sequence may vary considerably, but how frequently particular operations occur in the sequence is important. For example, for the sequence of operations $op_1, op_2, op_3, \ldots$ the worst case analysis renders the computational complexity for the entire sequence equal to

$$C(op_1, op_2, op_3, \ldots) = C_{worst}(op_1) + C_{worst}(op_2) + C_{worst}(op_3) + \ldots$$

whereas the average complexity determines it to be

$$C(op_1, op_2, op_3, \ldots) = C_{avg}(op_1) + C_{avg}(op_2) + C_{avg}(op_3) + \ldots$$

Although specifying complexities for a sequence of operations, neither worst case analysis nor average case analysis was looking at the position of a particular operation in the sequence. These two analyses considered the operations as executed in isolation and the sequence as a collection of isolated and independent operations. Amortized analysis changes the perspective by looking at what happened up until a particular point in the sequence of operations and then determines the complexity of a particular operation,

$$C(op_1, op_2, op_3, \ldots) = C(op_1) + C(op_2) \quad C(op_1) \quad \ldots$$

where C can be the worst, the average, the best case complexity, or very likely, a complexity other than the three depending on what happened before. To find amortized complexity in this way may be, however, too complicated. Therefore, another approach is used. The knowledge of the nature of particular processes and possible changes of a data structure is used to determine the function C which can be applied to each operation of the sequence. The function is chosen in such a manner that it considers quick operations as slower than they really are and time consuming operations as quicker than they actually are. It is as though the cheap (quick) operations are charged more time units to generate credit to be used for covering the cost of expensive operations that are charged below their real cost. It is like letting the government charge us more for social security than necessary so that at the end of the fiscal year the overpayment can be received back and used to cover the expenses of something else. The art of amortized analysis lies in finding an appropriate function C so that it overcharges cheap operations sufficiently to cover expenses of undercharged operations. The overall balance must be nonnegative. If a debt occurs, there must be a prospect of paying it.

Consider the operation of adding a new element to the vector implemented as a flexible array. The best case is when the size of the vector is less than its capacity because adding a new element amounts to putting it in the first available cell. The cost of adding a new element is thus $O(1)$. The worst case is when size equals capacity, in which case there is no room for new elements. In this case, new space must be allocated, the existing elements are copied to the new space, and only then can the new element be added to the vector. The cost of adding a new element is $O(size(vector))$. It is clear that the latter situation is less frequent than the former. But this depends on another parameter, capacity increment, which refers to how much the vector is increased when overflow occurs. In the extreme case, it can be incremented by just one cell, so in the sequence of m consecutive insertions, each insertion causes overflow and requires $O(size(vector))$ time to finish. Clearly, this situation should be delayed. One solution is to allocate, say, 1 million cells for the vector, which in most cases does not cause an overflow, but the amount of space is excessively large and only a small percentage of space allocated for the vector may be expected to be in actual use. Another solution to the problem is to double the space allocated for the vector if overflow occurs. In this case, the pessimistic $O(size(vector))$ performance of the insertion operation may be expected to occur only infrequently. By using this estimate, it may be claimed that in the best case, the cost of inserting m items is $O(m)$, but it is impossible to claim that, in the worst case, it is $O(m \cdot size(vector))$. Therefore, to see better what impact this performance has on the sequence of operations, the amortized analysis should be used.

In amortized analysis, the question is asked: What is the expected efficiency of a sequence of insertions? We know that the best case is $O(1)$ and the worst case is $O(size(vector))$, but also we know that the latter case occurs only occasionally and leads to doubling the size of the vector. In which case, what is the expected efficiency of one insertion in the series of insertions? Note that we are interested only in sequences of insertions, excluding deletions and modifications, to have the worst case scenario. The outcome of amortized analysis depends on the assumed amortized cost of one insertion. It is clear that if

$$amCost(push(x)) = 1$$

where 1 represents the cost of one insertion, then we are not gaining anything from this analysis because easy insertions are paying for themselves right away, and the insertions causing overflow and thus copying have no credit to use to make up for their high cost. Is

$$amCost(push(x)) = 2$$

a reasonable choice? Consider the table in Figure 2.6a. It shows the change in vector capacity and the cost of insertion when size grows from 0 to 18; that is, the table indicates the changes in the vector during the sequence of 18 insertions into an initially empty vector. For example, if there are four elements in the vector (size = 4), then before inserting the fifth element, the four elements are copied at the cost of four units and then the new fifth element is inserted in the newly allocated space for the vector. Hence, the cost of the fifth insertion is $4 + 1$. But to execute this insertion, two units allocated for the fifth insertion are available plus one unit left from the previous

fourth insertion. This means that this operation is two units short to pay for itself. Thus, in the units left column, −2 is entered to indicate the debt of two units. The table indicates that the debt decreases and becomes zero, one cheap insertion away from the next expensive insertion. This means that the operations are almost constantly executed in the red, and more important, if a sequence of operations finishes before the debt is paid off, then the balance indicated by amortized analysis is negative, which is inadmissible in the case of algorithm analysis. Therefore, the new best solution is now

$$\text{amortized}(\text{push}(x)) = 3$$

The table in Figure 2.6b indicates that we are never in debt and that the choice of three units for amortized cost is not excessive because right after an expensive insertion, the accumulated units are almost depleted.

In this example, the choice of a constant function for amortized cost is adequate, but usually, it is not. Define a potential function that assigns a number to a particu-

FIGURE 2.6

(a)

size	capacity	amortized cost	cost	units left
0	0			
1	1	2	0 + 1	1
2	2	2	1 + 1	2
3	4	2	2 + 1	0
4	4	2	1	1
5	8	2	4 + 1	2
6	8	2	1	−1
7	8	2	1	0
8	8	2	1	1
9	16	2	8 + 1	−6
10	16	2	1	−5
⋮	⋮	⋮	⋮	⋮
16	16	2	1	1
17	32	2	16 + 1	−14
18	32	2	1	−13
⋮	⋮	⋮	⋮	⋮

(b)

size	capacity	amortized cost	cost	units left
0	0			
1	1	3	0	3
2	2	3	1	5
3	4	3	2 + 1	5
4	4	3	1	7
5	8	3	4 + 1	7
6	8	3	1	9
7	8	3	1	11
8	8	3	1	13
9	16	3	8 + 1	13
10	16	3	1	15
⋮	⋮	⋮	⋮	⋮
16	16	3	1	17
17	32	3	16 + 1	19
18	32	3	1	21
⋮	⋮	⋮	⋮	⋮

lar state of a data structure ds that is a subject of a sequence of operations. The amortized cost is defined as a function

$$amCost(op_i) = cost(op_i) + potential(ds_i) - potential(ds_{i-1})$$

which is the real cost of executing the operation op_i plus the change in potential in the data structure ds as a result of execution of op_i. This definition holds for one single operation of a sequence of n operations. If amortized costs for all the operations are added, then the amortized cost for the sequence

$$\begin{aligned} amCost(op_1, \ldots, op_n) &= \sum_{i=1}^{n} (amCost(op_i) + potential(ds_i) - potential(ds_{i-1})) \\ &= \sum_{i=1}^{n} amCost(op_i) + potential(ds_n) - potential(ds_0) \end{aligned}$$

In most cases, the potential function is initially zero and is always nonnegative so that amortized time is an upper bound of real time. This form of amortized cost is used later in the book.

Amortized cost of including new elements in a vector can now be phrased in terms of the new form for the function amCost:

$$amCost(push_i) = \begin{cases} 0 & \text{if } size_i = capacity_i \text{ (vector is full)} \\ 3 & \text{otherwise} \end{cases}$$

To see that the function works as required, consider three cases. The first case is when a cheap pushing follows cheap pushing (vector is not extended right before the current push and is not extended as a consequence of the current push) and

$$amCost(push_i) = 1 + 2 - 2 = 3 - capacity_{i-1} - 2size_{i-1} - capacity_i = 3$$

because the capacity does not change, $size_i = size_{i-1} + 1$, and the actual cost equals 1. For expensive pushing following a cheap push,

$$amCost(push_i) = size_{i-1} + 2 - 2size_{i-1} + capacity_{i-1} = 3$$

because $size_{i-1} + 1 = capacity_{i-1}$ and the actual cost equals $size_i + 1 = size_{i-1} + 2$, which is the cost of copying the vector elements plus adding the new element. For cheap pushing following expensive pushing,

$$amCost(push_i) = 1 + 2size_i - capacity_i = 3$$

because $2size_i - 1 = capacity_i$ and actual cost equals 1. Note that the fourth case, expensive pushing following expensive pushing, occurs only once, when capacity changes from zero to one and from one to zero. In both cases, amortized cost equals 3.

2.10 Exercises

1. Explain the meaning of the following expressions:
   a. $f(n)$ is $O(1)$.
   b. $f(n)$ is $\Omega(1)$.
   c. $f(n)$ is $\Theta(1)$.
2. Assuming that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, prove the following statements:
   a. $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$.
   b. If a number k can be determined such that for all $n > k$, $g_1(n) \leq g_2(n)$, then $O(g_1(n)) + O(g_2(n))$ is $O(g_2(n))$.
   c. $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$ (rule of products).
   d. $O(g(n))$ is $O(g(n))$.
   e. c is $O(1)$.
3. Prove the following statements:
   a. $\sum_{i=1}^{n} i^2$ is $O(n^3)$ and more generally, $\sum_{i=1}^{n} i^k$ is $O(n^{k+1})$.
   b. $\max(n, n^2)$ is $O(n^2)$ but $\max(n, n^2)$ is not $\Theta(n^2)$.
   c. $n^k - n \lg n$ is $\Theta(n^k)$.
   d. 2^n is $O(n^2)$ and it is not $O(2^n)$.
   e. 2^{n+a} is $O(2^n)$.
   f. 2^{an} is not $O(2^n)$.
   g. $2^{\lg n}$ is $O(n^2)$.
4. Make the same assumptions as in Exercise 2 and, by finding counterexamples, refute the following statements:
   a. $f_1(n) - f_2(n)$ is $O(g_1(n) - g_2(n))$.
   b. $f_1(n)/f_2(n)$ is $O(g_1(n)/g_2(n))$.
5. Find functions f_1 and f_2 such that both $f_1(n)$ and $f_2(n)$ are $O(g(n))$, but $f_1(n)$ is not $O(f_2)$.
6. Is it true that
   a. If $f(n)$ is $\Theta(g(n))$, then $2^{f(n)}$ is $\Theta(2^{g(n)})$?
   b. $f(n) + g(n)$ is $O(\min(f(n), g(n)))$?
   c. $n!$ is $O(2^n)$?
7. The algorithm presented in this chapter for finding the longest ascending subarray in $A[1..n]$ keeps track of the longest ascending subarray seen so far while it scans the array from left to right. Suppose, instead, that you decide to solve this problem by finding the longest ascending subarray starting at $A[1]$, then the longest ascending subarray starting at $A[2]$, and so on. Stop if you ever reach a point where the longest ascending subarray found so far is longer than the number of elements remaining to be examined. Why does this algorithm also correctly solve the problem? Show that in the worst case, this algorithm is $O(n^2)$. Show that in the best case, this algorithm is $\Omega(n)$. Under what circumstances does the best case occur? What is the worst case?

Eliminate the search right away, convert up the worst case into the best. The change needed is in the outer loop, which now has one more test:

```
for(i = 0; i < n-1 && length < n-i; i++)
```

What is the worst-case number in the efficiency of the worst case still $O(n^2)$?

8. Find the complexity of the function used to find the kth integer in an unordered array of integers:

```
int smallest(int a[], int n, int k) {
    int i, j, min, tmp;
    for (i = 0; i < k; i++) {
        min = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[min])
                min = j;
        tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
    return a[k-1];
}
```

9. Determine the complexity of the following implementations of the algorithms for adding, multiplying, and transposing square matrices:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        m3[i][j] = m1[i][j] + m2[i][j];

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = m3[i][j] = 0; k < n; k++)
            m3[i][j] += m1[i][k] * m2[k][j];

for (i = 0; i < n - 1; i++)
    for (j = i+1; j < n; j++) {
        tmp = m1[i][j];
        m1[i][j] = m1[j][i];
        m1[j][i] = tmp;
    }
```

10. Find the computational complexity for the following functions:

a.
```
for (cnt1 = 0, i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        cnt1++;
```

b.
```
for (count2 = 0, i = 1; i <= n; i*=2)
    for (j = 1; j <= n; j++)
        count2++;
```

c.
```
for (count3 = 0, i = 1; i <= n; i *= 2)
    for (j = 1; j <= n; j++)
        count3++;
```

d.
```
for (count4 = 0, i = 1; i <= n; i *= 2)
    for (j = i; j <= 2*i; j++)
        count4++;
```

11. Find the average case complexity of sequential search if the probability of accessing the first cell is p, the probability of the second cell is q, and the probability of locating a number in any of the remaining cells is the same and equal to $\frac{1-p-q}{n-2}$.

12. Consider a process of incrementing a binary number by 1. For example, incrementing 0011 results in 0100 because 3 bits are changed to 0s before a 1 is changed to a 0. In the best case, only one bit is changed, as when incrementing 0110 to 0111. In the worst case, all the bits are changed, as when incrementing 1111 to 0000.

Number	Flipped Bits
000	
001	1
010	2
011	1
100	3
101	1
110	2
111	1

Using worst case assessment, we may announce that the cost of executing $n = 2^k - 1$ increments is $O(nk)$. Use amortized analysis to show that the cost of executing n increments is $O(n)$.

Bibliography

Computational Complexity

Papadimitriou, Christos H. *Computational Complexity*. Addison-Wesley, 1994.

Sipser, Michael. "Introduction to the Theory of Computation." *ACM SIGACT News* 27, no. 1 (March 1996): 27–29.

Knuth, Donald E. "Computer Science and Its Relation to Mathematics." *American Mathematical Monthly* 81, no. 4 (April 1974): 323–343.

Big-O, Ω, and Θ Notations

Brassard, G., "Crossing Complexity Barriers," *ACM SIGACT News* 15, no. 2 (1983): 34–42.

Knuth, Donald, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Reading, MA: Addison-Wesley, 1968.

Knuth, Donald, "Big Omicron and Big Omega and Big Theta," *SIGACT News* 8, no. 2 (April–June 1976): 18–24.

Vitányi, P.M.B. and Meertens, L., "Big Omega versus the Wild Functions," *SIGACT News* 16, no. 4 (1985): 56–59.

OO Notation

Kozen, D., *The Design and Analysis of Algorithms*, New York: Springer-Verlag, 1992.

Amortized Analysis

Tarjan, Robert E., "Amortized Computational Complexity," *SIAM Journal on Algebraic and Discrete Methods*, 1985, pp. 306–318.

Linked Lists

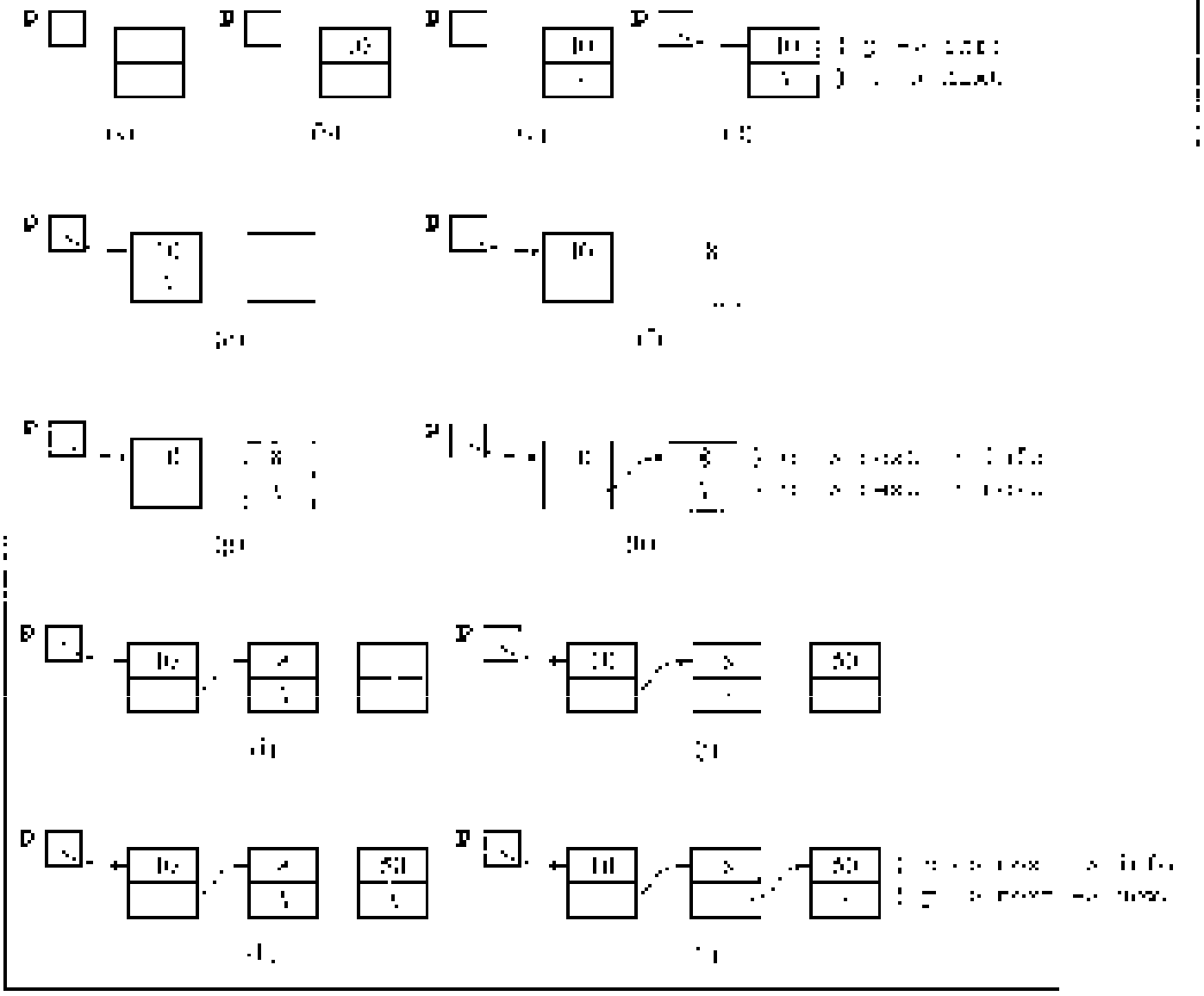
An array is a very useful data structure provided in programming languages. However, it has at least two limitations: (1) its size has to be known at compilation time, and (2) the data in the array are separated in computer memory by the same distance, which means that inserting an item inside the array requires shifting other data in this array. This limitation can be overcome by using 'linked structures'. A linked structure is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in memory and passing from one node of the linked structure to another is accomplished by storing the addresses of other nodes in the linked structure. Although linked structures can be implemented in a variety of ways, the most flexible implementation is by using pointers.

3.1 Singly Linked Lists

In its most common form, a data member that is a pointer to another node. Data nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a linked list, which is a data structure composed of nodes each containing some information and a pointer to another node in the list. We can move in a list only to its successor in this sequence, and the list is called a singly linked list. An example of such a list is shown in Figure 3.1. Note that only one variable pointing to access any node in the list. The last node on the list can be recognized by the null pointer.

Each node in the list in Figure 3.1 is an instance of the following class definition:

```
class IntNode {
public:
    IntNode();
```



```
        ...
    }
    ...
    ...
    ...
};
```

...

next IntLink reference creates a linked list. It is an auxiliary data member used to help maintain the linked list. It is indispensable for the implementation of the linked list but is unimportant and irrelevant from the user's perspective. Note that IntNode is defined in terms of itself because one data member, next, is a pointer to a node of the same type that is just being declared. This circularity, however, is permitted in C++.

The definition of a node also includes two constructors. The first constructor initializes the next pointer to null and leaves the value of data unspecified. The second constructor takes two arguments, one to initialize the data member and another to initialize the next member. The second constructor also covers the case when only one numerical argument is supplied by the user. In this case data is initialized to the argument and next to null.

Now let us insert the integer 5 in Figure 3.6. One way to create the three node linked list is to first generate the node to store the number 10, then the node to store 5, and finally the node containing 30. Each node has to be handled properly and incorporated into the list. Inserting 5 at the head of the list is illustrated in Figure 3.7. We start by executing the declaration and assignment

```
IntNode *p = new IntNode(10);
```

which creates the first node on the heap and makes the variable p point to this node. This is done in two steps. In the first step a new IntNode is created (Figure 3.7a). In the second step the data member of this node is set to 10 (Figure 3.7b). And in the third step the node's next member is set to null (Figure 3.7c). The null pointer is indicated with a slash in the pointer data member. Note that the slash in the next member is not a slash character. That is, the second and third steps of the initialization of data members of the new IntNode are performed by invoking the constructor IntNode(10), which turns into the constructor call IntNode(10, 0). The zero is used to initialize the next member of the newly created node to 0 (or NULL). This number is the address of the node, and it is shown as an arrow from the next data member to the node.

The second node is created with the assignment

```
p->next = new IntNode(30);
```

where p->next is the next member of the node pointed by p (see Figure 3.8a). After the list has been extended:

1. a new node (Figure 3.8a),
2. assigning to the data member of this node 30 (Figure 3.8b), and
3. setting its next member (Figure 3.8c), and finally
4. connecting the new node to the list by making the next member of the first node a pointer to the new node (Figure 3.8d).

Note that the data members of these nodes can be accessed using the arrow notation, which is easier than using a combination of * and . next.

The linked list is now extended by adding a third node with the assignment

```
p->next->next = new IntNode(50);
```

when getnext() is applied to the next to last node of the list, it returns the last node. The getnext() function can be used to traverse the entire list sequentially from beginning to end.

In processing the last node of a list, it is important to remember that the link field of the last node is NULL, indicating that there are no more nodes following it. When writing code to traverse a linked list, we must always stop when we reach the last node. Failure to do so usually results in the program losing access to all of the nodes in the list that precede the last node, and most likely will cause the program to terminate abnormally. For example, the getnext() function would return NULL if we tried to retrieve the node following the last node in the list. Attempting to apply getnext() to this NULL value to retrieve the node after it would produce an error, because NULL does not point to a node and thus has no link field. Trying to access a link field that doesn't exist usually causes a program to crash.

Another important observation about linked lists is that accessing the nodes is a sequential process. To reach a node in the middle of the list, we must start at the beginning and follow links through all of the nodes that precede the one we want. There is no way to directly access a node in the middle of the list. One way to always keep two pointers to the list is to have the last node point back to the first, as shown in Fig. 3.2.

Figure 3.2 A circularly linked list of integers.

```
//*************************  intCircList.h  ***************************
//            a circularly linked list to store integers

#ifndef INTCIRCLIST_H
#define INTCIRCLIST_H

class IntNode {
public:
    int info;
    IntNode *next;
    IntNode(int el, IntNode *ptr = 0) {
        info = el; next = ptr;
    }
};

class IntCircList {
public:
    IntCircList() {
        head = tail = 0;
    }
    ~IntCircList();
    int isEmpty() {
        return head == 0;
    }
    void addToHead(int);
```

Listing 3.2 (continued)

```
    void addNode(int);
    int  deleteFromHead(); // delete the head and return its node;
    int  deleteTail(); // delete the tail and return its node;
    void deleteNode(int);
    bool isInList(int) const;
private:
    IntNode *head, *tail;
};

#endif

//************************** IntSLList.cpp ***************************

#include <iostream.h>
#include "IntSLList.h"

IntSLList::~IntSLList() {
    for (IntNode *p; !isEmpty(); ) {
        p = head->next;
        delete head;
        head = p;
    }
}
void IntSLList::addToHead(int el) {
    head = new IntNode(el,head);
    if (tail == 0)
       tail = head;
}
void IntSLList::addToTail(int el) {
    if (tail != 0) {    // if list not empty:
        tail->next = new IntNode(el);
        tail = tail->next;
    }
    else head = tail = new IntNode(el);
}
int IntSLList::deleteFromHead() {
    int el = head->info;
    IntNode *p = head;
    if (head == tail)   // if only one node in the list;
        head = tail = 0;
```

Program 3.2 (continued)

```
    else head = head->next;
    delete tmp;
    return 0;
}
int CircularList::deleteNode(int key) {
    node *pre = tail->info;
    if (pre == key) { // if only one node in the list;
        delete pre;
        head = tail = 0;
    }
    else {              // if more than one node in the list,
        int found = 0;  // find the predecessor of tail;
        for (tmp = head; tmp->next != tail; tmp = tmp->next);
        delete tail;
        tail = tmp;     // the predecessor of tail becomes tail;
        tail->next = 0;
    }
    return 0;
}
void CircularList::deleteNode(int key) {
    if (head == 0)                          // if non-empty list;
        if (head == tail && key == head->info) { // if only one
            delete head;                        // node in the list;
            head = tail = 0;
        }
        else if (key == head->info) { // more than one node in the list
            int *tmp = head->next;
            head = head->next;
            delete tmp;             // and old head is deleted;
        }
        else {                      // more than one node in the list
            int *prev, *tmp;
            for (prev = tail, tmp = head->next; // find a node and
                 tmp != 0 && tmp->info != key; // its predecessor;
                 prev = prev->next, tmp = tmp->next);
            if (tmp != 0) {
                prev->next = tmp->next;
                if (tmp == tail)
                    tail = prev;
                delete tmp;
            }
```

Figure 3.2 (continued)

```
        }
;
bool IntSLList::isInList(int k) const {
    IntSLNode *tmp;
    for (tmp = head; tmp != 0 && tmp->info != k; tmp = tmp->next);
    return tmp != 0;
}
```

A singly linked list implementation in Figure 3.2 uses two classes: one class, IntSLNode, for nodes of the list, and another, IntSLList, for access to the list. The class IntSLList defines two data members, head and tail, which are pointers to the first and the last nodes of a list. The explains why all members of IntSLNode are declared public. Because particular nodes of the list are accessible through pointers, nodes are made inaccessible to outside objects by declaring head and tail private so that the information hiding principle is not easily compromised. If some of the members of IntSLNode were declared nonpublic, then classes derived from IntSLList could not access them.

An example of a list is shown in Figure 3.3. The list is declared with the statement

```
IntSLList list;
```

The list object in Figure 3.3a is not part of the list; it allows for having access to the list. To simplify the subsequent figures, node labels, pointers to the list are shown, but access is outside, and the head and tail members are marked as in Figure 3.3a.

Besides the head and tail members, the class IntSLList also defines a number of functions that allow us to manipulate the list. We now look more closely at some basic operations on linked lists presented in Figure 3.2.

3.1.1 Insertion

Adding a node at the beginning of a linked list is performed in four steps:

1. An empty node is created. It is empty in the sense that the program performing insertion does not assign any values to the data members of the node (Figure 3.3b).
2. The new empty node is joined to the list by making its next pointer point to the first node on the list, that is, the current value of head (Figure 3.3c).
3. Because the node is to be included at the front of the list, the head member becomes a pointer to the first node on the list, that is, the current value of head (Figure 3.3d).
4. There remains to be considered the case of the list being empty before insertion. In this case, both head and tail should point to the new node. This is accomplished by setting tail equal to head, which is now also pointing to the new node (Figure 3.3e).

FIGURE 3.3 A singly linked list of integers.

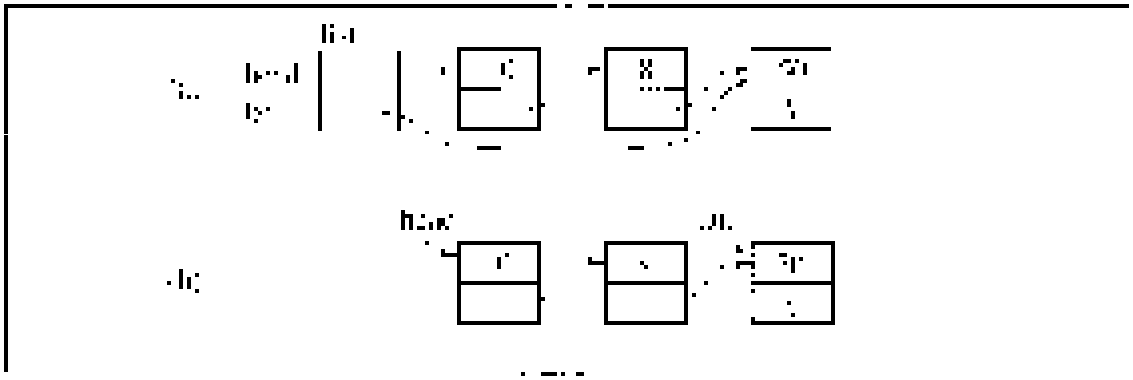
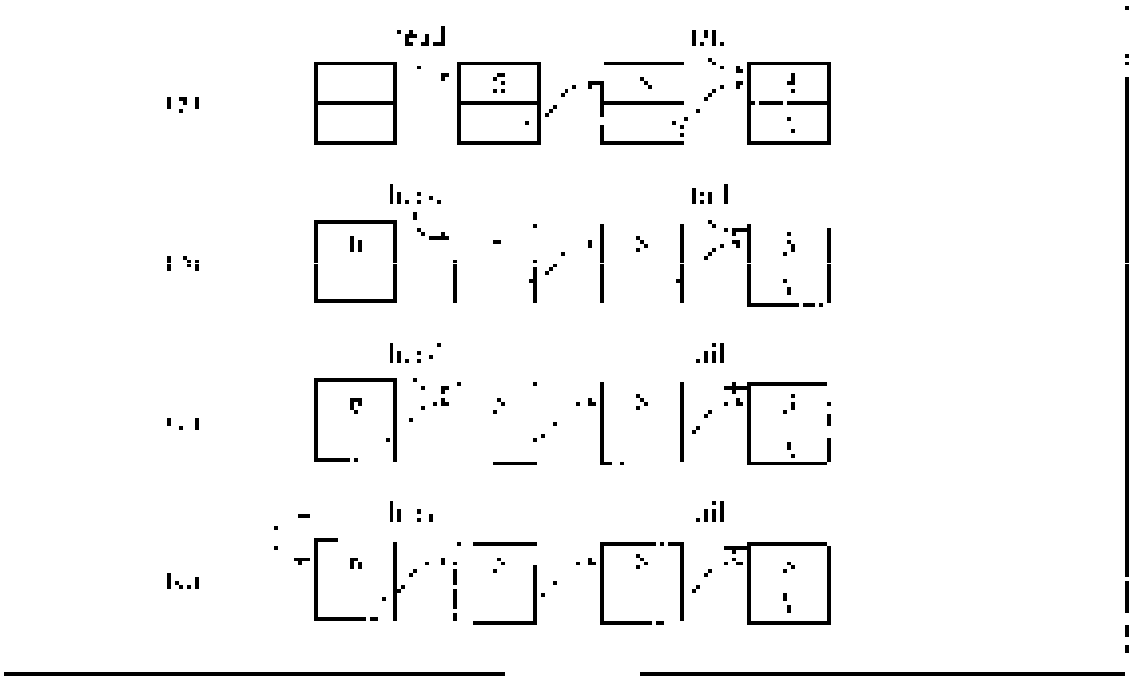


FIGURE 3.4 Inserting a new node at the beginning of a singly linked list.

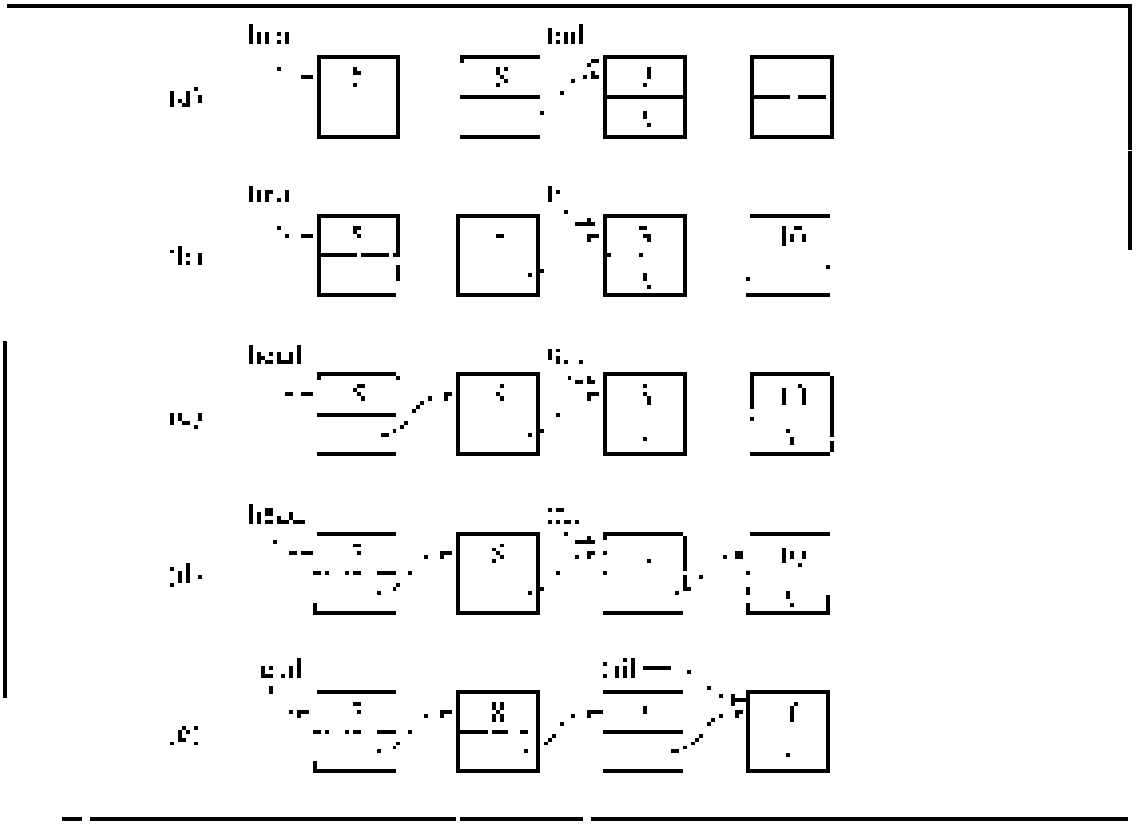


The four steps are executed by the next two instructions (see Figure 3.4). The function executes the first three steps of the process by calling the constructor `IntNode(61, head)`. The last step is executed directly by the function by assigning the address of the newly created node to `head`.

The member function `addToHead` is a complete function for the special case of inserting a new node in an empty linked list. It is correctly implemented for both `head` and `tail` are `nil`. Therefore, `temp` becomes `tail` as the only node of the new list. When inserting a node in an empty list only the `tail` needs to be updated.

The process of inserting a new node to the end of a list is similar.

Figure 3.5 Inserting a new node at the end of a singly linked list.

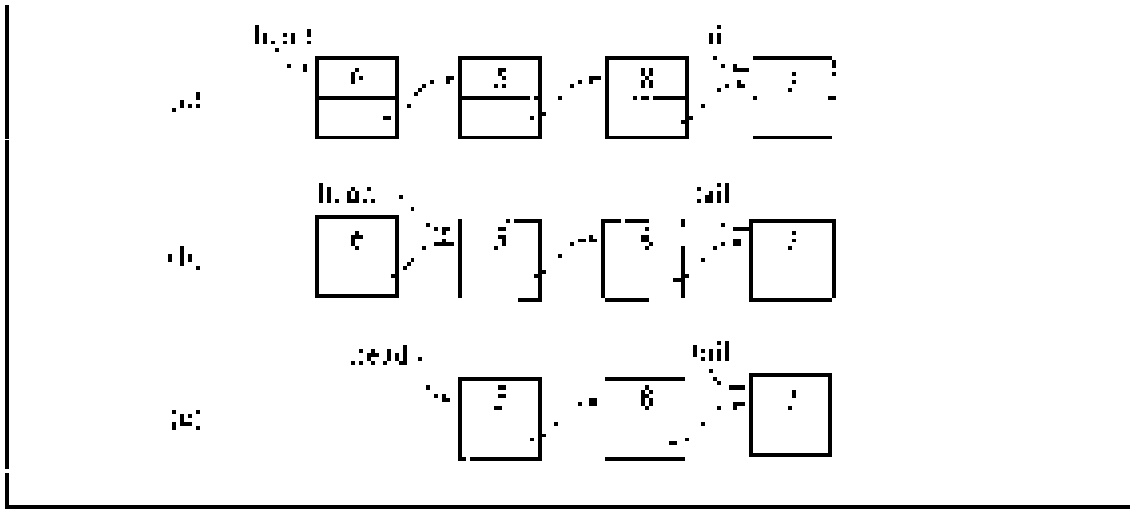


1. An available node is created (Figure 3.5(a)).
2. The node's data member is initialized to an integer (Figure 3.5(b)).
3. Because the node is being inserted at the end of the list, the next member is set to NULL (Figure 3.5(c)).
4. The node is now included in the list by making the next member of the last node of the list point to the newly created node (Figure 3.5(d)).
5. The new node follows all the nodes of the list, but it should be pointed to by the value of last, which now becomes the address of the new node (Figure 3.5(e)).

All these actions are executed in the else clause of `addatend()` (Figure 3.4). The else clause of the function is executed only if the linked list is not empty. If this were not included, the program may crash because in the else clause we make an assignment to the next member of the node referenced by last. In the case of an empty linked list, it is an attempt to access the data member of a nonexistent node.

The process of inserting a new node at the beginning of the list is very similar to the process of inserting a node at the end of the list. It is even simpler because it requires the manipulation of only two pointer members: head and last. For this reason, both addatbeg() and addatend() have the same time complexity. That is, regardless of the number of nodes in the list, the number of operations performed

FIGURE 3.6 Deleting a node at the beginning of a singly linked list



by the two member functions `insert` and `erase`, respectively. Note that because the `find` member function returns a `Position` object, it can be used to indicate where in the list a new node should be inserted. For example, a new node can be added at the end of the list. If the `tail` pointer were not used, then adding a node at the end of the list would be more time-consuming because we would first have to search the list node by node for the node with a `next` pointer equal to the `tail` of the list. The process of searching the list would take linear time, as opposed to the constant time required if we use `tail`.

3.1.2 Deletion

One deletion operation is `deleteList`, which deletes all the nodes in a linked list. This operation is implemented by the member function `makeEmpty`. To do so, we must first decide what to do with the header node. We can either keep it or delete it. The code that we present here implements the latter option. Our `makeEmpty` routine marches down the list, one node at a time, freeing the memory used by the node (Figure 3.6).

As the routine stands, the last node to be deleted is not the header, but the node before the last node in the list. We leave it as an exercise to the reader to remedy this situation.

The second routine of interest is `find`, which locates an object in the list for the first time. It returns a `Position` corresponding to the node in which the item was found. `find` uses a standard idiom of stepping through the list with a `while` loop until we have either exhausted the list or found the item we are looking for. In the latter case, we can then use the `Position` returned to perform other kinds of operations. The code does not depend on the `Position` class, but rather on the implementation of `Position`. Because `find` is a member function, it can access the `next` field of the node that is passed as a `Position`. Thus `find` returns a `Position` that is actually a pointer to the node that contains the item. The client code, which uses `List` and `Position` but does not directly manipulate `Position`'s `next` field, is thus unaware of this detail.

There are several ways to implement the `findPrevious` routine required by `remove`. Our routine uses the header and a `while` loop to advance `p` and `p->next` until we detect that we have gone past the end of the list.

```
int IntSLList::deleteFromHead() {
    assert(!isEmpty()); // terminate the program if empty;
    int el = head->info;
    . . . . . .
    return el;
}
```

The assert statement, used to test a condition in debugging mode, if the condition is false, the program is aborted. It's a simple solution to exceptional situations, even if the execution can be continued as in deleteFromTail() and deleteFromHead().

Another solution is to throw an exception and let the user decide what to do:

```
int IntSLList::deleteFromHead() {
    if (isEmpty())
        throw "Empty";
    int el = head->info;
    . . . . . .
    return el;
}
```

The throw clause with the string argument is expected to cause a match in a try-catch clause in the caller (or caller's caller etc.) also with the string argument which catches the exception as in

```
void f() {
    . . . . . .
    try {
        n = list.deleteFromHead();
        // do something with n;
    } catch(char *s) {
        cout << "Error: " << s << endl;
    }
    . . . . . .
}
```

This solution gives the caller some control over the abnormal situation without making it lethal to the program as with the use of the assert statement. The user is responsible for providing an exception handler in the form of the try-catch clause, with the solution appropriate to the particular case. If the clause is not provided, the program crashes when the exception is thrown. The function f() may only print a message that a list is empty when an attempt is made to delete a number from an empty list, another function g() may assign a certain value to n in such a case, and yet another function h() may find such a situation detrimental to the program and abort the program altogether.

The idea that the user is responsible for providing an action in the case of an exception is also presumed in the implementation given in Figure 3.2. The member function assumes that the list is not empty. To prevent the program from crashing, the

member function `isEmpty()` is added to the linked list class, and the user should use it as in

```
if (!list.isEmpty())
        = list.deleteFromHead();
else do not remove
```

Note that including a similar test in `deleteFromHead()` does not solve the problem (can you see this?):

```
int IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if non-empty list;
        int el = head->info;
        ...
        return el;
    }
    else return 0;
}
```

It is not a serious problem, but the user cannot be sure whether 0 is the program's output or the signal that the list is empty. Moreover, the caller does not know whether the returned 0 is the sign of a successful deletion from the list. To avoid any confusion, the caller must test if a statement to test whether the list is empty before calling `deleteFromHead()`. In this way, one `if` statement would be redundant.

To maintain uniformity in the interpretation of the return value, the function can be modified so that, instead of returning an integer, the function returns a pointer to an integer:

```
int* IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if non-empty list;
        int *el = new int(head->info);
        ...
        return el;
    }
    else return 0;
}
```

where 0 in the `else` clause is the null pointer, not the number 0. In this case, the function call

```
p = list.deleteFromHead();
```

results in a program crash if `deleteFromHead()` returns the null pointer.

Therefore, a test must be performed by the user before calling `deleteFromHead()` to check whether `list` is empty and a pointer variable has to be used:

```
int *p = list.deleteFromHead();
```

and then a test is performed after the call to check whether `p` is null or not. In either case, this means that the `if` statement in `deleteFromHead()` is redundant.

The second special case is when the list has only one node to be removed. In this case the list becomes empty, which requires setting last and head to null.

The second deletion operation consists in deleting a node from the end of the list and it is implemented as the member function deleteFromEnd(). The problem is that after removing a node, last should refer to the new end of the list, that is, last has to be moved backward one node. But moving backward is impossible because there is no direct link from the last node to its predecessor. Instead, its predecessor has to be found by searching for it from the beginning of the list and stopping right before last. This is accomplished with a temporary variable that is used to search the list within the for loop. The variable tmp is initialized to the head of the list, and then in each iteration of the loop it is advanced to the next node. If the list is as in Figure 3.2, then the first test finds the head node holding number 1, and executing the assignment tmp = tmp->next; tmp refers to the second node (Figure 3.2b). After the second iteration and executing the same assignment, tmp refers to the third node (Figure 3.2c). Because this node is also the next to last node, the loop is exited, after which the last node is deleted (Figure 3.2d). Because tmp is now pointing to a nonexisting node, it is immediately set to point to the next to last node currently pointed to by the first node. The effect is that the last node of the list, the node number 3 in this case, is removed (Figure 3.2e).

Note that in the for loop a temporary variable is used to search the list. If the loop were simplified to

```
for ( ; head->next != last; head = head->next);
```

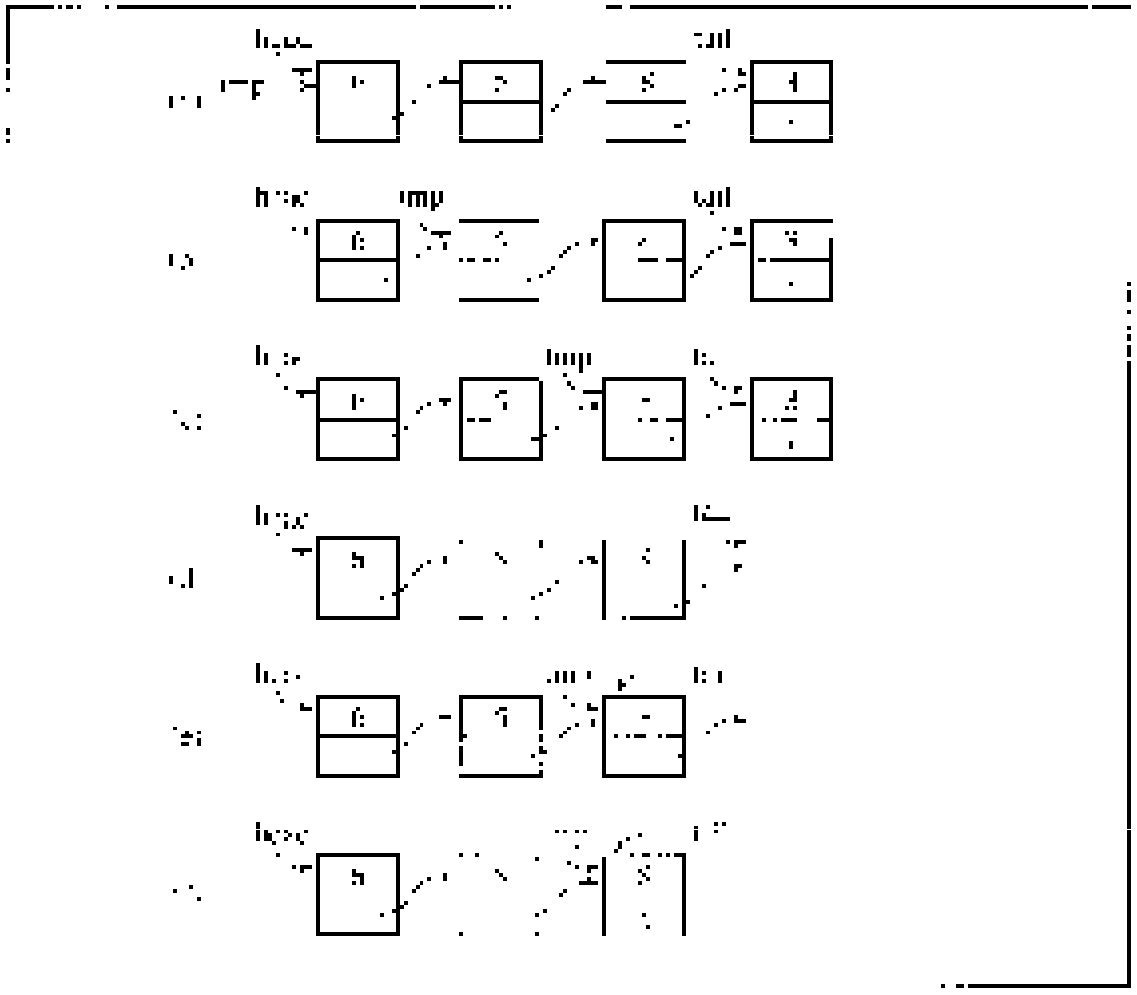
then the list would need only one variable, and access to the beginning of the list would be erased each time the loop is executed. In the next to last node, which is about to become the last node, it is absolutely critical that its address is not in a temporary variable as soon as head is moved. But the address to the beginning of the list is kept intact.

In removing the first node the two special cases are the same as in deleteFromFront(). If the list is empty, then nothing can be removed, but what should be done in this case is decided in the client program just as in the case of deleteFromFront(). The second case is when a single node list becomes empty after removing its only node, which also requires setting head and last to null.

The most time-consuming part in deleteFromEnd() is finding the next to last node performed by the for loop. The search in the loop performs $n - 1$ iterations in a list of n nodes, which is the main reason this member function takes $O(n)$ time to delete the last node.

The remaining deletion operation removes a node from the head or from the tail. It returns data from the node and returns an integer that happens to be in the node being removed. A different approach is when we want to delete a node that holds a particular integer, regardless of the position of this node in the list. It may be right at the beginning, at the end, or anywhere inside the list. Briefly, a node has to be located first and then detached from the list by linking the predecessor of this node directly to its successor. Because we do not know where the node may be, the process of finding and deleting a node with a certain integer is much more complex than the deletion operations discussed so far. The member function deleteNode() in Figure 3.3 is an implementation of this process.

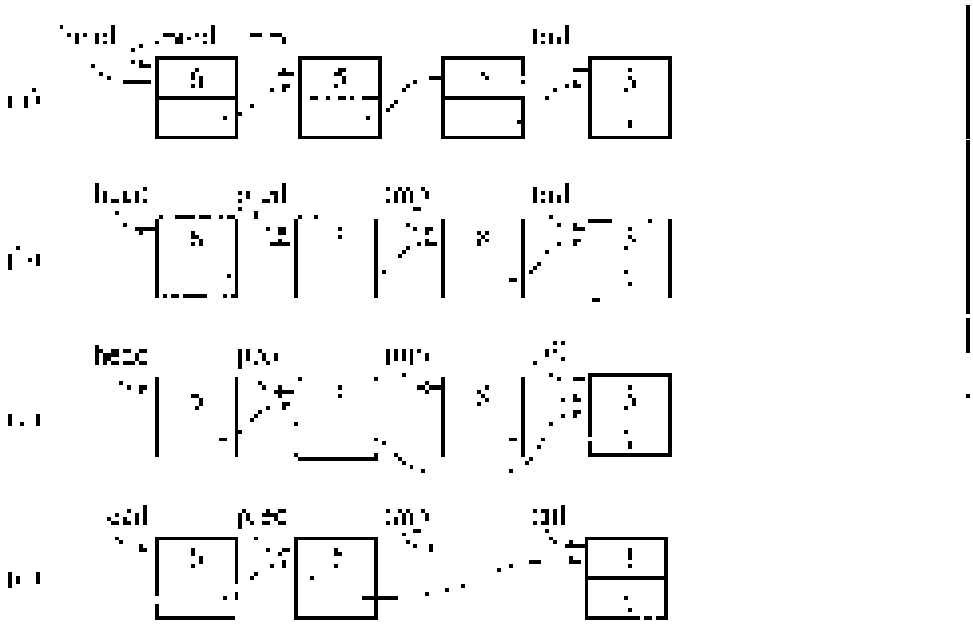
FIGURE 3.7 Deleting a node from the end of a singly linked list.



A node is removed from a doubly linked list by linking its predecessor to its successor. Because the list is only forward linked, the predecessor of a node cannot be reached from that node. One way to accomplish the task is to find the node to be removed first, by starting at the head and then moving to the node's predecessor. Another way is presented in DeleteNode(), as shown in Figure 3.8. Assume that we want to delete a node that holds number 8. If a function receives a variable pointed at that node, which is called ptr in the function, the node pointer to the first and second nodes of the list can be easily found (Figure 3.8a). Because the node holds number 8, the next node to be removed is pointed to by ptr->next and is stored in the temporary variable tmp (Figure 3.8b). Now the node can be deleted by linking ptr to tmp->next (Figure 3.8c). Finally, tmp is assigned to NULL to destroy the node and free the memory it occupies (Figure 3.8d).

In general, a singly linked list is more dynamic than an array. It can be used in applications

FIGURE 3.8 Detecting a node in a singly linked list.



1. A node might point to a node that is empty, in which case the list ends at this node.
2. Detecting read only node does not cause linked list. Both head and tail point to null.
3. If node might point back to the list with a cycle, then we have a list with repeated nodes.
4. Removing the last node of the list with a tail two moves, leading to the loss of tail.
5. A pattern of nodes that is not a cycle but is not a linked list. Do not do this.

The algorithm for detecting a cycle is deceptively simple. The head node is the first node, which takes $O(1)$ time to access. The worst case is when the last node in the list is pointed to, which requires traversing the entire list. Therefore, the algorithm runs in $O(n)$ time. We also use $O(1)$ space. Assuming that any node in the list is equally likely to be the loop point, the loop portion is on average $n/2$ nodes. The fast node moves at a rate of two nodes per iteration, so it gains on the slow node at a rate of one node per iteration. Therefore, the number of iterations required is on average

$$\frac{n/2}{2-1} = \frac{n/2}{1} = \frac{n}{2}$$

That is, on the average, the algorithm requires $O(n)$ steps to finish, just like in the worst case.

3.1.3 Search

The insertion and deletion operations modify linked lists. The searching operation searches an existing list to see whether or not a number is in it. We implement this operation in the function `isInList()`. The function uses the temporary variable `tmp` to run through the list, starting from the head node. The number stored in each node is compared to the number being sought, and if the two numbers are equal, `true` is returned; otherwise, `tmp` is updated to take on the next value in the list. The loop terminates when `tmp` becomes `NULL`, which is used to indicate that the number being sought is not in the list. The reason that the search was encapsulated in a function is because it was used in more than one location. That is why `isInList()` returns the result of comparing `tmp` to `NULL`. If `tmp` is not `NULL`, then the search was successful and `false` is returned. If `tmp` is `NULL`, then the search was unsuccessful and `false` is returned.

We leave writing a function to find and delete the element at a certain node (`findAndDelete()`) as an exercise. This function takes $O(1)$ time in the best case and $O(n)$ in the worst and average cases.

In the foregoing discussion, the operations on singly linked lists were stressed. However, a linked list is built for the sake of storing and processing information, not for its own intrinsic interest. The choice of approach used in this section is in order to keep the discussion simple. If we wanted a linked list for doubles or characters or arrays, then we would have to declare a new struct and a number of new functions. This is not difficult, but it is tedious. However, it is more advantageous to declare a struct only once, without needing to know in advance what type of data will be stored in it. This can be done very conveniently in C++ with templates. To illustrate the use of templates for list processing, the next section uses templates to define lists of integers and the list operations we will discuss to use those stored integers.

3.2 Doubly Linked Lists

The function `findAndDeleteFromTail()` introduces a problem with singly linked lists. The nodes in such lists contain only pointers to the successor. Therefore, there is no information to access the predecessor. For this reason, `deleteFromTail()` must work up the list with a loop which allows it to find the predecessor of `tail`. Although this predecessor is, so to speak, within sight, it is out of reach. We have to walk there. The list must be traversed in front of `tail` to delete it. For long lists and for frequent executions of `deleteFromTail()`, this may be an impediment to efficient operation. To avoid this problem, the linked list is redefined so that each node in the list has two pointers: one to the successor and one to the predecessor. A list of this type is called a **doubly linked list**, and is illustrated in Figure 3.9. Figure 3.10 contains a template specification for a generic `DoublyLinkedList` class.

Algorithms and functions for processing doubly linked lists are slightly more complicated than those for singly linked lists, but not by much. The reason for the increased complexity is that there is one more pointer

FIGURE 3.9 A doubly linked list.

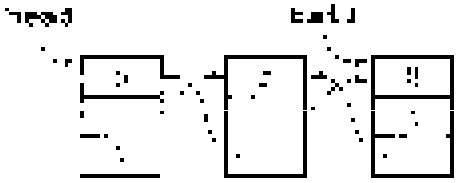


FIGURE 3.10 An implementation of a doubly linked list.

```
#include DOUBLY_LINKED_LIST
#define DOUBLY_LINKED_LIST

#include <iostream.h>

template<class T>
class node {
public:
    node() :
        next = prev = 0;
    }
    node(const T& d, node *n = 0, node *p = 0) :
        info = d; next = n; prev = p;
    }
    T info;
    node *next, *prev;
};

template<class T>
class doubly_linked_list {
public:
    doubly_linked_list() :
        head = tail = 0;
    }
    void add_to_head(const T&);
    T delete_from_head();
    ...
protected:
    node<T> *head, *tail;
};
```

Program 3.10 (continued)

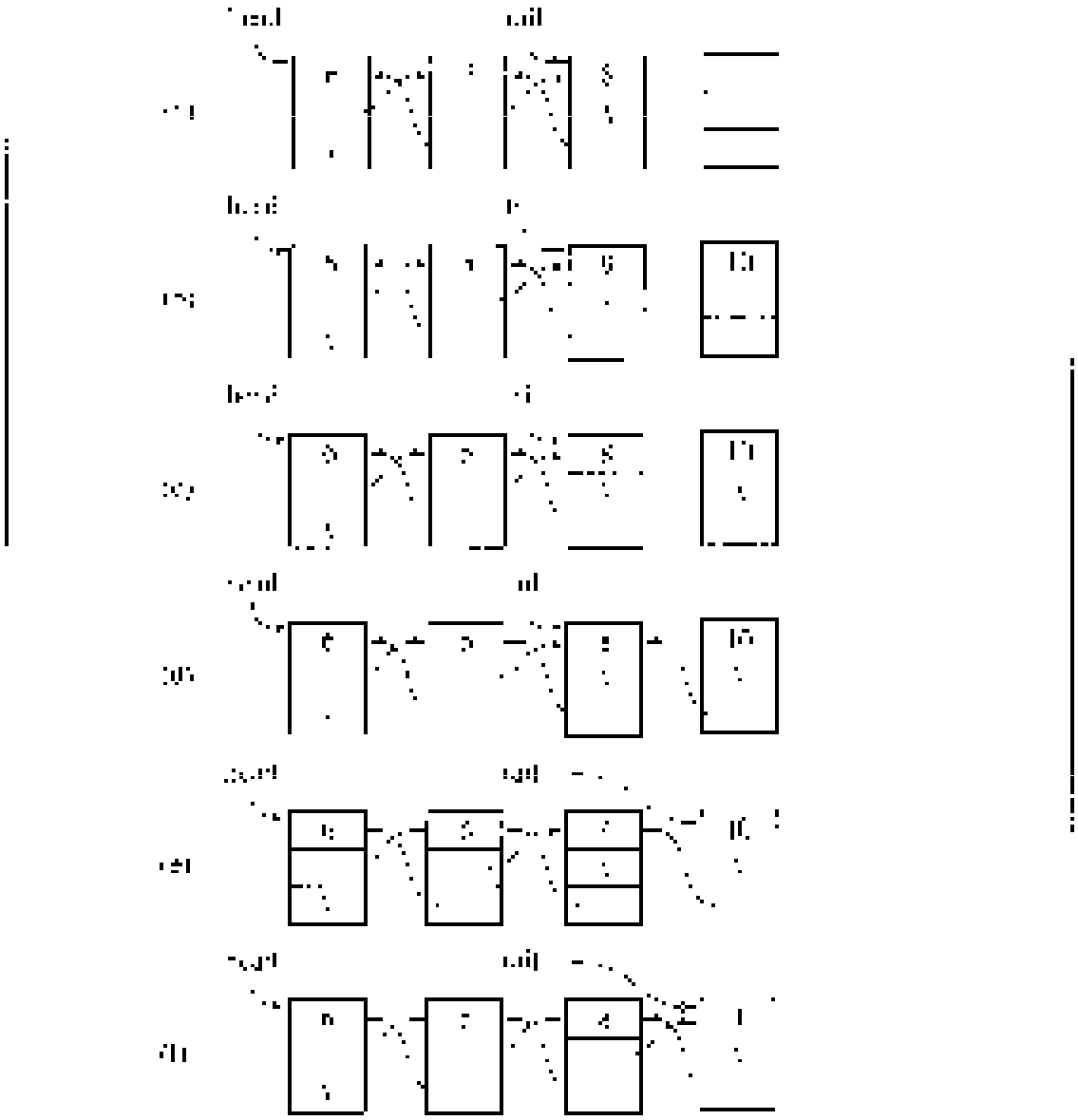
```
template<class T>
void DoubleLinkedList<T>::sendToTail(const T& x) {
    if (tail != 0) {
        tail = new Node<T>(x,0,tail);
        tail->prev->next = tail;
    }
    else head = tail = new Node<T>(x);
}
template<class T>
T DoublyLinkedList<T>::removeTail() {
    Node<T>* tail = this->tail;
    if (head == tail) { // if only one node in the list;
        delete head;
        head = tail = 0;
    }
    else {              // if more than one node in the list;
        tail = tail->prev;
        delete tail->next;
        tail->next = 0;
    }
    return x;
}
// . . . . . .
#endif
```

member to be maintained. Only two functions are discussed: a function to insert a node at the end of the doubly linked list and a function to remove a node from the end (Figure 3.10).

To add a node to a list the node has to be created, its data members properly initialized, and then the node needs to be incorporated into the list. Inserting a node at the end of a doubly linked list is performed by `sendToTail()` as illustrated in Figure 3.11. The process is performed in six steps.

1. A new node is created (Figure 3.11a) and then its two data members are initialized.
2. The `prev` member is set to point to the node pointed to by `tail` (Figure 3.11b).
3. The `next` member is null (Figure 3.11c).
4. `tail->prev->next` is set to the value of `tail` so that this node points to the last node in the list (Figure 3.11d), but now the new node becomes the last node (Figure 3.11e).

FIGURE 3.11 Adding a new node at the end of a doubly linked list

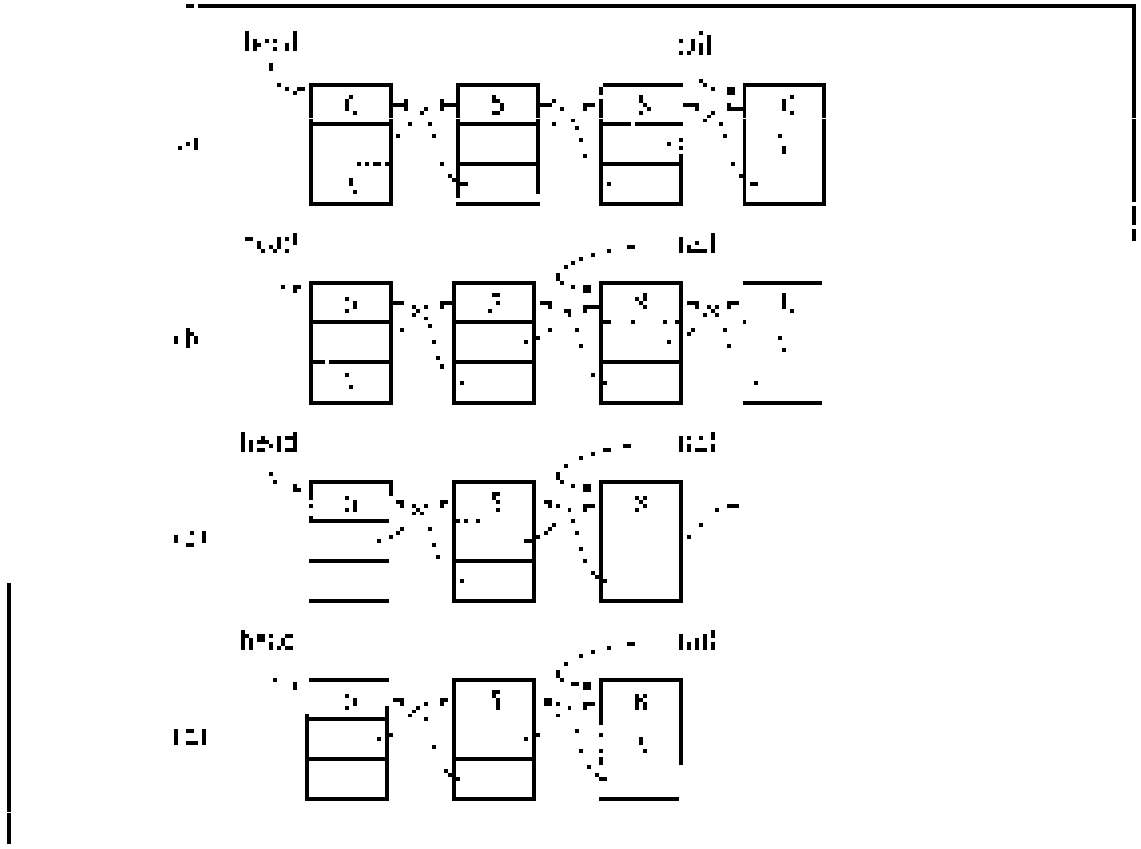


c. Set `next` in the new node to point to the new node (Figure 3.11c). But the new node is not yet accessible from the predecessor, currently `p`.

d. Set next member of the predecessor to point to the new node (Figure 3.11d).

A special case occurs in the last step if it is assumed in this step that the newly created node has a predecessor, so it accesses its prev member. It should be obvious that for an empty linked list the new node is the only node in the list, and that it has no predecessor. In this case, both `head` and `tail` point to this node, and the last step is to set `p` to `head` to point to this node. Note that step c (setting prev member of

FIGURE 3.12 Deleting a node from the middle of a doubly linked list



the value of `last` is assigned properly because for an empty list, `last` is null. Thus null becomes the value of the `previous` field of the new node.

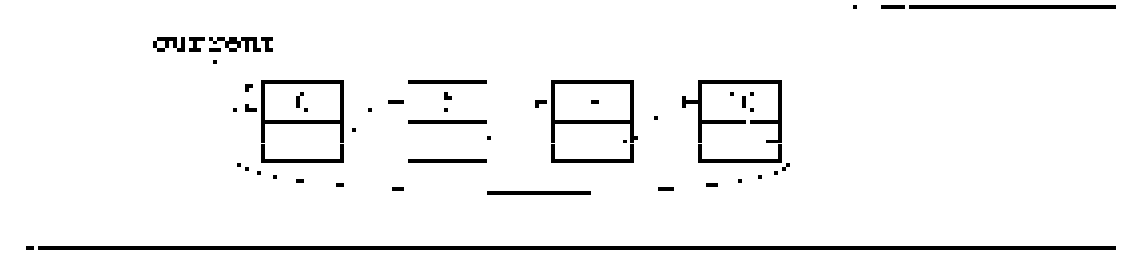
Deleting the last node from a doubly linked list is straightforward because there is direct access from the last node to its predecessor node. We need to remove the last node. When deleting the last node from the list, a new value, temporarily variable `old` is set to the value in the node, then `last` is set to its predecessor node, and the last and now redundant node is deleted (Figure 3.13). Finally, the next to last node becomes the last node. Because the new node is the last node, deleting reference `next` is set to null (Figure 3.13). The last step is returning the copy of an element stored in the removed node.

An attempt to delete a node from an empty list may result in a program crash. Therefore, the user has to check whether the list is not empty before making an attempt at deleting the last node from it. A general implementation of the method `deleteLast` for a doubly linked list follows. Assuming that `DoublyLinkedList` is a class, we have

```
public T deleteLast()
{
    if (last != null)
    {
        T old = last.element;
        last = last.previous;
        if (last == null)
            first = null;
        else
            last.next = null;
        return old;
    }
    return null;
}
```

Another special case is the deletion of the only node in a single-node list. In this case, both `first` and `last` are set to null.

FIGURE 3.13 A circular singly linked list.



Because of the immediate access utility of the last node, both `addToEnd()` and `deleteFromEnd()` execute in constant time.

Exercises and projects involving the implementation of circularly linked lists are found at the end of this section. Before we leave singly linked lists, however, we must discuss one more type of linked list that is used by many algorithms and data structures. Circular lists are singly linked lists in which the last node points back to the first node instead of containing a null pointer.

3.3 Circular Lists

A simple circular linked list is shown in Figure 3.13. The list is accessed through a pointer to the last node in the list. Access to the first node in the list is provided by the next field of the last node. Circular lists are useful for applications that access all of the nodes in a list on a regular basis. In such applications, once the current pointer is set to point to a node in the list, all of the nodes in the list can be accessed by repeatedly following the next field. Unlike singly linked lists, where traversal of the list must terminate when the null pointer at the end of the list is encountered, traversal of a circular list can continue indefinitely. Each node will be visited infinitely often unless explicit action is taken to remove nodes from the list.

Figure 3.13 shows a typical circular list. Note that we can access both the first node (`current.next`) and the last node (`current`) in the list in constant time. This makes it easy to add nodes to the end of the list as well as the front. Figure 3.14 shows the circular list after the node containing 42 has been added to the end of the list. Figure 3.15 shows the list after the node containing 42 has been removed from the front of the list. Figure 3.16 shows the list after the node containing 17 has been removed from the end of the list.

```
    void addToEnd(int item) {
        if (isEmpty()) {
            current = new IntNode(item);
            current.next = current;
        }
        else {
            current.next = new IntNode(current.next, item);
            current = current.next;
        }
    }
```

FIGURE 3.14 Inserting a node into a circular singly linked list: (a) and (b) before; (c) and (d) after.

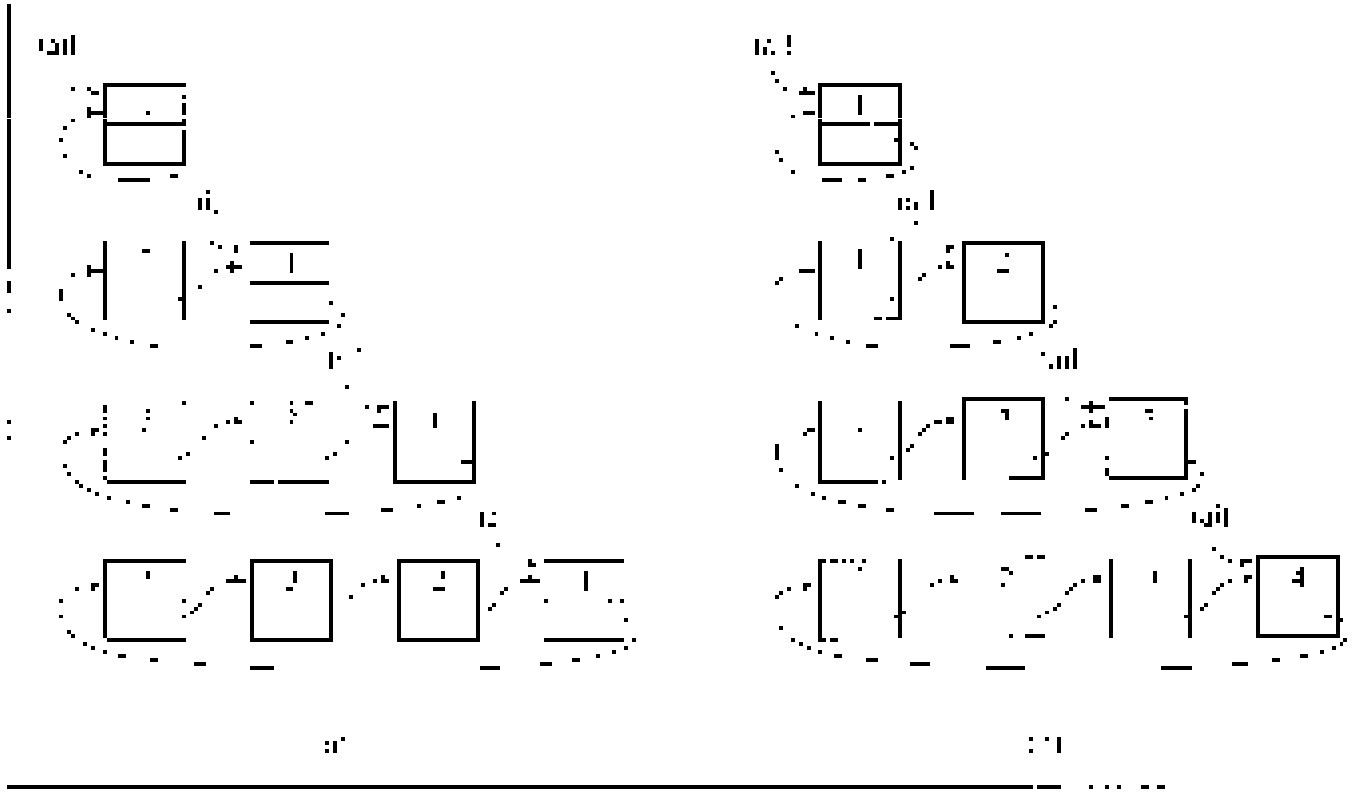
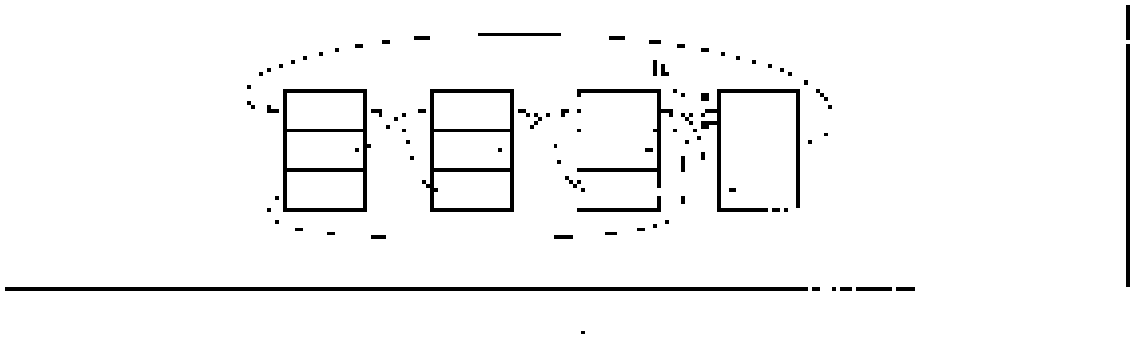


FIGURE 3.15 A circular doubly linked list.



The problem with this approach is that it will not work properly for the first node in the list. This is because the predecessor of the first node is the last node in the list, and we do not know what that node is. To delete the first node, we must traverse the entire list to find it. Thus, deleting the last node in a circular singly linked list is inefficient. To avoid the problem just described, we can add a dummy node to the list. This node does not store any data but serves only as a placeholder to indicate where the list begins and ends. The empty list is no longer represented by `NULL`; instead, it consists of the dummy node. Although a dummy node represents wasted space, we do not usually have to worry about wasting space unless we are working with many small lists. The benefit of using a dummy node is that it always simplifies the code. Figure 3.16 shows the empty list and the list containing two elements.

Circular lists can be traversed starting at any node in the list. Given a pointer `current` to the current node, we can move to the next node in the list with the statement

```
current = current->next;
```

Similarly, we can move to the previous node in the doubly linked list with the statement

```
current = current->prev;
```

Given these two statements, it is easy to write code that traverses the list, either forward or backward, looking for a node with a particular value stored in its `data` member. Figure 3.17 shows the code for inserting a new node into a circular doubly linked list. Deleting a node from a circular doubly linked list is also straightforward. Figure 3.18 shows the code for deleting a node.

FIGURE 3.16 A skip list with (a) evenly spaced nodes of different heights and (b) randomly spaced nodes of different heights. Only the skip list with pointer nodes clearly shown.

[illegible] again to [illegible] the last node in searching the first level ends at node 10, the sublist's first node has a key greater than the search key. Thus, the search moves down to the next lower level. The search is processed until successful or fails at the last node. The searching procedure is given in Figure 3.17.

Searching appears to be efficient. However, the design of skip lists can lead to very inefficient insertion and deletion procedures. To insert a new element, all nodes following the node just inserted have to be restructured, the number of pointers and the values of pointers have to be changed. To understand some of the disadvantages which skip lists suffer with respect to searching and avoid problems when restructuring the lists such as inserting and deleting nodes, the restrictions on the proportions of nodes of different levels have also to be relaxed. Not only the proportion of the number of nodes of different levels is kept, for example, 1/2 like in Figure 3.16, because the list in Figure 3.16 has been created with a random process. In this list, one node out of two nodes has one pointer, one node out of four nodes has two pointers, one node out of eight nodes has three pointers, and so on. The node list is searched exactly the same way as the original list. However, deletion does not require restructuring, and nodes are generated randomly. The distribution of the nodes on different levels is then adequate. How can this be accomplished?

Figure 3.17 An implementation of a skip list.

```
//**********-**********  gen8kipL.h  **********-**********
//                    generic skip list class

const int maxLevel = 4;

template<class K>
class SkipListNode {
public:
    SkipListNode() {
    }
    K key;
    skipListNode *next;
};

template<class K>
class SkipList {
public:
    SkipList();
    void chooseNewLevel();
    int  ChooseLevel();
    K* skipListSearch(const K&);
    void skipListInsert(const K&);
private:
    typedef SkipListNode<K> *nodeRef;
    nodeRef *cut[maxLevel];
    int power[maxLevel];
};

template<class K>
SkipList<K>::SkipList() {
    for (int i = 0; i < maxLevel; i++)
        cut[i] = 0;
}
template<class K>
void SkipList<K>::chooseNewLevel() {
    power[maxLevel-1] = (2 << (maxLevel-1)) - 1;  // 2^maxLevel - 1
    for (int i = maxLevel - 2, j = 0; i >= 0; i--, j++)
        power[i] = power[i+1] - (2 << j);         // 2^(j+1)
}
template<class K>
int SkipList<K>::chooseLevel() {
```

Figure 3.17 (continued)

```
    int i, r = rand() % powers[maxLevel-1] + 1;
    for (i = 1; i < maxLevel; i++)
        if (r < powers[i])
            return i-1; // return a level < the highest level;
    return i-1;         // return the highest level;
}
template<class T>
T* SkipList<T>::skipListSearch(const T& key) {
    nodePtr prev, curr;
    int lvl;                                // find the highest non-null
    for (lvl = maxLevel-1; lvl >= 0 && next[lvl]; lvl--);  // level;
    prev = curr = next[lvl];
    while (1) {
        if (key == curr->key)                       // success if equal;
            return &curr->key;
        else if (key < curr->key) {                 // if smaller, go down
            if (lvl == 0)                           // if possible,
                return 0;
            else if (curr == next[lvl])             // by one level
                curr = next[--lvl];                 // starting from the
            else curr = *(prev->next + --lvl);      // predecessor which
        }                                           // was the root;
        else {                                      // if greater,
            prev = curr;                            // go to the next
            if (*(curr->next + lvl) != 0)           // non-null node
                curr = *(curr->next + lvl);         // on the same level
            else {                                  // or to a list on a
                                                    // lower level;
                for (lvl--; lvl >= 0 && *(curr->next + lvl) == 0; lvl--);
                if (lvl >= 0)
                    curr = *(curr->next + lvl);
                else return 0;
            }
        }
    }
}
template<class T>
void SkipList<T>::skipListInsert(const T& key) {
    nodePtr curr[maxLevel], prev[maxLevel], newNode;
    int lvl, i;
```

FIGURE 3.17 (continued)

```
    curr[maxLevel-1] = root[maxLevel-1];
    prev[maxLevel-1] = 0;
    for (lvl = maxLevel - 1; lvl >= 0; lvl--) {
        while (curr[lvl] && curr[lvl]->key < key) { // go to the next
            prev[lvl] = curr[lvl];                  // at smaller;
            curr[lvl] = *(curr[lvl]->next + lvl);
        }
        if (curr[lvl] && curr[lvl]->key == key)     // don't include
            return;                                 // duplicates;
        if (lvl > 0)                                // go one level down
            if (prev[lvl] == 0) {                   // if not the lowest
                curr[lvl-1] = root[lvl-1];          // level, using a link
                prev[lvl-1] = 0;                    // either from the root
            }
            else {                                  // or from the predecessor;
                curr[lvl-1] = *(prev[lvl]->next + lvl-1);
                prev[lvl-1] = prev[lvl];
            }
    }
    lvl = chooseLevel();        // generate randomly level for newNode;
    newNode = new SkipListNode<T>;
    newNode->next = new nodePtr[sizeof(nodePtr) * (lvl+1)];
    newNode->key  = key;
    for (i = 0; i <= lvl; i++) {            // initialize next fields of
        *(newNode->next + i) = curr[i];     // newNode and reset to newNode
        if (prev[i] == 0)                   // either fields of the root
            root[i] = newNode;              // or next fields of newNode's
        else *(prev[i]->next + i) = newNode;    // predecessors;
    }
}
```

Assume that *maxLevel* = 4. For 15 elements, the required number of one pointer nodes is eight, two pointer nodes is four, three pointer nodes is two, and four pointer nodes is one. Each time a node is inserted, a random number r between 1 and 15 is generated, and if $r \leq 8$, then a node of level one is inserted. If $r \leq 12$, a second level node is inserted, if $r \leq 14$, it is a third level node, and if $r = 15$, the node of level four is generated and inserted. If *maxLevel* = 5, then for 31 elements the correspondence between the value of r and the level of node is as follows:

r	Level of Node to Be Inserted
31	5
29–30	4
25–28	3
17–24	2
1–16	1

To determine such a correspondence between r and the level of node for any maxLevel, the function chooseLevel() initializes the array powers[] by putting lower bounds for each range. For example, for maxLevel = 4, the array is {1, 9, 13, 15}, and for maxLevel = 5, it is {1, 17, 25, 29, 31}. chooseLevel() uses powers[] to determine the level of the node about to be inserted. Figure 3.17 contains the code for chooseLevel() and skipInsert(). Note that the levels range between 0 and maxLevel - 1 and not between 1 and maxLevel so that the array indexes can be used as levels. For example, the first level is level zero.

But we also have to address the question of implementing a node. The easiest way is to make each node have maxLevel pointers, but this is wasteful. We need only as many pointers per node as the level of the node requires. To accomplish this, the next member of each node is not a pointer to the next node, but to an array of pointer(s) to the next node(s). The size of this array is determined by the level of the node. The SkipListNode and a SkipList classes are defined, as in Figure 3.17. In this way, the list in Figure 3.16b is really a list whose first four nodes are shown in Figure 3.16c. Only now can an inserting procedure be implemented, as in Figure 3.17.

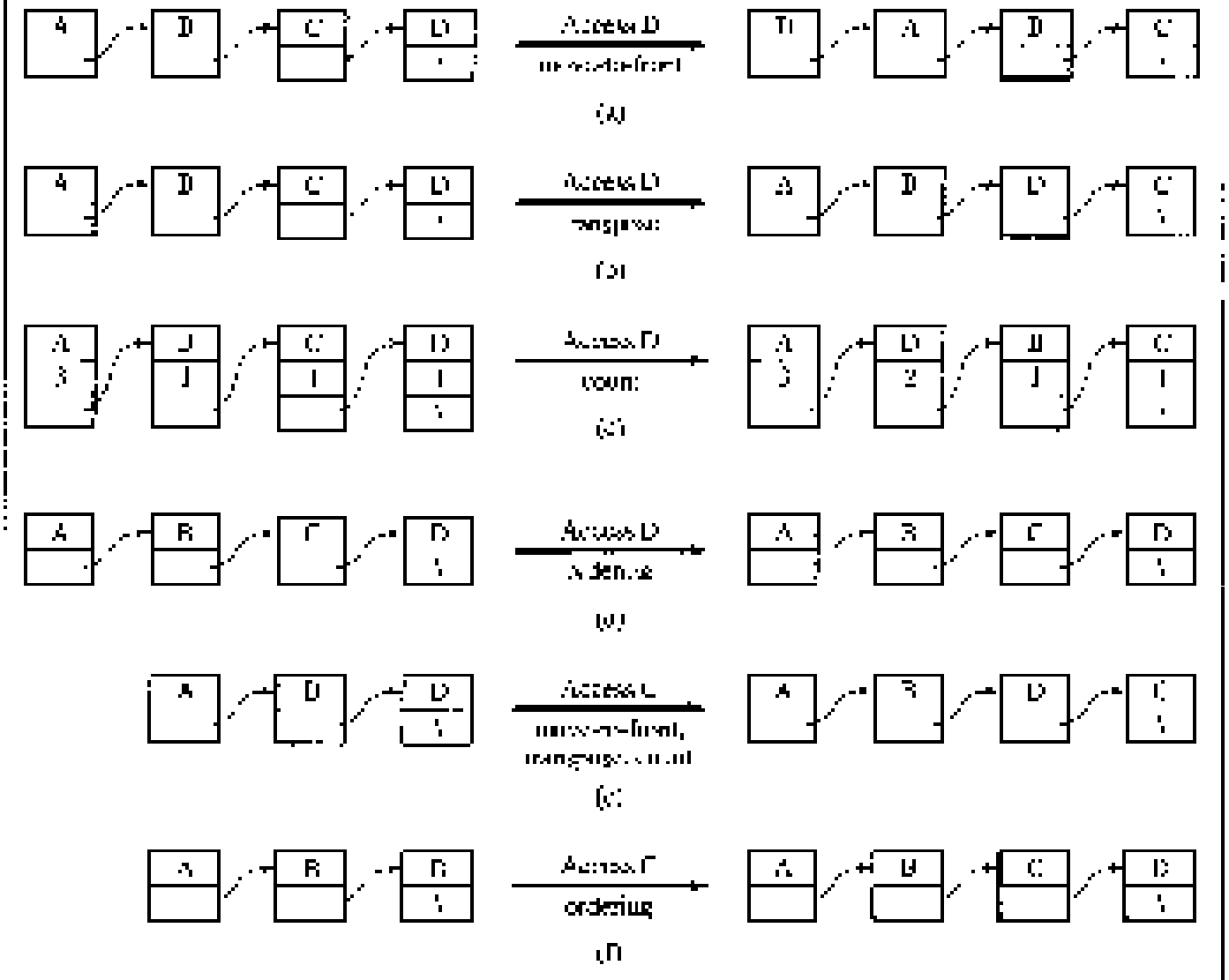
How efficient are skip lists? In the ideal situation, which is exemplified by the list in Figure 3.16a, the search time is $O(\lg n)$. In the worst situation, when all nodes are on the same level, the skip list turns into a regular singly linked list, and the search time is $O(n)$. However, the latter situation is unlikely to occur; in the random skip list, the search time is of the same order as the best case, that is, $O(\lg n)$. This is an improvement over the efficiency of search in regular linked lists. It also turns out that skip lists fare extremely well in comparison with more sophisticated data structures, such as self-adjusting trees or AVL trees (cf. Sections 6.2.2, 6.8), and therefore they are a viable alternative to these data structures (see also the table in Figure 3.20).

3.5 Self-Organizing Lists

The introduction of skip lists was motivated by the need to speed up the searching process. Although singly and doubly linked lists require sequential search to locate an element or to see that it is not in the list, we can improve the efficiency of the search by dynamically reorganizing the list in a certain manner. This organization depends on the configuration of data; thus, the stream of data requires reorganizing the nodes already on the list. There are many different ways to organize the lists, and this section describes four of them.

1. *Move to front method.* After the desired element is located, put it at the beginning of the list.

FIGURE 3.18 Accessing an element in a linked list and changes to the list depending on the self-organization technique applied: (a) move-to-front method, (b) transpose method, (c) count method, and (d) ordering method, in particular alphabetical ordering which leads to most comps. In the case when the desired element is not in the list, (e) the first three methods add a new node with this element at the end of the list and (f) the ordering method maintains an order on the list.



2. *Transpose method.* After the desired element is located, swap it with its predecessor unless it is at the head of the list.
3. *Count method.* Order the list by the number of times elements are being accessed.
4. *Ordering method.* Order the list using certain criteria useful for information retrieval.

In the first three methods, new information is stored in a node added to the end of the list (Figure 3.18e); in the fourth method, new information is stored in a node inserted somewhere in the list to maintain the order of the list (Figure 3.18f). An

FIGURE 3.19 Processing the stream of data: A C B C D A D A C A C C E B, by different methods of organizing linked lists. Linked lists are presented in an abbreviated form; for example, the transformation shown in Figure 3.18a is abbreviated as transforming list A B C D into list D A B C.

element searched for	plain	move-to-front	transpose	count	ordering
A:	A	A	A	A	A
C:	A C	A C	A C	A C	A C
B:	A C B	A C B	A C B	A C B	A B C
C:	A C B	C A B	C A B	C A B	A B C
D:	A C B D	C A B D	C A B D	C A B D	A B C D
A:	A C B D	A C B D	A C B D	C A B D	A B C D
D:	A C B D	D A C B	A C D B	D C A B	A B C D
A:	A C B D	A D C B	A C D B	A D C B	A B C D
C:	A C B D	C A D B	C A D B	C A D B	A B C D
A:	A C B D	A C D B	A C D B	A C D B	A B C D
C:	A C B D	C A D B	C A D B	A C D B	A B C D
C:	A C B D	C A D B	C A D B	C A D B	A B C D
E:	A C B D E	C A D B E	C A D E B	C A D B E	A B C D E
B:	A C B D E	B C A D E	C B A D E	B C A D E	A B C D E

example of searching for elements in a list organized by these different methods is shown in Figure 3.19.

With the first three methods, we try to locate the elements most likely to be looked for near the beginning of the list, most explicitly with the move-to-front method and most carefully with the transpose method. The ordering method already uses some properties inherent in the information stored in the list. For example, if we are storing nodes pertaining to people, then the list can be organized alphabetically by the name of the person or the city or in ascending or descending order using, say, birthday or salary. This is particularly advantageous when searching for information which is not in the list, since the search can terminate without scanning the entire list. Searching all the nodes of the list, however, is necessary in such cases using the other three methods. The count method can be subsumed in the category of the ordering methods if frequency is part of the information. In many cases, however, the count itself is an additional piece of information required solely to move nodes in the list; hence, it may be considered "natural" to the information at hand.

Analyses of the efficiency of these methods customarily compare their efficiency to that of optimal *static ordering*. With this ordering, all the data are already ordered by the frequency of their occurrence in the history of data so that the list is used only for searching, not for inserting new items. Therefore, this approach requires two passes through the body of data: one to build the list and another to use it for search access.

To experimentally measure the efficiency of these methods, the number of actual comparisons was compared to the maximum number of possible comparisons. The latter number is calculated by adding the length of the list at the moment of processing each element. For example, in the table in Figure 3.19, the body of data contains 24 letters, 5 of them being different, which causes that 24 letters were processed. The length of the list before processing each letter is recorded, and the result, $0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 5 = 66$, is used to compare the number of all these comparisons to this combined length. In this way, we know what percentage of the list was scanned during the entire process. For all the list organizing methods except optimal ordering, this combined length is the same; only the number of comparisons can change. For example, when using the move-to-front technique for the data in the table in Figure 3.19, 30 comparisons were made, which is 71.7% when compared to 45. The latter number gives the most possible cost, the combined length of intermediate lists every time all the nodes in the list are located. Each search with the reorganization requires only 20 comparisons, which is 65.2%.

These samples are in agreement with theoretical analyses which indicate that count and move-to-front methods are, in the long run, at most twice as costly as the optimal static ordering. The transpose method approaches, in the long run, the cost of the move-to-front method. In particular, with amortized analysis it can be established that the cost of accessing a list element with the move-to-front method is at most twice the cost of accessing this element on the list that uses optimal static ordering.

In a proof of this statement, the concept of inversion is used. For two lists containing the same elements, an inversion is defined to be a pair of elements (x, y) such that on one list x precedes y and on the other list y precedes x. For example, the list (C, B, D, A) has four inversions with respect to list (A, B, C, D): (C, A), (B, A), (D, A), and (C, B). Define the amortized cost to be the sum of actual cost and the difference between the number of inversions before accessing an element and after accessing it,

$$amCost(x) = cost(x) + Inversions\,Before(x) - Inversions\,After(x)$$

To assess this number, consider an optimal list $OL = (A, B, C, D)$ and a move-to-front list $MTF = (C, B, D, A)$. The access of elements usually changes the number of inversions. Let $displaced(x)$ be the number of elements preceding x in MTF but following x in OL. For example, $displaced(A) = 3$, $displaced(B) = 1$, $displaced(C) = 0$, and $displaced(D) = 0$. If $pos_{MTF}(x)$ is the current position of x in MTF, then $pos_{MTF}(x) - 1 - displaced(x)$ is the number of elements preceding x in both lists. It is easy to see that for D this number equals 2, and for the remaining elements it is 0. Now accessing an element x and moving it to the front of MTF creates $pos_{MTF}(x) - 1 - displaced(x)$ new inversions and removes $displaced(x)$ older inversions so that the amortized time to access x is

$$amCost(x) = pos_{MTF}(x) + pos_{MTF}(x) - 1 = depth_{MTF}(x) + depth_{MTF}(x)$$

$$2(pos_{MTF}(x) - depth_{MTF}(x)) - 1$$

where $cost(x) = pos_{MTF}(x)$. Accessing A transforms MTF = (C, B, D, A) into (A, C, B, D) and $amCost(A) = 2(4 - 1) - 1 = 5$. Then B is the new list is (B, C, D, A) and $amCost(B) = 2(2 - 1) - 1 = 1$. For C, the list does not change and $amCost(C) = 2(1 - 0) - 1 = 1$. Finally, for D, the new list is (D, C, B, A) and $amCost(D) = 2(3 - 0) - 1 = 5$. However, the number of comparisons preceding x in the two lists is not precisely the number of all the elements preceding x in MTF because $pos_{MTF}(x) = 1 + depth_{MTF}(x) + pos_{OL}(x) - 1$, so that

$$amCost(x) \le 2pos_{OL}(x) - 1$$

The amortized cost of accessing an element x in MTF is in excess of $pos_{OL}(x) - 1$ units from actual cost of access on OL. This excess is used to cover an additional cost of accessing elements in MTF for which $pos_{MTF}(x) > pos_{OL}(x)$; that is, elements that require more accesses on MTF than on OL.

It is important to stress that the amortized costs of single operations are meaningful in the context of sequences of operations. A cost of an isolated operation may seldom equal its amortized cost; however, in a sufficiently long sequence of accesses, each access on the average takes at most $2pos_{OL}(x) - 1$ time.

Figure 5.20 contains sample runs of the self-organizing lists. The first two columns of numbers refer to files containing programs, and the remaining columns refer to files containing English text. Except for alphabetical ordering, all methods improve their efficiency with the size of the file. The move-to-front and count methods are almost the same in their efficiency and both outperform the transpose, pairs, and ordering methods. The poor performance for smaller files is due to the fact that all of the methods are busy including new words in the lists, which requires an exhaustive search of the lists. Later, the methods concentrate on organizing the lists to reduce the number of searches. The table in Figure 5.20 also includes data for a skip list. There is an even-balancing difference between the skip list's efficiency compared to the other methods. However, keep in mind that in the table in Figure 5.20, only comparisons of data are included with no indication of the other operations needed for execution of the analyzed methods. In particular, there is no indication of how many pointers are used and relinked; which, when included, may make the difference between various methods less dramatic.

These sample runs show that for lists of modest size, the linked list suffices. With the increase in the amount of data and in the frequency with which they have to be accessed, more sophisticated methods and data structures need to be used.

5.6 Sparse Tables

In many applications, the choice of a table seems to be the most natural one, but space considerations may preclude this choice. This is particularly true if only a small fraction of the table is actually used. A table of this type is called a *sparse table* since the

FIGURE 3.20 Measuring the efficiency of different methods using formula (number of data comparisons)*(combined length) expressed in percentages.

Different Words/All Words	1-9/423	550/2647	156/347	609/1510	1163/5866	2013/33065
Optimal	36.4	17.6	28.5	24.3	16.2	10.0
Hash	21.6	70.3	20.2	6.7	5.7	25.1
Move-to-Front	19.5	31.3	21.2	31.5	20.5	18.4
Transpose	28.5	53.3	28.5	63.1	46.4	12.9
Count	31.6	34.1	61.2	54.7	32.6	19.5
Alphabetic Order	45.6	53.7	30.9	45.0	50.4	50.1
Skip List	23.1	5.5	5.1	5.6	4.2	3.2

table is presented already by rows and most of its cells are empty. In this case, the table can be replaced by a structure of linked lists.

As an example, consider the problem of storing grades for all students in a university for a certain semester. Assume that there are 8000 students and 300 classes. A natural implementation is a two-dimensional array grades where student numbers are indexes of the columns and class numbers the indexes of the rows (see Figure 3.21). An association of student names and numbers is represented by the one-dimensional array students and an association of class names and numbers by the array classes. The names do not have to be ordered. If order is required, then another array can be used where each array element is comprised of a record with two fields, name and number,[1] or the original array can be sorted each time an order is required. This, however, leads to the constant reorganization of grades and is not recommended.

Each cell of grades stores a grade obtained by each student after finishing a class. If signed grades such as A−, B+, or C are used, then two bytes are required to store each grade. To reduce the table size by one-half, the array gradeCodes in Figure 3.21c associates each grade with a letter which requires only one byte of storage.

The entire table (Figure 3.21d) occupies 8000 students · 300 classes · 2 byte = 2.4 million bytes. This table is very large but is sparsely populated by grades. Assuming that, on the average, students take four classes a semester, each column of the table has only four cells occupied by grades, and the rest of the cells, 296 cells or 98.7%, are unoccupied and wasted.

A better solution is to use two 2-dimensional arrays. classesTaken represents all the classes taken by every student and studentsInClasses represents all students participating in each class (see Figure 3.22). A cell of each table is an object with two data

[1] This is similar to an inverted table.

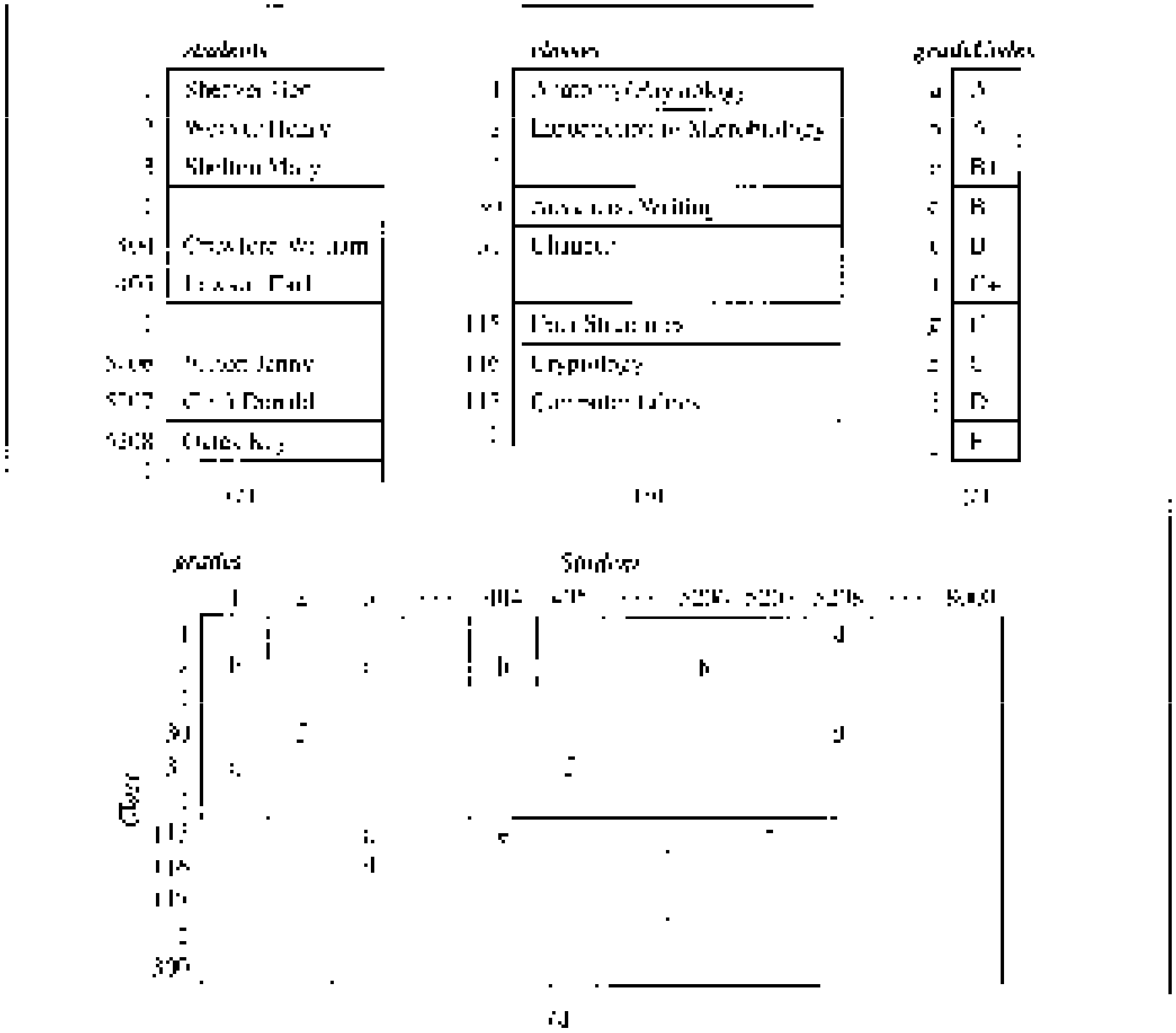
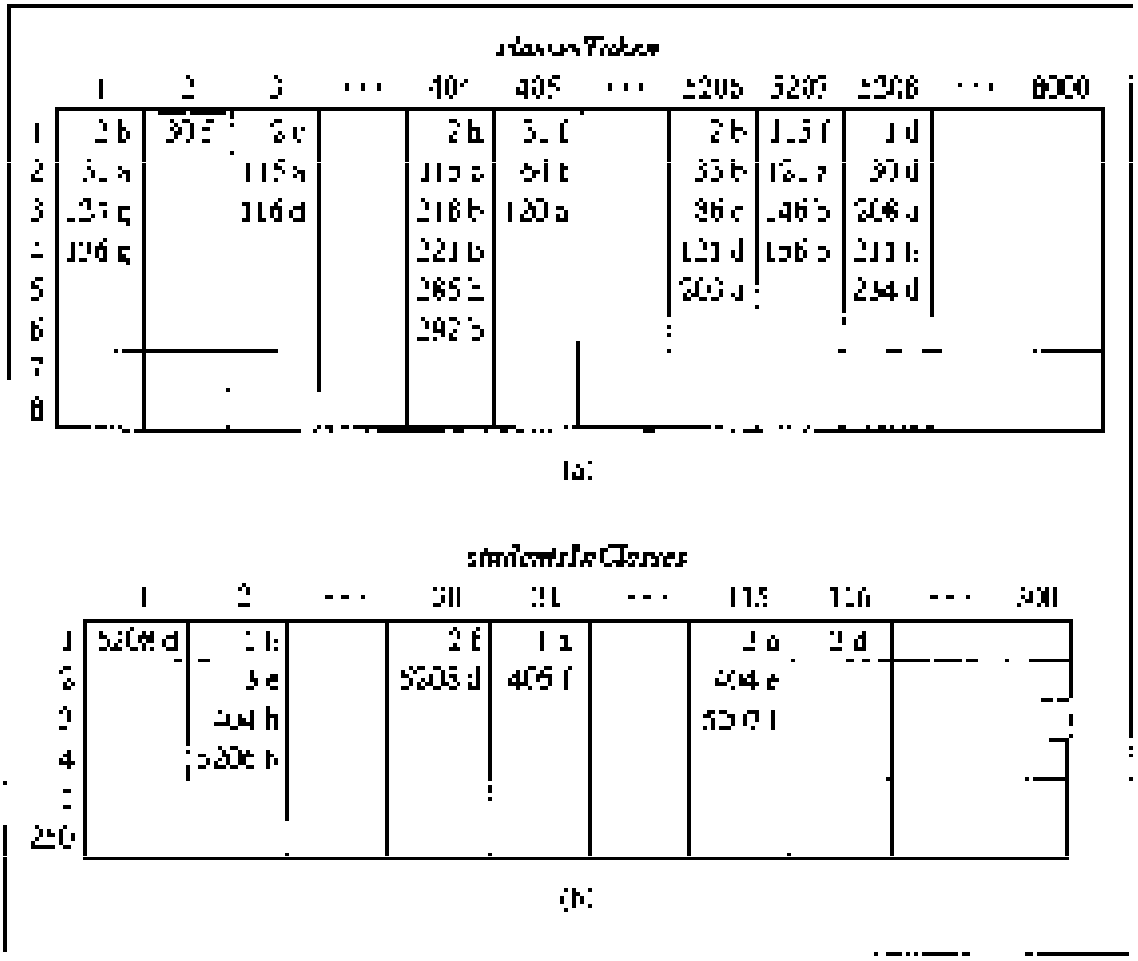


FIGURE 3.21 Arrays and sparse tables used for storing student grades.

enrollments. Since each class can be studied by many students, and each student can study many classes, we will need a table to store the grades for each student in each class. For example, if there are 250 students and 30 classes, then we will need a table with $250 \times 30 = 7,500$ entries. Since each entry in the table will store a single character, this table will require 7,500 bytes of memory.

Assume that the university at which this program is being implemented requires students to study at most 40 classes. With this assumption, we can store the grades for each student in a record containing 40 characters. For example, the record for student 1 will be stored in positions 1 through 40 of the table, the record for student 2 will be stored in positions 41 through 80, and so on. This table will require $250 \times 40 = 10,000$ bytes of memory. Although this table requires more than 25 percent more memory than the sparse table, it is much easier to access. The table in Figure 3.21 requires only 1,000 bytes of memory.

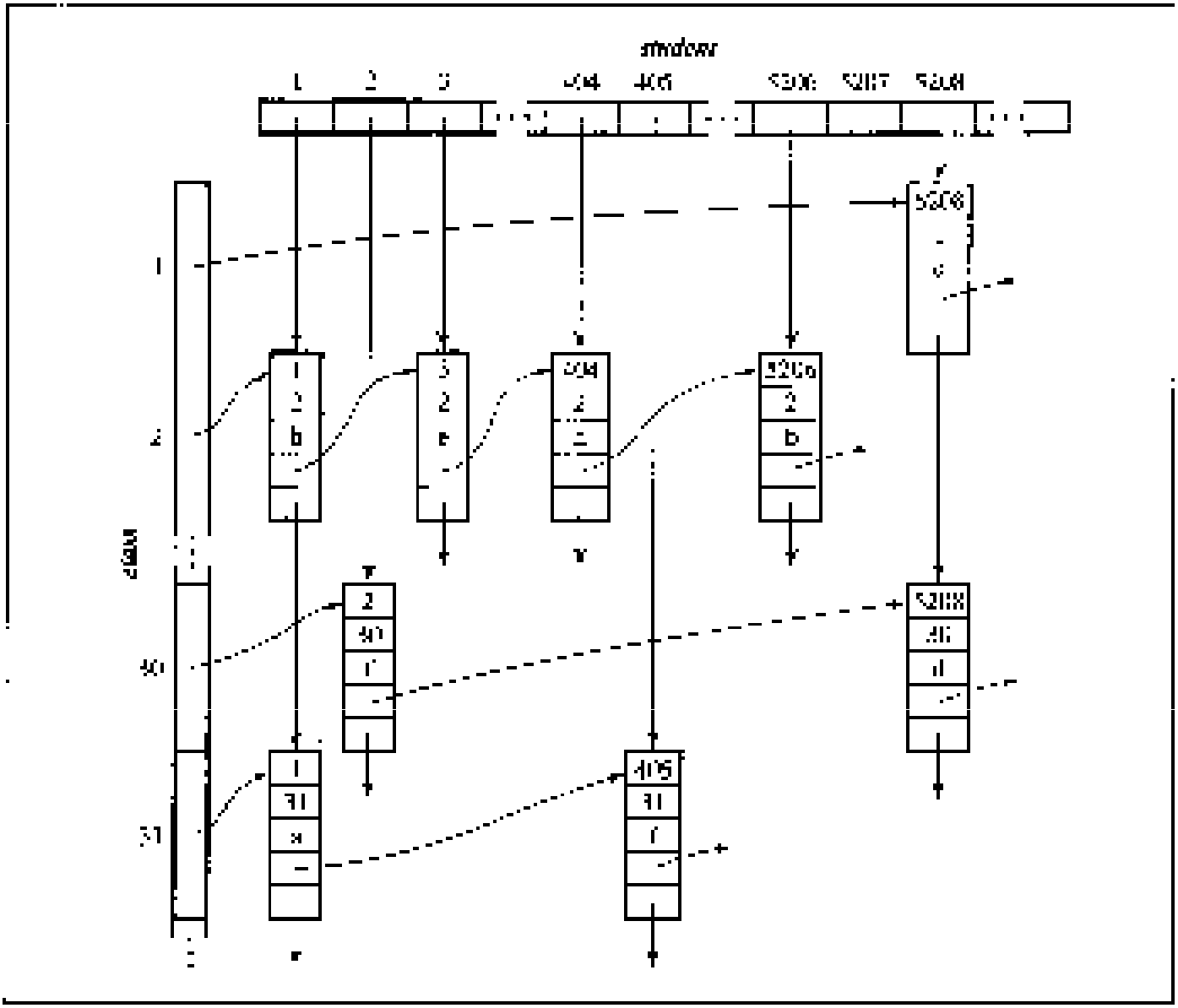
FIGURE 3.22 Two-dimensional arrays for storing student grades.



Although this is a much better implementation than before, it still suffers from a wasteful use of space: seldom if ever will both arrays be full since most classes have fewer than 250 students and most students take fewer than eight classes. This structure is also inflexible: If a class can be taken by more than 250 students, a problem occurs which has to be circumvented in an artificial way. One way is to create a nonexistent class which holds students for the overflowing class. Another way is to recompile the program with a new table size, which may not be practical at a future time. Another more flexible solution is needed that uses space frugally.

Two one-dimensional arrays of linked lists can be used, as in Figure 3.23. Each cell of the array class is a pointer to a linked list of students taking a class, and each cell of the array studentClasses indicates a linked list of classes taken by a student. The linked lists contain nodes of five data members: student number, class number, grade, a pointer to the next student, and a pointer to the next class. Assuming that each pointer requires only two bytes, one node occupies nine bytes, and the entire structure can be stored in 10000 students · 4 classes (on the average) · 9 bytes = 288,000 bytes, which is approximately 11% of the space required for the first implementation and about 70% of the space of the second. No space is used unnecessarily, there is no restriction imposed on the number of students per class, and the lists of students taking a class can be processed randomly.

FIGURE 3.23 Student grades implemented using linked lists



3.7 LISTS IN THE STANDARD TEMPLATE LIBRARY

The list sequence container is an implementation of various operations on the nodes of a linked list. The STL implements a list as a generic doubly linked list with pointers to the head and to the tail. An instance of such a list that stores integers is presented in Figure 3.9.

The class `list` can be used in a program only if it is included with the instruction

```
#include <list>
```

The member functions included in the list container are presented in Figure 3.24.

TABLE 3.24 An alphabetical list of member functions in the class list.

Member Function	Action and Return Value
`void assign(first, last);`	remove all the nodes in the list and insert in their place the elements in the range indicated by iterators first and last.
`void assign(n, val = T());`	remove all the nodes in the list and insert in their place n elements with value val. (If val is not provided, default constructor T() is used.)
`T& back();`	return the element in the last node of the list
`const T& back() const`	return the element in the last node of the list
`iterator begin();`	return an iterator that references the first node of the list
`const_iterator begin() const`	return a const iterator that references the first node of the list
`void clear();`	remove all the nodes in the list
`bool empty() const`	return true if the list includes no nodes and false otherwise
`iterator end();`	return an iterator to the imaginary node at the end of the list
`const_iterator end() const`	return a const iterator to the imaginary node at the end of the list
`iterator/void erase(i);`	remove the node referenced by iterator i
`iterator/void erase(first, last);`	remove the nodes in the range indicated by iterators first and last
`T& front();`	return the element in the first node of the list
`const T& front() const`	return the element in the first node of the list
`iterator insert(i, e = T());`	insert e before the node referenced by iterator i and return an iterator referencing the new node
`void insert(i, n, e);`	insert n copies of e before the node referenced by iterator i
`void insert(i, first, last);`	insert elements from the range indicated by first and last before the node referenced by iterator i. The elements are copied from the range indicated by iterators first and last.
`list();`	construct an empty list
`list(n, val = T());`	construct a list with n copies of val (or T())
`list(first, last);`	construct a list with the elements from the range indicated by iterators first and last
`list(list);`	copy constructor
`size_type max_size() const`	return the maximum number of nodes for the list
`void merge(list);`	for the sorted current list and list, remove all nodes from list and insert them in the sorted order in the current list
`void merge(list, f);`	for the sorted current list and list, remove all nodes from list and insert them in the current list. The sorted order is given by the two argument Boolean function f().

FIGURE 3.24 (continued)

`void pop_back()`	remove the last node of the list
`void pop_front()`	remove the first node of the list
`void push_front(el)`	insert el at the head of the list
`void push_back(el)`	insert el at the end of the list
`reverse_iterator rbegin()`	return an iterator that references the last node of the list
`const_reverse_iterator rbegin() const`	return a const iterator that references the last node of the list
`void remove(el)`	remove from the list all the nodes that include el
`void remove_if(f)`	remove the nodes for which a one argument Boolean function f() returns true
`reverse_iterator rend()`	return an iterator that is before the first node of the list
`const_reverse_iterator rend() const`	return a const iterator that is before the first node of the list
`void resize(n, el = T())`	make the list have n nodes by adding n - size() more nodes with element el or by discarding size() - n nodes from the end of the list
`size_type size() const`	return the number of nodes in the list
`void sort()`	sort elements of the list in ascending order
`void sort(f)`	sort elements of the list in the order specified by a two-argument Boolean function f()
`void splice(i, lst)`	remove the nodes of list lst and insert them into the list before the position referenced by iterator i
`void splice(i, lst, j)`	remove from list lst the node referenced by iterator j and insert it into the list before the position referenced by iterator i
`void splice(i, lst, first, last)`	remove from list lst the nodes in the range indicated by iterators first and last and insert them into the list before the position referenced by iterator i
`void swap(lst)`	swap the contents of the list with the contents of list lst
`void unique()`	remove duplicate elements from the sorted list
`void unique(f)`	remove duplicate elements from the sorted list when testing a duplicate as specified by a two-argument Boolean function f()

A new list is generated with the instruction

```
list<T> lst;
```

where T can be any data type. If it is a user-defined type, the type must also include a default constructor which is required for initialization of new nodes. Otherwise, the compiler is unable to compile the member functions with arguments initialized by the default constructor. These include one constructor and functions `resize()`, `assign()`, and one version of `insert()`. Note that this problem does not arise when creating a list of pointers to user-defined types, as in

```
list<T*> ptrLst;
```

The working of most of the member functions has already been illustrated in the case of the vector container (see Figure 3.4 and the discussion of these functions in Section 3.3). Vector containers have only three member functions not found in the list container (`at()`, `capacity()`, and `reserve()`), but there are a number of list member functions that are not found in the vector container. Examples of their operation are presented in Figure 3.25.

Figure 3.25 A program demonstrating the operation of list member functions.

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

void main() {
    list<int> lst1;              // lst1 is empty
    list<int> lst2(3,7);         // lst2 = (7 7 7)
    for (int j = 1; j <= 5; j++) // lst1 = (1 2 3 4 5)
        lst1.push_back(j);
    list<int>::iterator i1 = lst1.begin(), i2 = i1, i3;
    i2++; i2++; i2--;
    list<int> lst3(++i1,i2);     // lst3 = (2 3)
    list<int> lst4(lst1);        // lst4 = (1 2 3 4 5)
    i1 = lst3.begin();
    lst4.splice(++i1,lst2);      // lst2 is empty,
                                 // lst4 = (1 7 7 7 2 3 4 5)
    lst2 = lst1;                 // lst2 = (1 2 3 4 5)
    i2 = lst2.begin();
    lst4.splice(i1,lst2,++i2);   // lst2 = (1 3 4 5),
                                 // lst1 = (1 7 7 7 2 2 3 4 5)
    i2 = lst2.begin();
```

Continues

FIGURE 3.25 (continued)

```
    i1 = i2;
    lst1.splice(i1,lst2,i2,++i2); // lst2 = {3 4 3};
                                  // lst4 = {1 7 7 7 2 1 2 3 4 5}
    lst4.remove(1);               // lst4 = {7 7 7 2 2 3 4 5};
    lst4.sort();                  // lst4 = {2 2 3 4 5 7 7 7}
    lst4.unique();                // lst4 = {2 3 4 5 7};
    lst1.merge(lst2);             // lst1 = {1 2 3 3 4 4 5 5},
                                  // lst2 is empty
    lst3.reverse();               // lst3 = {3 2}
    lst4.reverse();               // lst4 = {7 5 4 3 2}
    lst3.merge(lst4,greaterInt());    // lst3 = {7 5 4 3 3 2 2},
                                      // lst4 is empty
    lst3.remove_if(bind2nd(not_equal_to<int>(),7));// lst3 = {7 7 7}
    lst3.unique(not_equal_to<int>());   // lst3 = {7}
}
```

3.8 DEQUES IN THE STANDARD TEMPLATE LIBRARY

A deque (double-ended queue) is a list that allows fast direct access to both ends of the list, particularly to insert and delete elements. Hence, a deque can be loosely modeled as a doubly linked list with pointer data members head and tail as discussed in Section 3.7. Moreover, as pointed out in the previous section, the container list uses a doubly linked list already. The STL, however, adds another functionality to the deque: namely, easy direct access to any position of the deque, just as in arrays and vectors. Vectors are discussed in Section 1.8. Their poor performance for insertion and deletion of elements at the front, but these operations are quickly performed for doubly linked lists. This means that the STL deque should combine the behavior of a vector and a list.

The member functions of the STL container deque are listed in Figure 3.26. The functions are basically the same as those available for lists, with few exceptions. Deques do not include function splice(), which is specific to list, and functions merge(), remove(), sort(), and unique(), which are also available as algorithms, and list only implements them as member functions. The most significant difference is the function at() (and its equivalent operator[]) that is unavailable in list. The latter function is available in vector, and if we compare the set of member functions in vector (Figure 1.3) and in deque, we see only a few differences. vector does not have pop_front() and push_front(), as does deque, but deque does not include functions capacity() and reserve(), which are available in vector. A few operations are illustrated in Figure 3.27. Note that for lists

FIGURE 3.26 A list of member functions in the class deque.

Member Function	Operation
`void assign(first, last);`	remove all the elements in the deque and insert in their place elements from the range indicated by iterators `first` and `last`
`void assign(n, el = T());`	remove all the elements in the deque and insert `n` copies of `el`
`T& at(n)`	ref. — the element in position `n` of the deque
`const T& at(n) const`	return the element in position `n` of the deque
`T& back()`	return the last element in the deque
`const T& back() const`	return the last element in the deque
`iterator begin()`	return an iterator that references the first element of the deque
`const_iterator begin() const`	return a const iterator that references the first element of the deque
`void clear()`	remove all the elements in the deque
`deque();`	construct an empty deque
`deque(n, el = T())`	construct a deque with `n` copies of element of type `T`; `el` is not provided, default constructor `T()` is used
`deque(deq)`	copy constructor
`deque(first, last)`	construct a deque and initialize it with values from the range indicated by iterators `first` and `last`
`bool empty() const`	return `true` if the deque is empty; return `false` otherwise
`iterator end()`	return an iterator that is past the last element of the deque
`const_iterator end() const`	return a const iterator that is past the last element of the deque
`iterator erase(iterator i)`	remove the element referenced by iterator `i`
`iterator erase(first, last)`	remove the elements in the range indicated by iterators `first` and `last`
`T& front()`	ref. — the first element in the deque
`const T& front() const`	return the first element in the deque
`iterator insert(i, el = T())`	insert `el` before the element indicated by iterator `i` and return an iterator that references the newly inserted element
`void insert(i, n, el);`	insert `n` copies of `el` before the element indicated by iterator `i`

Continued

FIGURE 3.26 (continued)

`void insert(i, first, last)`	inserts elements from list x before element pointed by first and before the element pointed by iterator i; first and last are either const_iterators or const * pointers
`size_type max_size() const`	returns the maximum number of elements in the deque
`T& operator[i]`	subscript operator
`void pop_back()`	removes the last element of the deque
`void pop_front()`	removes the first element of the deque
`void push_back(e)`	inserts e at the end of the deque
`void push_front(e)`	inserts e at the beginning of the deque
`reverse_iterator rbegin()`	returns a reverse iterator that points to the last element of the deque
`const reverse_iterator rbegin() const`	returns a const reverse iterator that points to the last element of the deque
`reverse_iterator rend()`	returns a reverse iterator that is before the first element of the deque
`void resize(n, e = T())`	make the deque have n positions by adding n - size() new positions with element e or by discarding overflowing size() - n positions from the end of the deque
`reverse_iterator rend()`	returns a reverse iterator that is before the first element of the deque
`const reverse_iterator rend() const`	returns a const reverse iterator that is before the first element of the deque
`size_type size() const`	returns the number of elements in the deque
`void swap(dq)`	swaps the content of the deque with the content of another deque dq

only autoincrement and autodecrement were possible for iterators, but for deques we can add any number to iterators. For example, `dq.begin() + 2` is legal for deques, but not for lists.

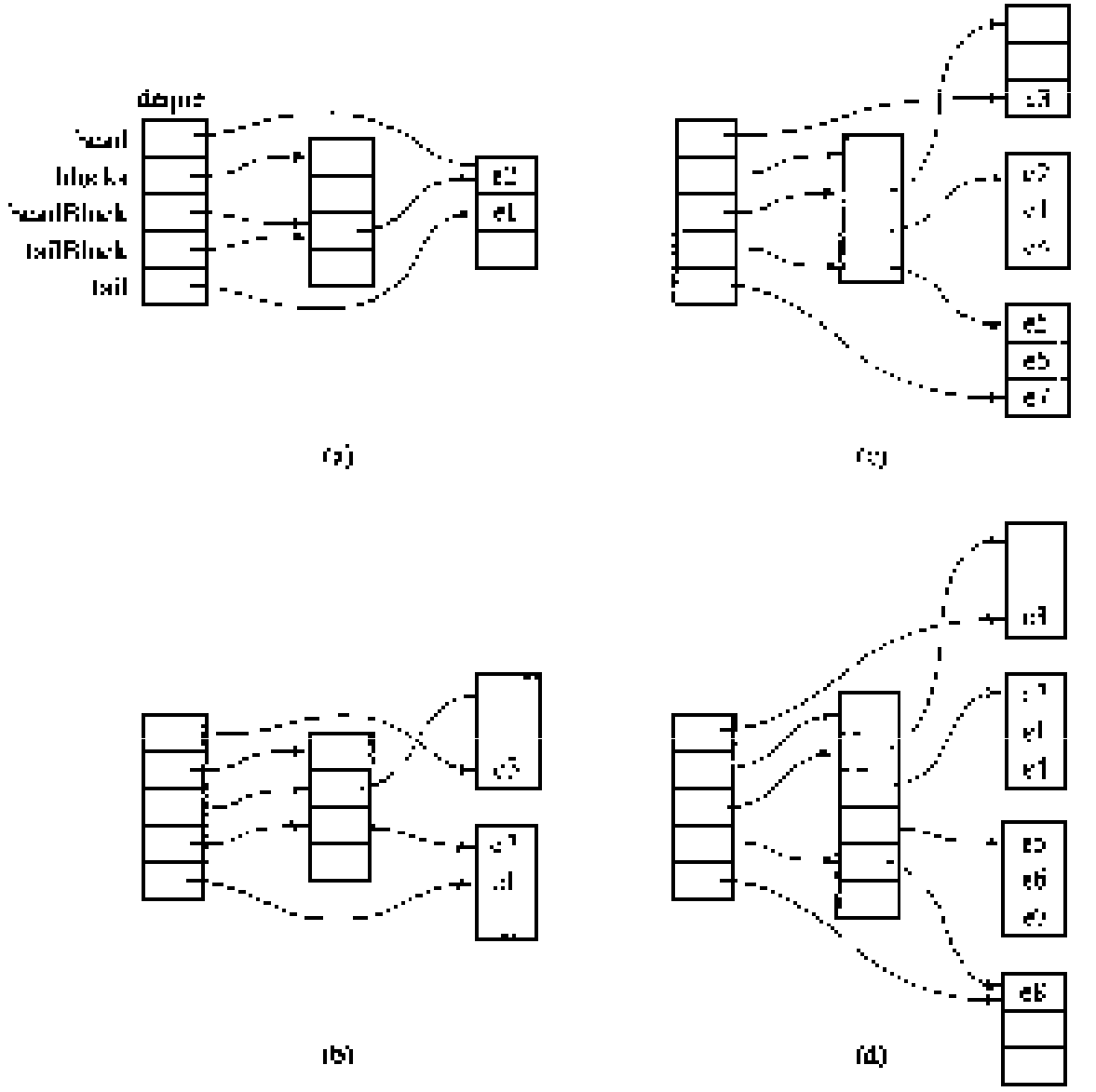
A very interesting aspect of the STL deque is its implementation. Random access can be simulated in doubly linked lists having in the definition of `operator[](int n)` a loop that scans sequentially the list and jumps at the nth node. The STL implementation solves this problem differently. An STL deque is not implemented as a linked list but as an array of pointers to blocks or arrays of data. The number of blocks changes dynamically depending on storage needs, and the size of the array of pointers changes accordingly. (We encounter a similar approach applied in extendible hashing, in Section 11.8.1.)

Program 3.27 A program demonstrating the operation of deque member functions.

```
#include <iostream.h>
#include <algorithm>
#include <deque>

using namespace std;

void main() {
    deque<int> dq1;
    dq1.push_front(1);                          // dq1 = (1)
    dq1.push_front(2);                          // dq1 = (2 1)
    dq1.push_back(3);                           // dq1 = (2 1 3)
    dq1.push_back(4);                           // dq1 = (2 1 3 4)
    deque<int> dq2(dq1.begin()+1,dq1.end()-1);  // dq2 = (1 3)
    dq1[1] = 5;                                 // dq1 = (2 5 3 4)
    dq1.erase(dq1.begin());                     // dq1 = (5 3 4)
    dq1.insert(dq1.end()-1,2,6);                // dq1 = (5 3 6 6 4)
    sort(dq1.begin(),dq1.end());                // dq1 = (3 4 5 6 6)
    deque<int> dq3;
    dq3.insert(dq3.size(),dq2.size());          // dq3 = (0 0 0 0 0 0 0)
    merge(dq2.begin(),dq2.end(),dq1.begin(),dq1.end(),dq3.begin());
    // dq1 = (3 4 5 6 6) and dq2 = (1 3) ==> dq3 = (1 3 3 4 5 6 6)
}
```

To discuss one possible implementation, assume that the array of pointers has four cells and an array of data has three cells, that is, `blockSize = 3`. An object deque includes fields `head`, `tail`, `headBlock`, `tailBlock`, and `blocks`. After execution of `push_front(e1)` and `push_front(e2)` with an initially empty deque, the situation is as in Figure 3.28a. First, the array `blocks` is created and then one data block accessible from a middle cell of `blocks`. Next, `e1` is inserted in the middle of the data block. The subsequent calls place elements consecutively in the first half of the data array. The third call to `push_front()` cannot successfully place `e3` in the current data array, therefore, a new data array is created and `e3` is located in the last cell (Figure 3.28b). Now we execute `push_back()` four times. Element `e4` is placed in an existing data array accessible from deque through `tailBlock`. Elements `e5`, `e6`, and `e7` are placed in a new data block, which also becomes accessible through `tailBlock` (Figure 3.28c). The next call to `push_back()` alters the pointer array `blocks` because the last data block is full and the block is accessible for the last cell of `blocks`. In this case, a new pointer array is created which contains (in this implementation) twice as many cells as the number of data blocks. Next, the pointers from old array `blocks` are copied to the new array, and then a new data block can be created to accommodate

Figure 3.28 Changes in the deque in the process of pushing new elements.



element e8 using insertAt (Figure 3.28d). This is an example of the worst case for adding between whenMoreSize and minUnderSize − 2 cells have to be copied from the old array to the new one. Therefore, in the worst case, the pushing operations takes O(n) time to perform. But assuming that blockSize is a large number, the worst case can be expected to occur very infrequently. Most of the time the pushing operation requires constant time.

Inserting an element into a deque is very simple conceptually. To insert an element at the first half of the deque, the front element is pushed onto the deque, and the elements that should precede the new element are copied to the preceding cell. Then

the new element can be placed in the desired position. To insert an element into the second half of the deque, the last element is pushed onto the deque, and elements that should follow the new element in the deque are copied to the next cell.

With the discussed implementation, a random access can be performed in constant time. For the situation illustrated in Figure 3.25, that is, with declarations

```
T **blocks;
T **headBlock;
T *head;
```

the subscript operator can be overloaded as follows:

```
T operator[] (int n) {
    if (n < blockSize - (head - *headBlock))    // if n is
        return *(head + n);                     // in the first
    else {                                      // block:
        n -= blockSize - (head - *headBlock);
        int q = n / blockSize + 1;
        int r = n % blockSize;
        return *(*(headBlock + q) + r);
    }
}
```

Although access to a particular position requires several arithmetic, dereferencing, and assignment operations, the number of operations is constant for any size of the deque.

3.9 Concluding Remarks

Linked lists have been introduced to overcome limitations of arrays by allowing dynamic allocation of necessary amounts of memory. Also, linked lists allow easy insertion and deletion of information, since such operations have a local impact on the list. To insert a new element at the beginning of an array, all elements in the array have to be shifted to make room for the new item; hence, insertion has a global impact on the array. Deletion is the same. So should we always use linked lists instead of arrays?

Arrays have some advantages over linked lists, mainly that they allow random accessing. To access the tenth node in a linked list, all nine preceding nodes have to be passed. In the array, we can go to the tenth cell immediately. Therefore, if an immediate access of any element is necessary, then an array is a better choice. This was the case with binary search, and it will be the case with most sorting algorithms (see Chapter 9). But if we are constantly accessing only some elements—the first, the second, the last, and the last—and if changing the structure is the core of an algorithm, then using a linked list is a better option. A good example is a queue, which is discussed in the next chapter.

Another advantage in the use of arrays is space. To build items in arrays, the cells have to be of the size of the items. In linked lists, we store one item per node and the

node also includes at least one pointer; in doubly linked lists, the node contains two pointers. For large linked lists, a significant amount of memory is reserved to store the pointers. Therefore, if a problem does not require many shifts of data, then keeping an oversized array may not be wasteful at all if its size is comparable to the amount of space needed for the linked structure storing the same data as the array.

3.10 Case Study: A Library

This case study is a program that can be used in a real library to enable users of books in the library to check out books to people, and to return them.

As this program is a practice in the use of linked lists, it must accomplish its tasks in terms of such lists. But to make the program more interesting, it uses lists of linked lists that also contain cross-references (see Figure 3.20).

First, there could be a list including all authors of all books in the library. However, as authors' names are compared character-by-character, searching such a list can be time-consuming. Searching can be sped up by choosing one of the two following strategies:

- The list can be ordered alphabetically, and the search can be interrupted if we find the name we encounter an author's name greater than the one we are searching for, or if we reach the end of list.
- We can use an array of pointers to the author structures and indexed with letters; each slot of the array points to the linked list of authors whose names start with the same letter.

The best strategy is to combine both approaches. However, in this case study, only the second approach is used, and the reader is urged to amplify the program by adding the first approach. Note that articles a, an, and the at the beginning of the titles should be disregarded during the sorting operation.

The program uses an array `catalog` of all the authors of the books included in the library and an array `people` of all the people who have used the library at least once. Both arrays are indexed with letters so that, for instance, `position catalog['F']` refers to a linked list of all the authors whose name starts with F.

Because we can have several books by the same author, one of the data members of the author node refers to the list of books of this author that can be found in the library. Similarly, because each person can check out several books, the node corresponding to this person contains a reference to the list of books currently checked out by this person. This fact is also indicated by storing the person number of the checked out books in the node pertaining to the person who is taking the book out.

Books can be returned, and that fact should be reflected by removing the appropriate nodes from the list of the checked-out books of the person who returns them. The person member in the node related to the book that is being returned has to be reset to null.

The program defines four classes: `Author`, `Book`, `Person`, and `CheckedOutBook`. To define different types of linked lists, the STL containers are used, in particular, the library `<list>`. But because of the requisite table cross-referencing as shown in

FIGURE 3.29 Linked lists in a library system.

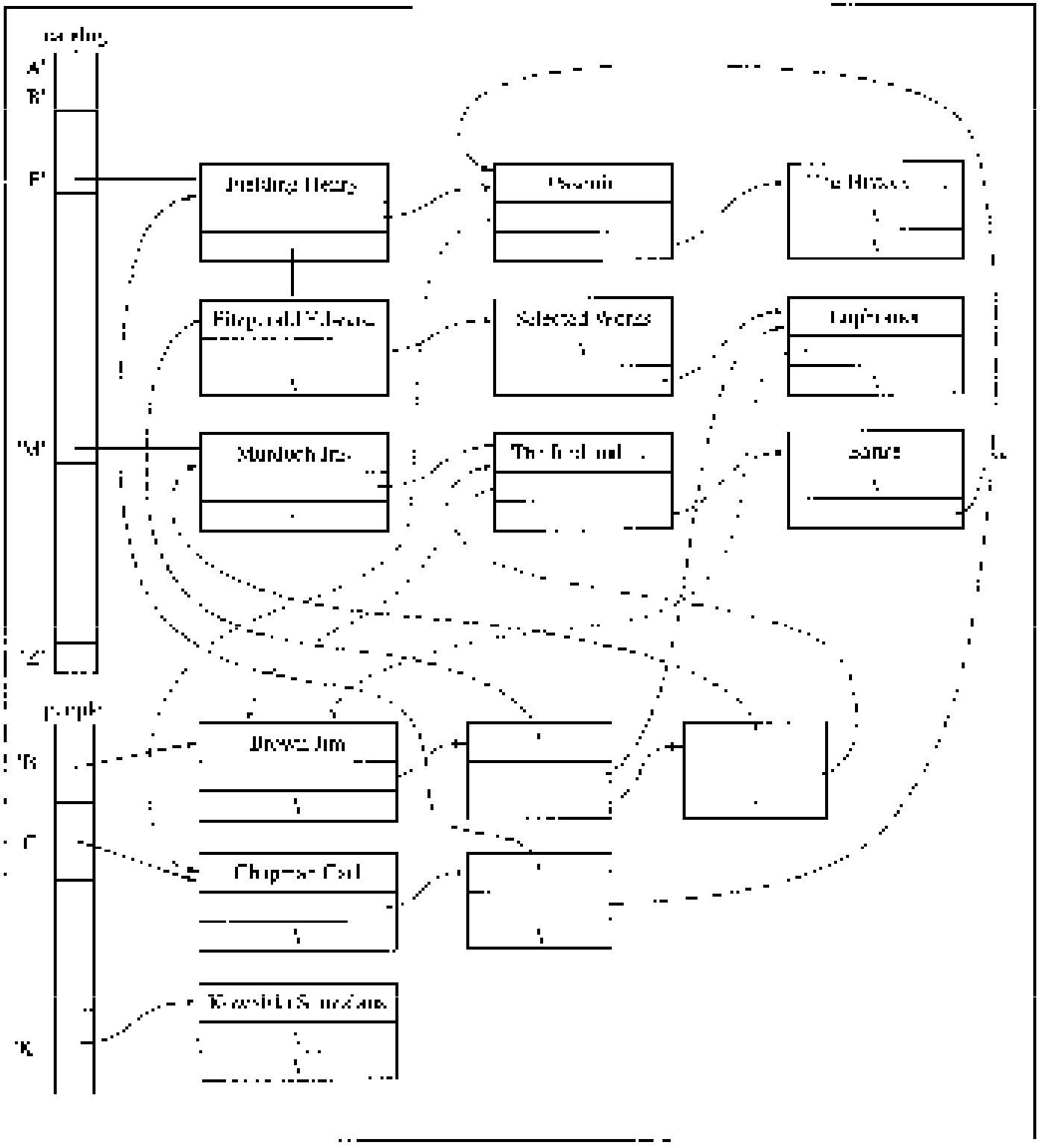


Figure 3.28. In most cases, we do not create lists of objects of a particular class type but lists of pointers to the objects. For example, catalog is not defined as an array of linked lists of Author objects,

```
list<Author> catalog['Z'+1];
```

but as an array of pointers to Author objects

```
list<Author*> catalog['Z'-'A'+1];
```

The only exception is the declaration

```
list<CheckedOutBook> books;
```

in class Person, as this is because the two data members of class CheckedOutBook are pointers already.

Creating lists of pointers to nodes poses certain challenges. In particular, the generic algorithm find() cannot be used because it compares node addresses rather than nodes themselves; hence, finding a particular piece of information is always unsuccessful. To solve the problem, the program uses a generic function find_if(). This takes a list list of pointers to nodes as the first argument and a node el to be searched for as the second argument. Locally, it defines an iterator ref for the list, and for each node of the list, a comparison **ref == el is performed. In this comparison, iterator ref is doubly dereferenced. The left asterisk in **ref dereferences iterator ref because *ref is a node in list currently pointed to by ref, and the right asterisk dereferences the pointer in the node *ref so that **ref is an information node pointed to by a pointer in the node pointed to by ref. This information node is compared to el using an overloaded operator ==.

The same double dereferencing is used in overloading the output operator << for lists.

```
template<class T>
ostream& operator<< (ostream& out, const list<T>& lst) {
    for (list<T>::iterator ref = lst.begin(); ref != lst.end(); ref++)
       out << **ref; // overloaded <<
    return out;
}
```

which requires overloading the same operator for three classes used in the program.

The program allows the user to choose one of the five operations: including a book, checking a book out, returning it, showing the current status of the library, and exiting the program. The operation is chosen after a menu is displayed and a proper number is entered. The cycle of displaying the menu and executing an elected operation ends with choosing the exit option. Here is an example of the screen for a situation shown in Figure 3.28.

```
Library has the following books:

The Lion King
    * Penguin    checked out to Chapman Carl
    * The History of Ice Cream
```

```
Fitzgerald Edward
    * Selected Works
    * Euphranor   checked out to Brown Jim
Murdoch Iris
    * The Red and the Green   checked out to Brown Jim
    * Bruno
    * The Bell

The following people are using the library:

Brown Jim has the following books:
    * F Fitzgerald Edward, Euphranor
    * Murdoch Iris, The Red and the Green
Chapnor Carl has the following books:
    * F Fitzgerald Edward, Euphranor
Kowalski Stanislaus has no books
```

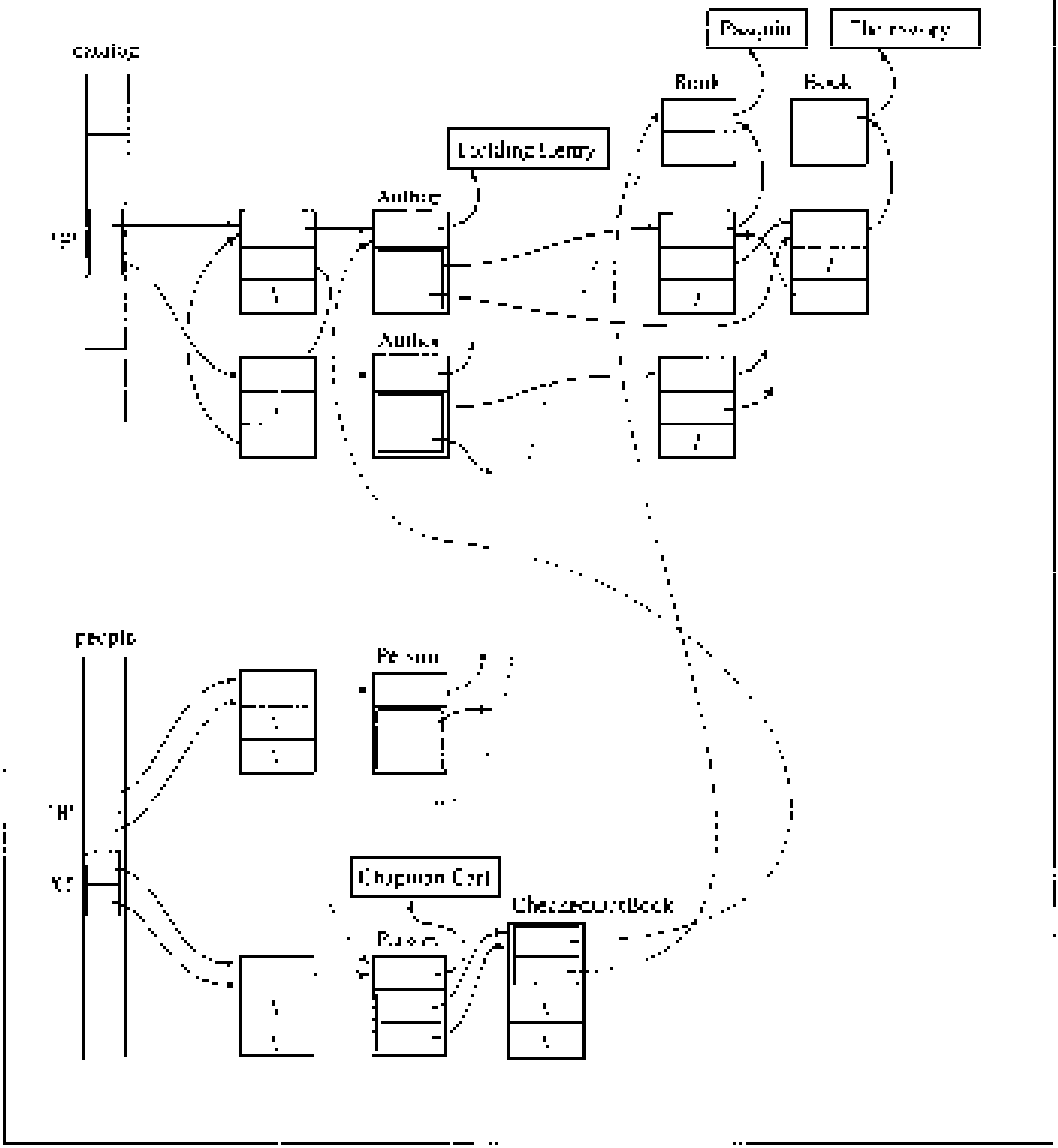
Note that the diagram in Figure 3.29 is somewhat simplified, since strings are not stored directly in structures, only pointers to strings. Hence, technically, each name and title should be shown outside structures with links leading to them. A fragment of Figure 3.29 is shown in Figure 3.30 with implementation details shown more explicitly. Figure 3.31 contains the code for the library program.

3.11 Exercises

1. Assume that a circular doubly linked list has been created, as in Figure 3.22. After each of the following assignments, indicate changes made in the list by showing which links have been modified. The second assignment should make changes in the list modified by the first assignment and so on.

```
list->next->next->next = list->next;

list->prev->prev->prev = list->next->next->next->prev;

list->next->next->next->next->prev = list->prev->prev->prev;

list->next = list->next->next;

list->next->prev->next = list->next->next->next;
```

2. How many nodes does the shortest linked list have? The longest linked list?
3. The linked list in Figure 3.11 was created in Section 3.2 with three assignments. Create this list with only one assignment.
4. Merge two ordered singly linked lists of integers into one ordered list.
5. Delete an arbitrary node in a linked list. Be sure that such a node exists.


Penguin
Book
Author
Author
people
Person
Person
Chairman Card
Person

Listing 3.31 The library program

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

class Person;          // Forward declaration;

class Book {
public:
    Book() {
        pages = 0;
    }
    bool operator== (const Book& bk) const {
        return strcmp(title,bk.title) == 0;
    }
private:
    char *title;
    Person *person;
    friend ostream& operator<< (ostream&,const Book&);
    friend ostream& operator<< (ostream&,const Person&);
    friend class CheckedOutBook;
    friend void CheckInBook();
    friend void checkOutBook();
    friend void returnBook();
};

class Author {
public:
    Author() {
    }
    bool operator== (const Author& rhs) const {
        return strcmp(name,rhs.name) == 0;
    }
private:
    char *name;
    list<Book*> books;
    friend ostream& operator<< (ostream&,const Author&);
    friend ostream& operator<< (ostream&,const Person&);
    friend class CheckedOutBook;
    friend void displayBook();
```

Program 3.31 continued

```
    friend void checkOutBook();
    friend void returnBook();
};

class CheckedOutBook {
public:
    CheckedOutBook(Author *ar = 0, Book *bk = 0) {
        author = ar; book = bk;
    }
    bool operator== (const CheckedOutBook& bk) const {
        return strcmp(author->name,bk.author->name) == 0 &&
               strcmp(book->title,bk.book->title) == 0;
    }
    bool operator!= (const CheckedOutBook& bk) const {
        return !(*this == bk);
    }
    bool operator< (const CheckedOutBook &bk) const {
        if (strcmp(author->name,bk.author->name) == 0)
            return strcmp(book->title,bk.book->title) < 0;
        return strcmp(author->name,bk.author->name) < 0;
    }
    bool operator> (const CheckedOutBook &bk) const {
        return !(*this == bk) && !(*this < bk);
    }
private:
    Author* author;
    Book* book;
    friend ostream& operator<< (ostream&, const Person&);
    friend void checkOutBook();
    friend void returnBook();
};

class Person {
public:
    Person() {
    }
    bool matches(const char *) const {
        return strcmp(name,n) == 0;
    }
private:
    char *name;
```

Filename: 3-31 (continued)

```
    vector<CheckedOutBook> books;
    friend ostream& operator<<(ostream&, const Person&);
    friend ostream& operator<<(ostream&, const Person*);
    friend void checkOutBook();
    friend void returnBook();
};

Book catalog['Z'+1];
Person people['Z'-'A'+1];

template<class Iter>
ostream& operator<<(ostream& out, const pair<Iter,Iter>& lst) {
    for (Iter it = lst.first; it != lst.second; it++)
        out << *it << " // overloaded <<
    return out;
}

ostream& operator<<(ostream& out, const Book* bk) {
    out << "    " << bk->title;
    if (bk->person != 0)
        out << " - checked out to " << bk->person->name; // overloaded <<
    out << endl;
    return out;
}

ostream& operator<<(ostream& out, const Person& p) {
    out << p.name << " // overloaded <<
    return out;
}

ostream& operator<<(ostream& out, const Person* p) {
    out << p->name;
    if (!p->books.empty()) {
        out << " has the following books:\n";
        list<CheckedOutBook>::iterator bk = p->books.begin();
        for ( ; bk != p->books.end(); bk++)
            out << "    " << bk->author->name << ", "
                << bk->book->title << endl;
    }
    else out << " has no books\n";
    return out;
```

Figure 3.31 (continued)

```
}

template<class I1, class I2>
list<I2>::iterator find(const list<I2>& lst, const I1& el) {
    for (list<I2>::iterator ret = lst.begin(); ret != lst.end(); ret++)
        if (*ret == el) // overloaded ==
            break;
    return ret;
}

char* getString(char *name) {
    char s[82], *result;
    cout << name;
    cin.getline(s,80);
    while (strlen(s) == s[81] != '\0') ;  // discard overflow
    result = new char[strlen(s)+1];           // characters;
    for (i = 0; result[i] = toupper(s[i]); i++);
    return result;
}

void status() {
    register int i;
    cout << "Library has the following books:\n\n";
    for (i = 'A'; i <= 'Z'; i++)
        if (!catalog[i].empty())
            cout << catalog[i];
    cout << "\nThe following people are using the library:\n\n";
    for (i = 'A'; i <= 'Z'; i++)
        if (people[i].empty())
            cout << people[i];
}

void initialize() {
    Author *newAuthor = new Author;
    Book *newBook = new Book;
    newAuthor->name = getString("Enter author's name: ");
    newBook->title = getString("Enter title of the book: ");
    list<Author*>::iterator oldAuthor =
        find(people[newAuthor->name[0]], newAuthor);
    if (oldAuthor == people[newAuthor->name[0]].end()) {
        newAuthor->books.push_front(newBook);
```

Figure 3.31 (continued)

```
            catalog[newAuthor.name[0]].push_front(newAuthor);
        }
        else (*oldAuthor).books.push_front(newBook);
    }

    void checkOutBook() {
        Person *person = new Person;
        Author author;
        Book book;
        list<Author>::iterator authorIt;
        list<Book>::iterator bookIt;
        person->name = getString("Enter person's name: ");
        while (true) {
            author.name = getString("Enter author's name: ");
            authorIt = findInCatalog(author.name[0]).authors;
            if (authorIt == catalog[author.name[0]].end()) {
                cout << "Misspelled author's name\n";
            else break;
        }
        while (true) {
            book.title = getString("Enter the title of the book: ");
            bookIt = find(*authorIt->books.begin(),
            if (bookIt == *authorIt->books.end()) {
                cout << "Misspelled title\n";
            else break;
        }
        list<Person>::iterator personIt;
        personIt = find(people[person->name[0]].begin(), people[
        checkedOutBook checkedOutBook(*authorIt->name,
        if (personIt == people[person->name[0]].end()) { // a new person
            person->books.push_front(checkedOutBook);     // in the library
            people[person->name[0]].push_front(person);
            bookIt->person = people[person->name[0]].begin();
        }
        else {
            (*personIt)->books.push_front(checkedOutBook);
            (*bookIt)->person = personIt;
        }
    }

    void returnBook() {
```

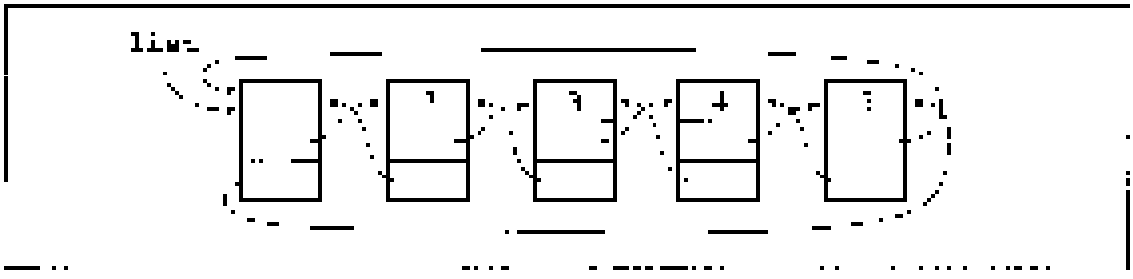
Program 3.31 (continued)

```
    Person person;
    Book book;
    Author author;
    List<Person*>::Iterator personItr;
    List<Book*>::Iterator bookItr;
    List<Author*>::Iterator authItr;
    while (true) {
        screen.cout << "Enter person's name: ";
        personItr = find(people.begin(), people.end(),
        if (personItr == people.end())
            cout << "Misspelled person's name\n";
        else break;
    }
    while (true) {
        author.name = getString("Enter author's name: ");
        authItr = find(catalog.begin(), catalog.end(),
        if (authItr == catalog.end())
            cout << "Misspelled author's name\n";
        else break;
    }
    while (true) {
        book.title = getString("Enter the title of the book: ");
        bookItr = find(authItr->books.begin(), authItr->books.end(),
        if (bookItr == authItr->books.end())
            cout << "Misspelled title\n";
        else break;
    }
    CheckedOutBook checkedOutBook(*authItr, *bookItr);
    (*bookItr)->person = 0;
    (*personItr)->books.remove(checkedOutBook);
}

int menu() {
    int option;
    cout << "\nEnter one of the following options:\n"
         << "1. Include a book in the catalog. Check out a book\n"
         << "3. Return a book\n4. Status\n5. Exit\n"
         << "Your option? ";
    cin >> option;
    cin.get();          // discard '\n'
    return option;
}
```

Program 3.31 (continued)

```
void main()
    while (true)
        switch (menu()) {
            case 1: findUnderstaffed(); break;
            case 2: checkOvertime(); break;
            case 3: examineBook(); break;
            case 4: status(); break;
            case 5: exit();
            default: cout << "Invalid option. Try again." ;
        }
```

Figure 3.32 A circular doubly linked list.



6. Delete from list L_1 nodes whose positions are to be found in an ordered list L_2. For instance, if $L_1 = (A\ B\ C\ D\ E)$ and $L_2 = (2\ 4)$, then the second and the fourth nodes are to be deleted from L_1; the eighth node does not exist, and after deletion, $L_1 = (A\ C\ E)$.

7. Delete from list L_1 nodes corresponding positions mentioned in ordered lists L_2 and L_3. For instance, if $L_1 = (A\ B\ C\ D\ E\ F)$, $L_2 = (2\ 4\ 8)$, and $L_3 = (3\ 3)$, then after deletion, $L_1 = (A\ C\ F)$.

8. Delete from an ordered list L_1 nodes corresponding positions indicated in list L_2 (duplicates removed). If $L_1 = (1\ 3\ 5\ 7\ 9)$, then after deletion, $L_1 = (1\ 5)$.

9. A linked list can also be used to implement a deque (pronounced "deck"), a double-ended queue. Implement a deque using linked lists.

10. Write a menu-driven program to check whether two simply linked lists are the same or not.

11. Write a method for a class to print a singly linked list in reverse order. Use recursion.

12. Insert a new node into a singly linked list (as defined in this section) after the node pointed to by p in this list, provided p is not the last node. Do not use a loop or any other operation.

13. Attach a singly linked list to the end of another singly linked list.

14. Put numbers in a singly linked list in ascending order. Use this operation to find the median in the list of numbers.

15. How can a singly linked list be implemented so that it supports constant-time insertion at both the front and the rear?

16. Insert a node in the middle of a doubly linked list.

17. Write code for class `IntDoublyLinkedList` for a circular singly linked list that includes equivalents of the member functions listed in Figure 3.2.

18. Write code for class `IntCircularDoublyLinkedList` for a circular doubly linked list that includes equivalents of the member functions listed in Figure 3.2.

19. How likely is the worst case for searching a singly linked list to occur?

20. Consider the member functions `front`, `transpose`, `insert`, and `remove` in `SinglyLinkedList`.

    (a) How does each affect a list maintained by these methods to be ordered?

    (b) In what case does each of these methods require the most search of the list for such search, assuming that only elements in the list are removed or found?

21. In the discussion of self-organizing lists, only the number of comparisons was considered as the measure of efficiency. This measure can, however, be greatly affected by a particular implementation of the list. Discuss how the efficiency of the move-to-front heuristic, counting only comparisons, is affected in the case when the list is implemented as

    (a) an array

    (b) a singly linked list

    (c) a doubly linked list

22. For doubly linked lists, it is easy to write methods that traverse the list in both directions. Write methods for `SinglyLinkedList` and `DoublyLinkedList` that traverse the list in reverse order. Write a method for `SelfOrganizingSinglyLinkedList` that searches a list of items and moves the item found (if any) to the front of the list. Write a method for `SelfOrganizingDoublyLinkedList` that searches a list of items and moves the item found (if any) to the front of the list.

    The swapping technique requires only one pass through the list. The target node is found, then swapped with its predecessor, then swapped with its predecessor, and so on, until it reaches the front of the list. Assuming that only comparisons of the data, what is the worst case for a move-to-front doubly linked list search? Is it different from the worst case for a move-to-front singly linked list search?

23. What is the maximum number of comparisons for a successful search for the list in Figure 3.19?

24. Adapt the binary search to linked lists. How efficient can this search be?

3.12 Programming Assignments

1. Fary fractions of level one are defined as sequence $[\frac{0}{1}, \frac{1}{1}]$. This sequence is increased to form a sequence $[\frac{0}{1}, \frac{1}{2}, \frac{1}{1}]$ sequence $[\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}]$ at level three, sequence $[\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}]$ at level four, and so forth. At each level, a new fraction $\frac{p}{q}$ is inserted between every neighboring pair $\frac{a}{b}$ and $\frac{c}{d}$ for which $bc - ad = 1$. Write a program which for a number n entered by the user creates—by constructing and storing in a linked list of fractions at level n and then displays them.

2. Write a simple airline ticket reservation program. The program should display a menu with the following options: reserve a ticket, cancel a reservation, check whether a ticket is reserved for a particular person, and display the passengers. The information is maintained in an appropriate linked list of names. In a simple version of the program, assume that tickets are reserved for only one flight. In a fuller version, place no limit on the number of flights. Create a linked list of flights with each node linked to a pointer to a linked list of passengers.

3. Read Section 17.1 about the heap sort algorithm. Implement the algorithm and verify with linked lists and compare their efficiency.

4. Write a program to simulate managing files on disk. Define the disk as a one-dimensional array `disk` of size `numOfSectors * sizeOfSector`, where `sizeOfSector` indicates the number of characters stored in one sector. For the sake of debugging, make it a very small number. A pool of available sectors is kept in a linked list consisting of `struct` field structures: two fields to indicate ranges of available sectors and one `next` field. Files are kept in a linked list of `struct` field structures: file name, the number of characters in the file, a pointer to a linked list of sectors where the contents of the file can be found, and the `next` field.

   a. In the first part, implement functions to save and retrieve files. Saving files requires finding a sufficient number of sectors from pool of free disk. Files do not have to be contiguous, so a linked list assigned to the file may contain several nodes. If no sufficient number of available sectors is found, assigned to the file. Deletion of a file only requires removing its entry corresponding with the file from the list of files and moving the sectors assigned to the file back to the pool of free sectors. Appropriate changes to the pool of free sectors are made in both operations.

   b. File fragmentation slows down file retrieval. In the ideal situation, one cluster of sectors is assigned to one file. However, after many operations with files it may not be possible. Extend the program to include a function `defragDisk()` to transfer files to contiguous sectors, that is, to create a situation illustrated in Figure 3.20. Fragmented files `file1` and `file2` occupy more than one cluster of sectors after `defragDisk()` is finished. However, particular care should be taken not to overwrite sectors occupied by other files. For example, `file1` requires eight sectors. Five sectors are free at the beginning of `disk`, but sectors 3 and 4 are occupied by `file2`. Therefore, a file from occupying such sectors has to be moved first by searching `disk`. The contents of these sectors must be

transferred to unoccupied positions, which requires updating the sectors belonging to the linked list. Only then can the released sectors be utilized. One way of accomplishing this is by copying from the end from which one file's copied chunks of sectors of another file into an area of the disk large enough to accommodate these chunks. In the example in Figure 4.33, contents of file1 are first copied to sectors 0 through 4, and then copying is to open and suspended because sector 5 is occupied. Thus, contents of sectors 7 and 8 are moved to sectors 5 and 6, and the copying of file1 is resumed.

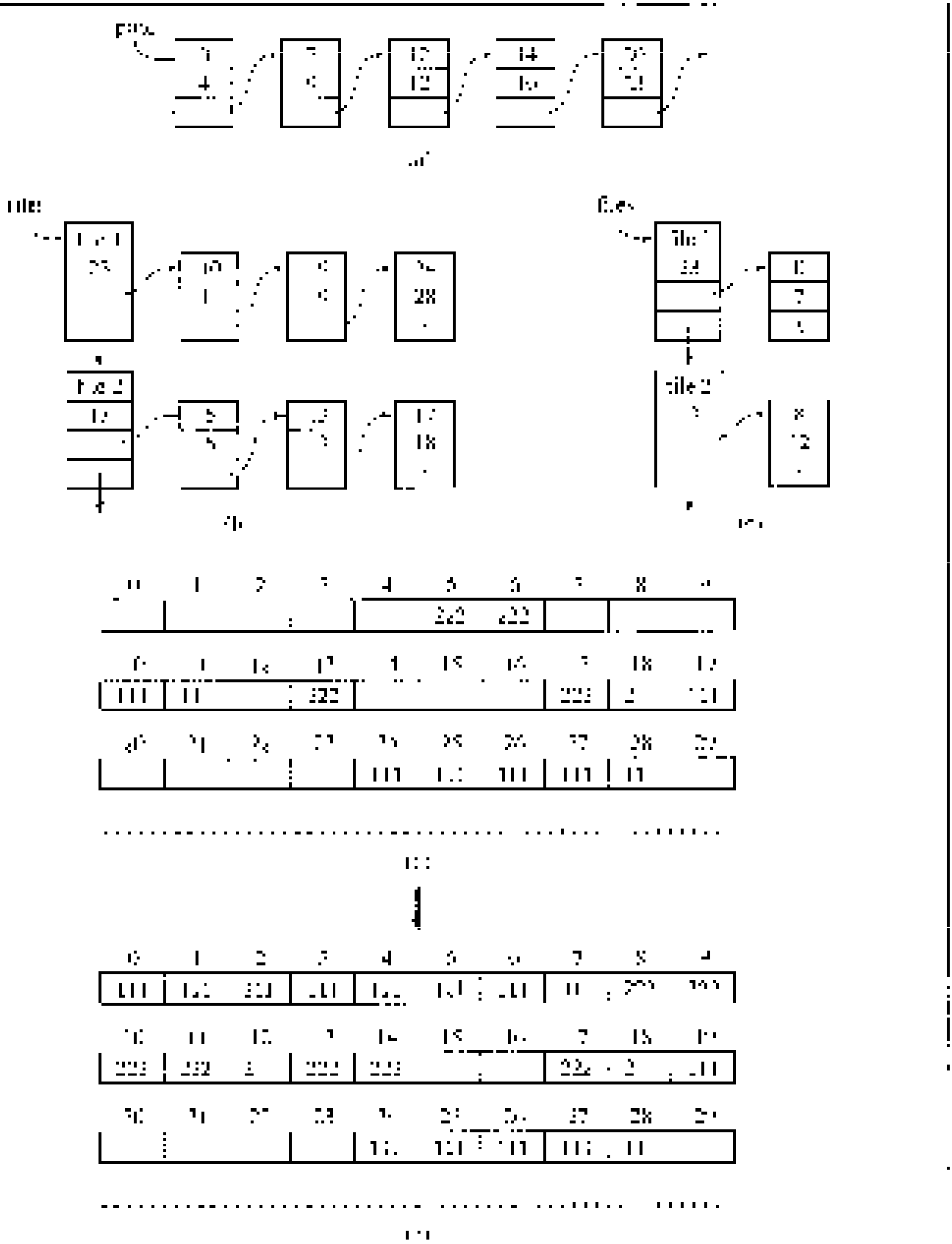
5. Write a simple line editor. Records of a text file are linked list nodes. Start the program with a command line file, after which a prompt appears along with the line number. Each line is entered with a number as the following is, if so, insert the text to be followed before that line. If a text is not followed by a number then insert the new text as a new line. Lines are entered with two numbers and a zero, or the number following it, depending on the program, lines, or the current line. Delete lines with the command, d, which is similar to the d command. If a is entered, then append the text to the existing lines. Here is a sample session of entering the text in a file from scratch:

```
edit textfile
1> This is line 1
2>
3> And another line
4> i 3
3> This is inserted line
4> Now insert line
5> i 1
1> This is line 1
2>
3> This is appended line
4> Now insert line
5> And another line      // This is now line 5, not 4;
6> d 2                   // line 5, since 1 was deleted from line 6;
4> 1.                    // line 4, since one line was deleted;
 > This is line 1
1> This is appended line // this and the following lines
2> Now insert line       // now have new numbers;
3> And another line
4> x
```

6. Enhance the case study program in the chapter to use a hash table in addition to the list. Insert each word into the hash table as well as the linked list, using this information to check one of the program's assertions by adding a new operation, such as `hash`, allowing the same book to be checked out at the same time as many times as the number of copies in the library.

7. Test the efficiency of skip lists. In addition to the functions given in the chapter, implement `skipListDelete` and then compare the number of node accesses in

FIGURE 3.33 Linked lists used to allocate disk sectors for files: (a) a pool of available sectors, two files, (b) before and (c) after putting them in contiguous sectors; the situation in sectors of the disk (d) before and (e) after this operation.



searching, deleting, and inserting for large numbers of elements. Compare this efficiency with the efficiency of linked lists and ordered linked lists. Use your program on a randomly generated order of operations to be executed on the elements. These elements should be processed in random order. Time your program for various sizes.

8. Write a random access file program to maintain a short mailing list. Records have been initialized for the following variables: last name, first name, address, city, state, and zip code. Create a linked list of these variables and, for each name, create a linked list of associated variables. In both lists, store pointers to the next possible record as well as the previous record. Also, store pointers to the next possible record as well as the previous record in the sorted order of the last name. Allow the user to input a last name and display the first name, address, city, state, and zip code. Allow the user to input a first name and display the last name, address, city, state, and zip code. Allow the user to input a city and display the last name, first name, address, state, and zip code. Allow the user to input a state and display the last name, first name, address, city, and zip code. Allow the user to input a zip code and display the last name, first name, address, city, and state. Allow the user to input a last name and first name and display the address, city, state, and zip code. Allow the user to input a last name and address and display the first name, city, state, and zip code. Allow the user to input a last name and city and display the first name, address, state, and zip code. Allow the user to input a last name and state and display the first name, address, city, and zip code. Allow the user to input a last name and zip code and display the first name, address, city, and state. Allow the user to input a first name and address and display the last name, city, state, and zip code. Allow the user to input a first name and city and display the last name, address, state, and zip code. Allow the user to input a first name and state and display the last name, address, city, and zip code. Allow the user to input a first name and zip code and display the last name, address, city, and state. Allow the user to input a last name, first name, and address and display the city, state, and zip code. Allow the user to input a last name, first name, and city and display the address, state, and zip code. Allow the user to input a last name, first name, and state and display the address, city, and zip code. Allow the user to input a last name, first name, and zip code and display the address, city, and state. Allow the user to input a last name, address, and city and display the first name, state, and zip code. Allow the user to input a last name, address, and state and display the first name, city, and zip code. Allow the user to input a last name, address, and zip code and display the first name, city, and state. Allow the user to input a last name, city, and state and display the first name, address, and zip code. Allow the user to input a last name, city, and zip code and display the first name, address, and state. Allow the user to input a last name, state, and zip code and display the first name, address, and city. Allow the user to input a first name, address, and city and display the last name, state, and zip code. Allow the user to input a first name, address, and state and display the last name, city, and zip code. Allow the user to input a first name, address, and zip code and display the last name, city, and state. Allow the user to input a first name, city, and state and display the last name, address, and zip code. Allow the user to input a first name, city, and zip code and display the last name, address, and state. Allow the user to input a first name, state, and zip code and display the last name, address, and city. Allow the user to input a last name, first name, address, and city and display the state and zip code. Allow the user to input a last name, first name, address, and state and display the city and zip code. Allow the user to input a last name, first name, address, and zip code and display the city and state. Allow the user to input a last name, first name, city, and state and display the address and zip code. Allow the user to input a last name, first name, city, and zip code and display the address and state. Allow the user to input a last name, first name, state, and zip code and display the address and city. Allow the user to input a last name, address, city, and state and display the first name and zip code. Allow the user to input a last name, address, city, and zip code and display the first name and state. Allow the user to input a last name, address, state, and zip code and display the first name and city. Allow the user to input a last name, city, state, and zip code and display the first name and address. Allow the user to input a first name, address, city, and state and display the last name and zip code. Allow the user to input a first name, address, city, and zip code and display the last name and state. Allow the user to input a first name, address, state, and zip code and display the last name and city. Allow the user to input a first name, city, state, and zip code and display the last name and address. Allow the user to input a last name, first name, address, city, and state and display the zip code. Allow the user to input a last name, first name, address, city, and zip code and display the state. Allow the user to input a last name, first name, address, state, and zip code and display the city. Allow the user to input a last name, first name, city, state, and zip code and display the address. Allow the user to input a last name, address, city, state, and zip code and display the first name. Allow the user to input a first name, address, city, state, and zip code and display the last name. Allow the user to input a last name, first name, address, city, state, and zip code and display the last name, first name, address, city, state, and zip code

Stacks and Queues

As the first chapter explained, abstract data types allow us to delay the specific implementation of a data type until it is well understood what operations are required to operate on the data. In fact, these operations dictate which implementation of the data type is most efficient for a particular situation. This situation will be illustrated by two data types, stacks and queues, which are described by a list of operations. Only after the list of the required operations is determined do we present some possible implementations and compare them.

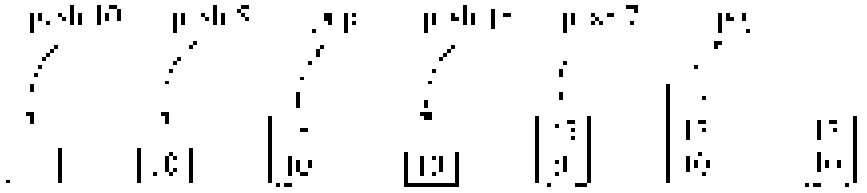
4.1 Stacks

A stack is a linear data structure in which items can be accessed only at one end of it, called the top. Stacks are like trays in a cafeteria. New trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called a *last-in first-out* (LIFO) structure.

A tray can be taken only if there are trays on the stack, and a tray can be added to the stack only if there is enough room, that is, if the stack is not too high. Therefore, a stack is defined in terms of operations which change its status and operations which check its status. The operations are as follows:

- `clear()`—Clear the stack.
- `isEmpty()`—Check to see if the stack is empty.
- `push(e)`—Put the element `e` on the top of the stack.
- `pop()`—Take the topmost element from the stack.
- `topEl()`—Return the topmost element in the stack without removing it.

FIGURE 4.1 A series of push and pop operations on a stack.



A series of push and pop operations is shown in Figure 4.1. After pushing an integer onto an empty stack, the stack contains only that number. After pushing 5 onto the stack, the number is placed on top of 10 so that, when the popping operation is executed, it is removed from the stack because it arrived after 10, and is therefore on top of the stack. After pushing 15 and then 7, the rightmost element, 7, and this number are removed when executing the popping operation, after which the stack contains 15 on top of 10 and, therefore, 15 above it.

Generally, the stack is very useful in situations when data have to be stored and then retrieved in reverse order. One application of the stack is in matching delimiters in a program. This is an important example because delimiter matching is part of any compiler. No program is considered correct if the delimiters are unmatched.

In C++ programs we have the following delimiters: parentheses '(' and ')', square brackets '[' and ']', curly brackets '{' and '}', and comment delimiters '/*' and '*/'. Here are examples of C++ statements that use delimiters properly:

```
a = b + (c - d * (e + f));
v[20] = b[a[9] + 5] + 10 * k + 1;
while (a[i] != a[j]) { i = i + 1; /* Initialize a[i] and a[j] */
```

These examples are statements in which the matching occurs.

```
a = b = (c - d * (e - f));
v[20] = b[a[9] + 5] + 10 * k + 1;
while (a[i] != a[j]) { i = i + 1; /* Initialize a[i] and a[j] */
```

A particular delimiter can be separated from its match by other delimiters. That is, delimiters can be nested. However, a pair of delimiters is not closed properly after all the delimiters following it are properly matched. For example, in the construction of the loop

```
while (a[(i + 1)] != b);
```

the first opening parenthesis must be matched with the last closing parenthesis. For this reason only after the second opening parenthesis is matched with the second closing parenthesis is the first opening parenthesis matched with the first closing parenthesis.

The delimiter matching algorithm reads a character from the C++ program and stores it on a stack if it is an opening delimiter. If a closing delimiter is encountered, the delimiter popped off the stack is compared with it. If they match, processing continues. If not, processing discontinues by signaling an error. The processing of the C++ program ends successfully after the end of the program is reached and the stack is empty. Here is the algorithm:

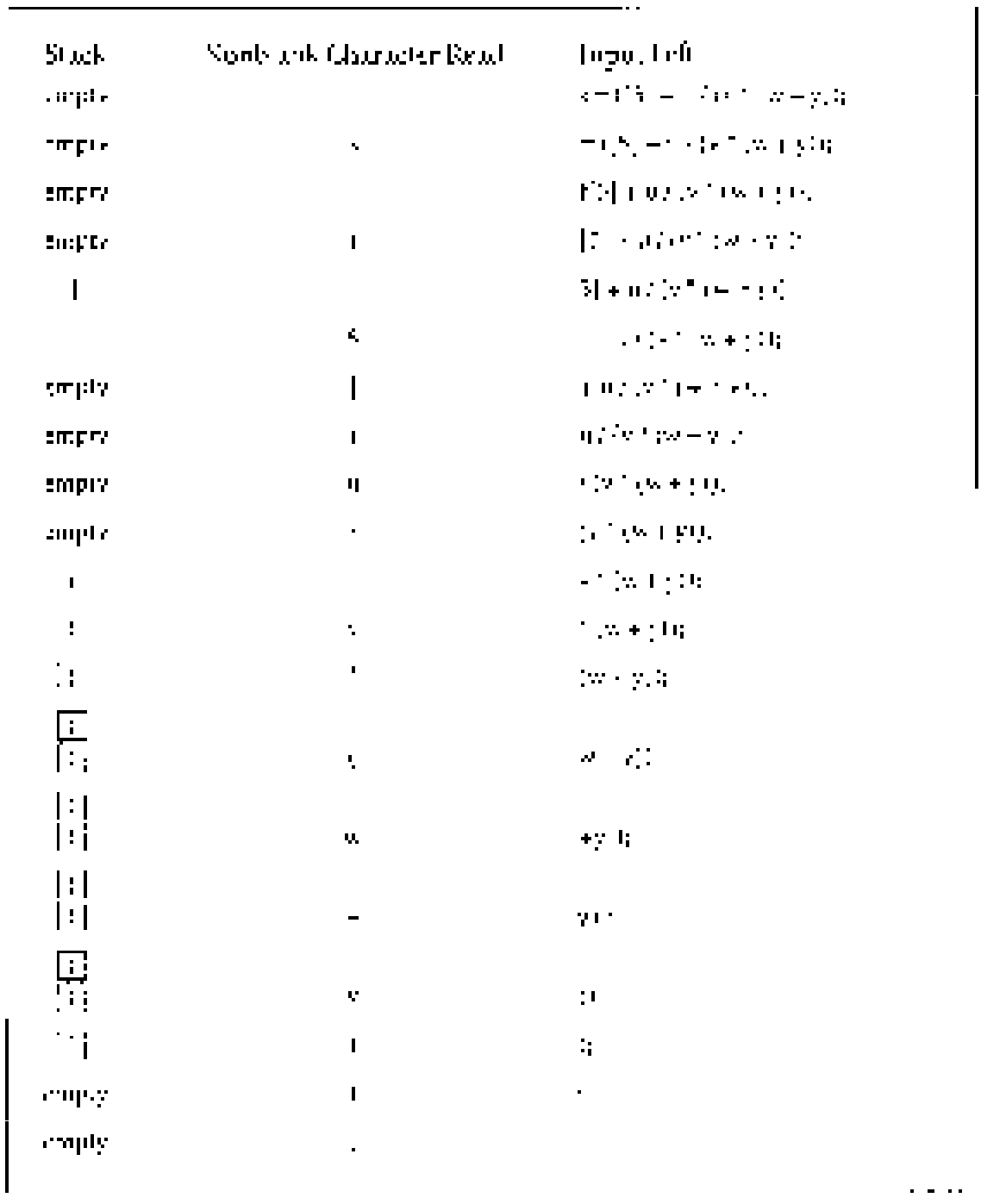
```
declare a character stack
while not end of file
    read a character
    if the character is a '('
        push it on the stack
    else if it is a ')'
        if the stack is not empty
            pop a character off the stack
        else it is unmatched
            output error message
    else if it is a '[' or '{'
        push it on the stack
    else if it is a ']' or '}'
        if the stack is not empty and the character popped off matches
            the current
        then do nothing
        else it is unmatched
            output error message
    else it is another character
        ignore it
if the stack is not empty
    output error message for each character on the stack
else
    output success message
```

Figure 4.2 shows the processing that occurs when applying this algorithm to the statement

```
a=b[(c+d)*(e+f)];
```

The first column in Figure 4.2 shows the contents of the stack at the end of the iteration whose character is taken from the corresponding line. The first line shows the initial contents of the stack, which is initialized to the first character of the input, a letter, and in the first iteration of the loop, the character is simply ignored. This is the case as shown in the second row in Figure 4.2. Then the next character, equal sign, is read in and ignored, and so is the letter b after reading the left bracket is pushed onto the stack so that the stack now has two elements, the left bracket. Reading a '(' does not change the stack, but after the '(' bracket becomes the second item, the topmost character is popped off the stack and compared with the

FIGURE 4.2 Processing the statement `x = (3 * (y + z));` with the algorithm for infix-to-postfix conversion.



Because the popped element (left bracket) matches the right bracket, the processing of input continues. After reading and discarding the letter u, a slash is read and the algorithm checks whether it is part of the comment delimiter by reading the next character. If the next character is not an asterisk, the slash is not a beginning of a comment, so a left parenthesis is pushed onto the stack and processing continues, as shown in Figure 4.2. After reading the last character in the expression, the loop is exited and the stack is checked. Because it contains the matched left parenthesis, success is announced.

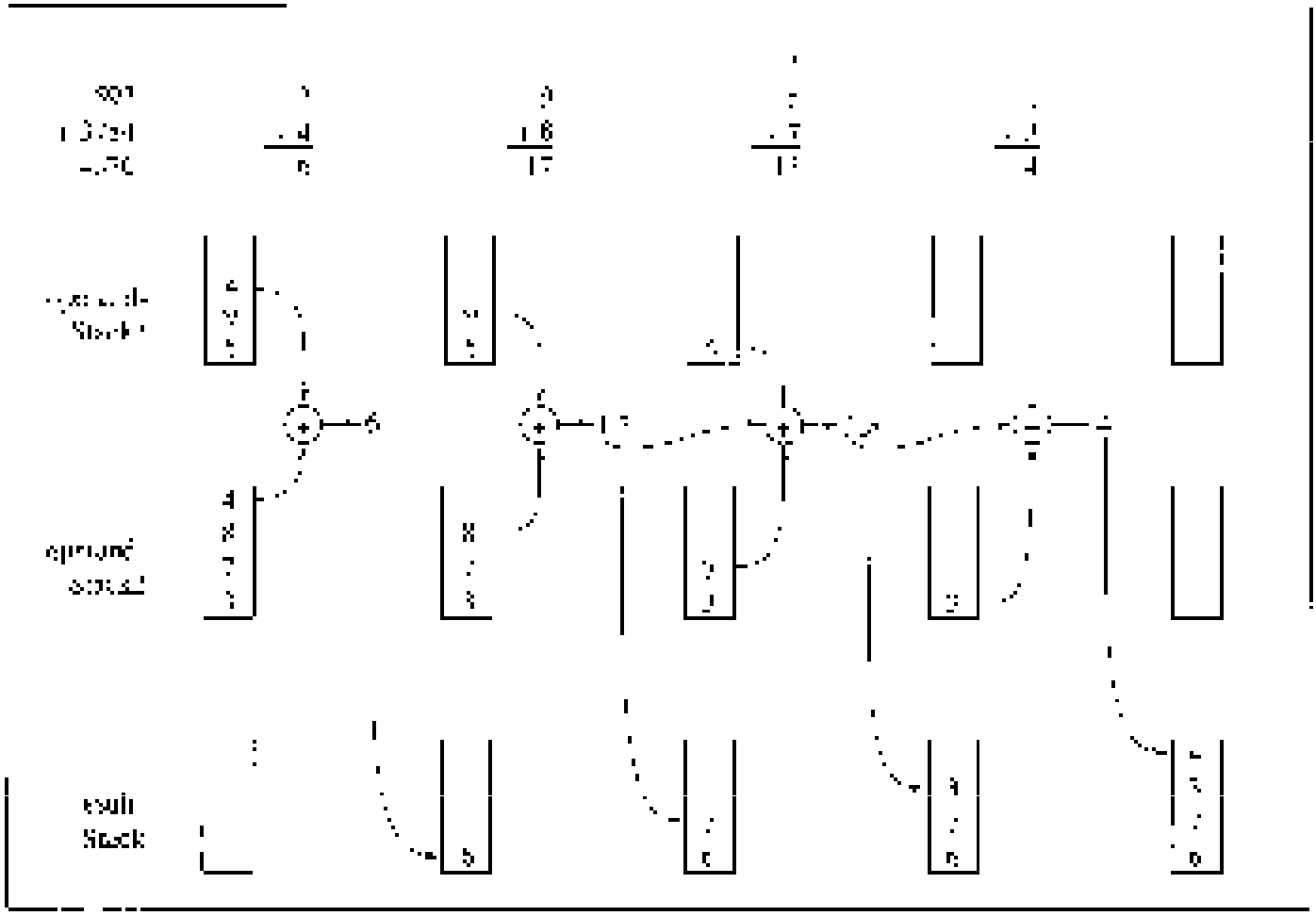
As an example of stack application, consider adding two large numbers. The largest unsigned integer is limited, so we are not able to add 18,234,364,483,979,273,947,465,658,494,373 and 25,469. Such integer variables cannot hold such large values, let alone their sum. The problem can be solved if we read the numbers as strings of characters, store the numbers corresponding to these characters on two stacks, and then perform addition by popping numbers from the stacks. The pseudocode for this algorithm is as follows:

```
adding two gigantic integers:
    read the two strings, first number into stack1, second number into stack2;
    store the length of the first number in length1;
    store the length of the second number in length2;
    carry = 0;
    while stack1 or stack2 is not empty
        pop a number from stack1 if stack1 is not empty and add them to result;
        pop a number from stack2 if stack2 is not empty
        add them to result;
        store the ones part of result in carry;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```

Figure 4.3 shows an example of application of this algorithm. It is used to add numbers 987 and 1781 as follows:

1. Numbers corresponding to the characters in the first number are pushed onto stack1, and numbers corresponding to the characters of the second number onto stack2. Note that we pop digits off the stacks.
2. Numbers 1 and 4 are popped from the stacks and the result, 5, is pushed onto result stack.
3. Numbers 8 and 7 are popped from the stacks and the ones part of their sum, 5, is stored in result. The tens part of the result, number 1, is stored in a variable carry for addition in the next step.
4. Numbers 9 and 7 are popped from the stacks, added to the carry, and the ones part of the result, 7, is pushed onto result stack. The tens part, 1, becomes a value of the variable carry.
5. One stack is empty so a number is popped from the non-empty stack, added to carry, and the result is stored on result stack.
6. Both stacks are empty, so the numbers from result stack are popped and printed as the final result.

Figure 4-3 An example of adding numbers 394 and 5836 using stacks



Consider now the implementation of our abstract stack data structure. We could provide an appropriate iterator through the existing vector and list classes, but it is just as easy to implement the stack operations operating on the stack.

A natural implementation of a stack is a doubly linked list, that is, a vector. Figure 4-4 contains the beginning of a stack class definition that can be used to store any kind of object. Also, a linked list can be used to implement a stack (Figure 4-5).

Figure 4-6 shows the same sequence of push and pop operations as Figure 4-4, with the difference that now the stack is implemented as a vector (Figure 4-6b) and a linked list (Figure 4-6c).

The linked list implementation matches the abstract stack more closely in that it includes only those elements that are on the stack because the number of nodes in the list is the same as the number of stack elements. In the vector implementation, the capacity of the stack can often surpass its size.

The vector implementation, like the linked list implementation, does not force the programmer to make a commitment at the beginning of the program concerning the size of the stack. If the size can be reasonably assessed in advance, then the predicted size can be used as a parameter for the stack's constructor to reserve in advance a

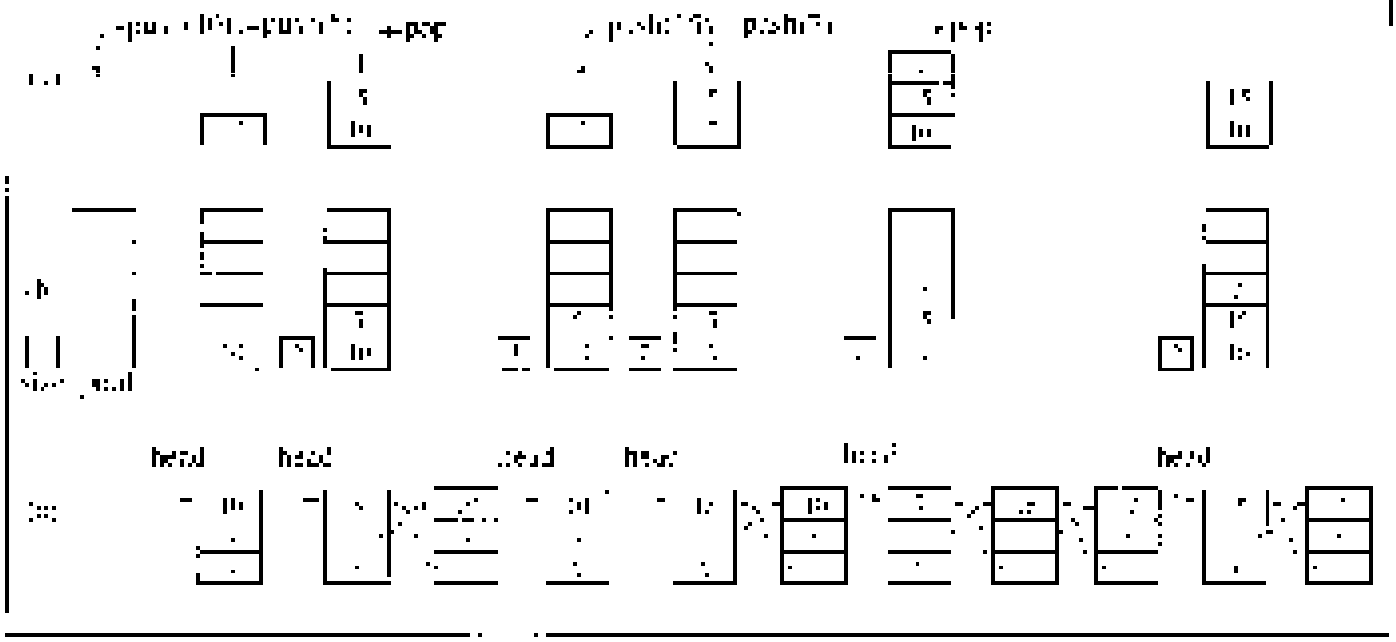
Figure 4.4 A vector implementation of a stack.

```
//*************************** vecstack.h ***************************
// A stack class for vector-like data structures
//******************************************************************

#ifndef STACK
#define STACK

#include <vector>

using namespace std;

template<class T, int capacity = 30>
class stack {
public:
    stack() {
        pos = 0; v.reserve(capacity);
    }
    void clear() {
        pos = 0; v.clear();
    }
    bool isEmpty() const {
        return pos == 0;
    }
    T& top() {
        return v[pos-1];
    }
    T pop() {
        T el = v[--pos];
        v.pop_back();
        return el;
    }
    void push(const T& el) {
        v.push_back(el);
        pos++;
    }
private:
    vector<T> v;
    int pos;
};

#endif
```

Program 4.5 Implementing a stack as a linked list

```
//*************************  genStackBack.h  ****************************
//     generic stack defined as a doubly linked list

#ifndef GS_STACK
#define GS_STACK

#include <list>

using namespace std;

template<class T>
class GSBack {
public:
    GSBack() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();
    }
    T& top() {
        return lst.back();
    }
    T pop() {
        T el = lst.back();
        lst.pop_back();
        return el;
    }
    void push(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

FIGURE 4.6 A series of operations executed on an abstract stack (a) and the stack implementations with (b) an array and (c) a linked list.



exceed the specified capacity. In this way, an exception is raised to keep the stack elements in a new larger stack before we push a new item. This is the same idea we used to expand capacity.

It is easy to see that in the array and linked list implementations, popping and pushing are executed in constant time $O(1)$. However, in the worst case, pushing an element onto a full stack requires allocating a new memory and copies the items from the existing vector to a new vector. The worst case in the worst case pushing takes $O(n)$ time to finish.

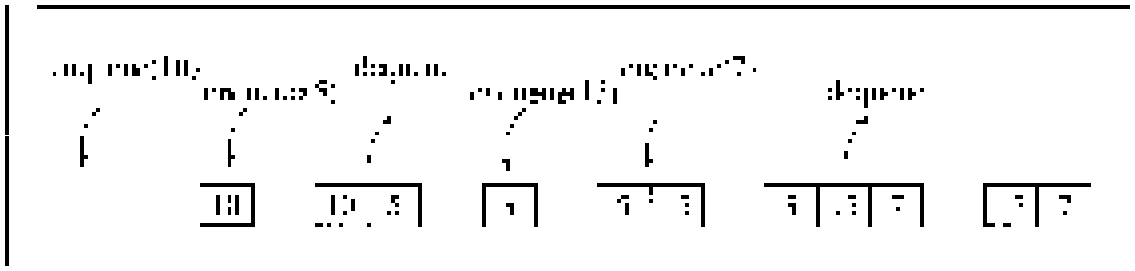
4.2 Queues

A queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. The order that the elements leave a queue is the same order in which they are added. A queue is an FIFO (first-in, first-out) structure.

Queue operations are similar to stack operations. The following operations are needed to properly manage a queue:

- `clear()`—Clear the queue.
- `isEmpty()`—Return true if the queue is empty.

FIGURE 4.7 A series of operations on a queue.



- enqueue(x)—Insert x at the rear of the queue.
- dequeue()—Remove the first element from the queue.
- front()—Return the first element in the queue without removing it.

A series of enqueue and dequeue operations is shown in Figure 4.7. This time—unlike for stacks—we don't have to be concerned about the beginning of the queue array. Instead, the elements are enqueued on one end and dequeued from the other. For example, after enqueuing 10 and then 5, the dequeue operation removes 10 from the queue and returns it.

For an application of queues, consider the following poem written by Lewis Carroll:

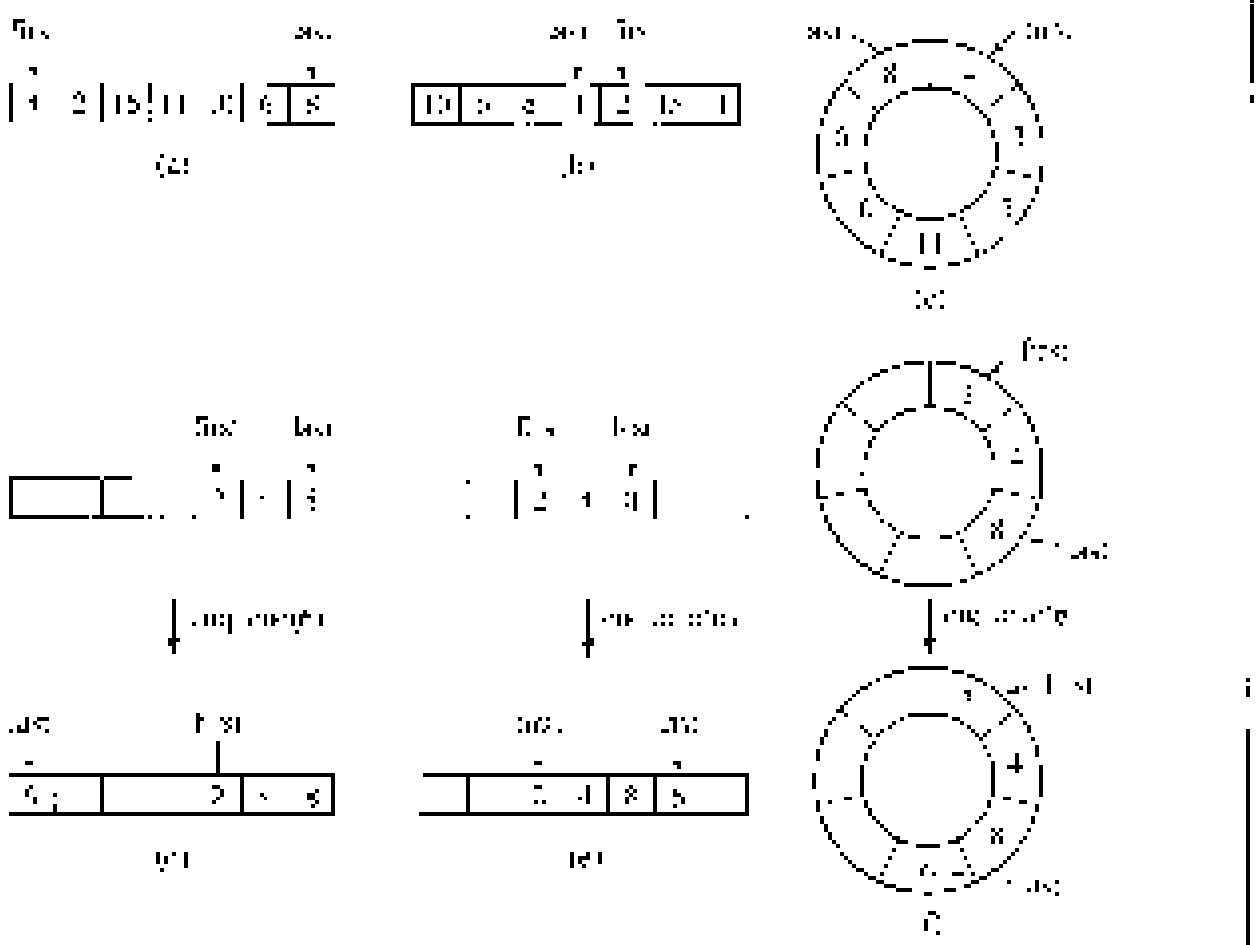
Speed bonny boat, like a bird on the wing,
Onward o'er the foam!
Though the crew's not required to perform anything,
Still it's cheering to know they're at home.

The poem is defined so that by removing words from the end, we get the next word of the poem. In order to produce the poem, we enqueue the first word. Then, each time we dequeue a word, we enqueue the next word after it. This type of processing is common in simulations, where a queue is used to hold customers waiting for service. Each customer is an object that stores the time the customer arrived in the queue. When the customer reaches the front of the queue, the difference between the current time and the time the customer arrived is the time the customer spent waiting in the queue. The following pseudocode illustrates this processing:

```
averageTimeWaiting()
    while there are more customers
        read the next customer
        enqueue the customer's arrival time in the queue
        if the queue is not empty
            dequeue and process a customer
```

There are many more applications to follow, but first consider the problem of implementing a queue.

FIGURE 4.8 (a) The two possible configurations in an array implementation of a queue when the queue is full. (b) Elements can be viewed as in a circular array. (c) Enqueuing a number in the queue in (a) or (b). (d)–(f) Three successive dequeue operations on the queue in (c), with the last element (first the one in the rear and then the one in the middle).



One possible queue implementation is to use a circular array. Although it cannot be the best choice, it is instructive. Elements are added at the end of the queue, and they may be removed from the beginning, thereby releasing array cells. These cells should not be wasted. Therefore, they are utilized to enqueue new items. Hence, when the end of the queue may occur at the beginning of the array. This situation is best visualized as a circular array as Figure 4.8(a) illustrates. The queue is full if the last element immediately precedes the first element in clockwise direction. The last element is, however, because a circular array is implemented with a "normal" array, the queue is full if either the last element is in the last cell and the first element is in the first cell (Figure 4.8(a)) or if the last element is right after the first (Figure 4.8(b)). Similarly, enqueuing and dequeuing have to consider the possibility of wrapping around the array when adding or removing elements. For example, enqueuing can be viewed as operating on a circular array (Figure 4.8(c)), but in reality it is operating on a one-dimensional array. Therefore, if the last element is in

be the next node to be removed, and available[rear] is the beginning of the array where a new node will be placed (Figure 4.5c). The next pointer is always the predecessor of the node pointed to by the previous pointer in the list sequence (Figure 4.6b). These two simultaneous traversals will be described in pseudocode viewed as circular (Figure 4.6d).

Figure 4.5 continued Illustration of insertion and deletion in a circular queue

A pseudocode algorithm for insertion is a combination of the previous two algorithms (Figure 4.7).

In the algorithm for deletion, the statements assigning and dequeuing will be executed only if queue > 0 (to provide a doubly linked list is used in the first implementation). Otherwise, a singly linked list can be used. Deletion, requiring only O(1) operations, proceeds by traversing the list from top to bottom. A discussion of the implementation is in Section 4.1.2.

Figure 4.8 shows how to use sequence and sequence operations (Figures 3.2 and 3.3) to implement the queue implemented as an array (Figure 4.1) or as a linked list (Figure 4.4). The linked list keeps only the numbers that are logically in the queue, as indicated by Figure 4.1b, not the array. The array includes all the numbers up to the high point, after which new numbers are just added starting from the beginning of the array.

Queues are frequently used in situations in the event that a developer and mathematically sophisticated theory of queues exists called queuing theory in which queues are characterized and analyzed and models are built which use queues. In queuing processes, there are a number of customers coming to servers to receive service. The customers queue up if the servers are busy. Therefore, customers have to wait in queues before they are served, and they spend some amount of time while they are being served. By customers, we mean not only people but also objects. For example, packets in a network may be in the process of being transmitted in a machine, trucks waiting for service at a weighing station on an interstate, or barges waiting for a sluice to be open so they can pass through a channel. All wait in queues. The most familiar examples are lines at stores, post offices, or banks. The type of problems posed in simulations are: How many servers are needed to avoid long queues? How large must the waiting space be to be sure the entire queue fits in it? Is it cheaper to increase the speed of or to open more servers?

As an example, consider Bank One which, over a period of a number of months, records the number of customers coming to the bank and the amount of time needed to serve them. The table in Figure 4.12a shows the number of customers who arrived during 15-minute intervals throughout the day. For 15% of such intervals, no customer arrived; for 20%, only one arrived, etc. Currently, six clerks are employed, no lines are ever observed, and the bank management wants to know whether six clerks are too many. Would five suffice? Four? Maybe even three? Can lines be expected to be tolerable? To answer these questions, a simulation program is written which applies the probability data and checks different scenarios.

The number of customers depends on the value of a randomly generated number between 1 and 100. The table in Figure 4.12a identifies five ranges of numbers from 1 to 100, based on the percentages of 15-minute intervals that had 0, 1, 2, 3, or 4 customers. If the random number is 21, then the number of customers is 1. The random

Program 4.9 Array implementation of queue

```
//*************************  queueArrayQueue.h  ************************
//              queue = queue implemented as an array

- Linked array queue
-Define Array Queue

template<class T, int size = 100>
class ArrayQueue {
public:
    ArrayQueue() {
        first = last = -1;
    }
    void enqueue(T);
    T dequeue();
    bool  empty() {
        return first == -1 || last == size-1  || first    last + 1;
    }
    bool  operator!() {
        return first == -1;
    }
private:
    int first, last;
    T storage[size];
};

template<class T, int size>
void ArrayQueue<T,size>::enqueue(T el) {
    if (!isFull()) {
        if (last == size-1  ||  last == -1) {
            storage[0] = el;
            last = 0;
            if (first == -1)
                first = 0;
        }
        else storage[++last] = el;
    }
    else cout << "Full queue\n";
}

template<class T, int size>
T ArrayQueue<T,size>::dequeue() {
```

Listing 4.9 (continued)

```
    T tmp;
    tmp = storage[first];
    if (first == last)
        last = first = -1;
    else if (first == size-1)
        first = 0;
    else first++;
    return tmp;
}

#endif
```

Listing 4.10 Doubly linked list implementation of queue.

```
//*************************** genQueue.h ****************************
//  generic queue implemented with doubly linked list

#ifndef QUEUE
#define QUEUE

#include <iostream>

using namespace std;

template <class T>
class Queue {
public:
    Queue() {
    }
    void clear() {
        while(!isEmpty())
            dequeue();
    }
    bool isEmpty() const {
        return (front == 0);
    }
    T front() {
```

FIGURE 4.10 (continued)

```
        return list.front();
    }
    T dequeue() {
        T el = list.front();
        list.pop_front();
        return el;
    }
    void enqueue(const T& el) {
        list.push_back(el);
    }
protected:
    List<T> list;
};

#endif
```

FIGURE 4.11 A sequence of operations on a queue object: (a) abstract queue of characters; (b) queue implemented with an array; (c) queue with a linked list.

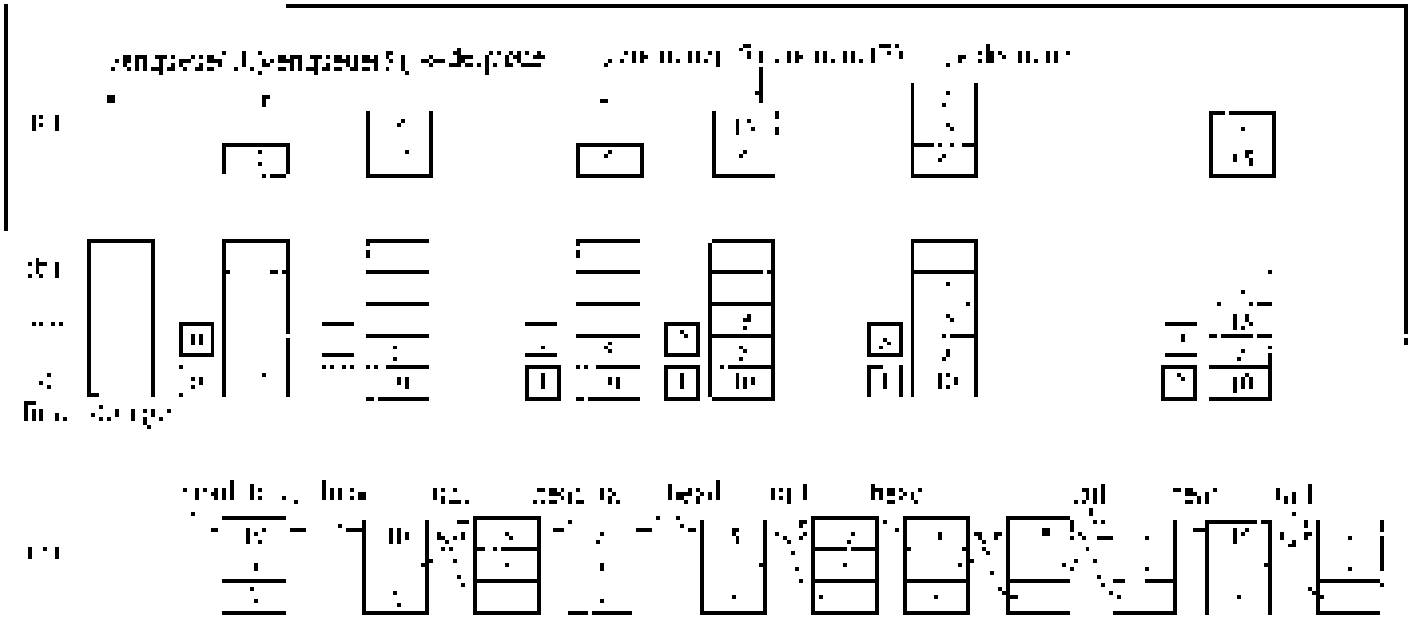


EXHIBIT 4.12 Bank One sample (a) data for number of arrived customers per one-minute interval and (b) transaction time in seconds per customer

Number of customers per minute	Percentage of one-minute intervals	Range
0	25	1–25
1	20	26–35
2	15	36–50
3	10	51–75
4	10	76–85
5	10	86–95
6+		

The amount of time needed for service in seconds	Percentage of customers	Range
20	10	
25	10	
30	20	
35	10	31–40
40	5	41–45
45	10	46–55
50	10	56–65
55	10	66–75
60	5	76–80
65	5	81–85
70	5	86–90
75	5	91–95
80	5	96–100

number is 30, then the number of customers is 4. This method simulates the rate of customers arriving at Bank One.

In addition, outputs of the modified Excel worksheet indicate that no customers arrived in 25 seconds or 25 seconds of transactions. This is represented in seconds as indicated in Figure 4.12b. The table in 4.12b includes ranges for random numbers to generate a length of transaction in seconds.

Figure 4.13 compares the projected simulation customer arrival and transaction time at Bank One. The program uses the values in the table to provide the percentages of 1-minute intervals depending on the number of the arrived customers. The empirical data is used to access the distribution of time needed for service. The amount of time is obtained by multiplying the index value given randomly by 10. For example, when the sample is divided by 10, a random value of 10 for the next customer required 1 minute 40 seconds for service. The program then calculates the length of the customer transaction in seconds.

For each customer represented by the empirical data, the number of arriving customers is randomly chosen, and the service time, the transaction time, is also

Fig. 4.13 Bank line example implementation code

```
#include <iostream.h>
#include <stdlib.h>
#include "queue.h"

int option(int percents[]) {
    register int i = 0, choice = rand()%100+1, pos;
    for (pos = percents[0]; pos < choice; pos += percents[i-1], i++);
    return i;
}

void main() {
    int arrivals[] = {15,20,25,10,30};
    int services[] = {3,3,3,10,5,10,10,3,15,25,10,15};
    int clocks[4] = {0,0,0,0}, numOfClerks = sizeof(clocks)/sizeof(int);
    int customers, i, numOfServices = 100, x;
    double maxWait = 0.0, waitSum = 0.0, theServiceTime = 0.0;
    Queue<int> theQ;
    randomize();
    for (i = 0; i <= numOfClerks; i++) {
        cout << "i = " << i;
        for (int j = 0; j < numOfClerks; j++)// clear each window accrued
            if (clocks[j] < 60)             // at most 60 seconds from line
                clocks[j] = 0;              // left to service the current
        while (clocks[i] >= 60)             // customer by clerk i;
        customers = arrivals[option(arrivals)];
        for (int j = 0; j < customers; j++) {// compute all new customers
            x = services[option(services)*10];  // for current service time
            theQ.enqueue(x);                // into queue;
            theSum += x;
        }
        // compute customers when clerks are available
        for (j = 0; j < numOfClerks && !theQ.isEmpty(); j++)
           if (clocks[j] < 60) {
                x = theQ.dequeue();         // assign next item and customer
                clocks[j] += x;             // to a clerk if available and
                theSum -= x;                // is still below 60 secs;
           }
        while (j < i);
        if (!theQ.isEmpty()) {
```

FIGURE 4.13 (continued)

```
        theNextCust();
        nextExit = theCustWithSmallestTime();
        if (nextExit < currExit)
            currExit = nextExit;
    }
    else theClerk[nextExit].busy = false;
    }
    cout << "\n\nFor " << numCustServed << " customers, there was a total "
         << theClerk[1].timeWaiting + theClerk[2].timeWaiting << " minutes\n"
         << "of waiting time with an average of " << (theClerk[1].timeWaiting
         << " minutes per customer.\n";
}
```

randomly determined. The function `option()` generates a random number, limits the range into which it falls, and then outputs the position, which is stored the number of customers or a zero if the number of seconds.

Executions of the program indicate that six and five clerks are too many. With four clerks, service is performed promptly 70% of the time there is a customer waiting customers. However, three clerks are always busy and there is always a long line of customers waiting. Bank management would certainly decide to employ four clerks.

4.3 PRIORITY QUEUES

In many situations, simple queues are inadequate, since first-in/first-out scheduling must be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue. On roads with toll booths, some vehicles may be put through immediately, even without paying (police cars, ambulances, fire engines, etc.). In a sequence of processes, process P_2 may need to be executed before process P_1 for the proper functioning of the system, even though P_1 was put on the queue of waiting processes before P_2. In situations like these, a modified queue, or *priority queue*, is needed. In priority queues, elements are dequeued according to their priority and their current position.

The problem with priority queues is finding an efficient implementation which allows relatively fast enqueuing and dequeuing. Since elements may arrive randomly at the queue, there is no guarantee that the first element will be the most relevant to be dequeued and that the elements put at the end will be the last candidates for executing. The situation is complicated because a wide spectrum of reasonable priority criteria can

be used in different ways such as a queue or a stack, but it is not appropriate for the implementation of a scheduler because it does not support the notion of priorities.

Priority queues are used to implement schedulers. The idea is that each process has a priority associated with it. The process with the highest priority is always chosen from the queue of processes which is ready to run. The standard template library provides a priority_queue template class which implements this concept.

Lists are also used to implement queues. A queue is a first-in first-out data structure. Elements are added to the back of the queue and removed from the front of the queue. Lists are ideal for implementing queues because they support constant time insertion and deletion at the ends of the list. Vectors are not suitable for implementing queues efficiently because inserting and deleting elements at the front of a vector requires linear time.

Another common application of queues is simulation. Computer simulations are often used to model events in the real world. For example, a simulation of a bank might be used to determine how many tellers are required to ensure that customers do not wait too long in line. Simulations are also frequently used to model the behavior of components in integrated circuits before the circuits are physically built. Typically, simulations involve maintaining a queue of customers. Each customer has associated with it the time that they entered the queue and the time required to service their transaction. When a teller finishes with a customer, the next customer is removed from the queue. The customer's wait time can then be computed by subtracting the time they entered the queue from the current time. Many variations of this basic simulation are possible. For example, customers may balk (leave immediately) if the queue is too long when they arrive. Or, customers may switch lines if another line is shorter.

Another application of queues is discrete event simulation [Bryant 1977, Misra 1986]. Here, a single global clock is maintained by the simulation program. Each object in the system being simulated has an associated simulation time clock. The object's clock indicates how far into the future the object has been simulated. Objects communicate with each other by sending time-stamped messages. A message is not processed by the receiving object until the receiving object's clock reaches the time stamp on the message. Queues are used to hold messages that are waiting to be processed.

4.4 Stacks in the Standard Template Library

A generic stack class is also provided in the standard template library. The stack class is not a container class in its own right. Rather, it relies on the services of an underlying container class to actually store the elements. The stack class is said to be a container adaptor because it takes an existing container class and adapts it to support stack operations. By default, the stack class uses the deque container class as the underlying implementation. However, the user can also choose either list or vector with the following declarations:

```
stack<int> stack1;                    // deque by default
stack<int,vector<int> > stack2;       // vector
stack<int,list<int> > stack3;         // list
```

Member functions in the container stack are listed in Figure 4.11. Note that the return type of top() is void and is not a reference to the top element. To access the top element, the member function top() has to be used. Therefore, the popping operation discussed in this chapter has to be implemented with a call to top() followed by a call to pop(). Because popping operations in user programs are not suited for capturing the popped element most of the time and not only for

Figure 4.14 A list of stack member functions.

Member Function	Operation
bool empty() const	returns true if the stack includes no elements and false otherwise
void pop()	removes the top element of the stack
void push(T)	inserts an element at the top of the stack
size_type size() const	returns the number of elements in the stack
stack()	creates an empty stack
T& top()	returns a reference to the top element
const T& top() const	returns a const reference to the top element

removing it. The desired popping operation is really a sequence of the two member functions from the container's stack. If we call them in one operation, a new class can be created that inherits all the operations from a stack and redefines pop(). This is a solution we discuss in the case study at the end of the chapter.

4.5 Queues in the Standard Template Library

The queue container is not determined by default as the container deque and the user may opt for using the container list instead. An attempt to use the container vector results in a compilation error. Since the pop() is implemented as a call to pop_front(), which is assumed to be a member function of the underlying container and vector does not include such a member function. For the list of queue's member functions see Figure 4.15. A short program in Figure 4.16 illustrates the operations of the member functions. Note that the dequeueing operation discussed in this chapter is implemented by front() followed by pop(), and the enqueueing operation is implemented with the function push().

4.6 Priority Queues in the Standard Template Library

The priority_queue container (Figure 4.17) is implemented with the container vector by default, and the user may choose the container deque. priority_queue maintains an order in the queue by keeping an element with the higher priority in

FIGURE 4.15 A list of queue member functions

Member Function	Operation
`TK back()`	[illegible]
`const TK back() const`	[illegible]
`bool empty() const`	[illegible]
`TK front()`	[illegible]
`const TK front() const`	[illegible]
`void pop()`	[illegible]
`void push(TK)`	[illegible]
`queue()`	[illegible]
`size_type size() const`	[illegible]

FIGURE 4.16 An example application of queue's member functions

```
#include <iostream>
#include <queue>

using namespace std;

void main() {
    queue<int> q1;
    queue<int,list<int> > q2; // Choose a queue between single precision > >
    q1.push(1); q1.push(2); q1.push(3);
    q2.push(4); q2.push(5); q2.push(6);
    q1.push(q2.back());
    while (!q1.empty()) {
        cout << q1.front() << " ";     // 1 2 3 6
        q1.pop();
    }
    while (!q2.empty()) {
        cout << q2.front() << " ";     // 4 5 6
        q2.pop();
    }
}
```

Table 4.17 A list of some priority_queue member functions.

Member function	Operation
`bool empty() const`	returns true if the priority_queue has no elements and false otherwise
`void pop()`	removes the element at the top of the priority_queue
`void push(x)`	uses the push operation to insert x into the priority_queue
`priority_queue()`	creates an empty priority_queue that uses the operator < to determine priority
`priority_queue(first, last, f)`	creates a priority_queue that uses the function f to order elements of the queue and adds the elements from the range indicated by iterators first and last
`size_type size() const`	returns the number of elements in the priority_queue
`T& top()`	returns a reference to the element in the priority_queue with the highest priority
`const T& top() const`	returns the element in the priority_queue with the highest priority

front of the queue. To accomplish this, a two argument Boolean function is used by the insert operation push(), which removes the elements in the queue as it satisfies its requirement. The function can be supplied by the user otherwise, the operation < is used and the element with the highest value is considered to have the highest priority. If the highest priority is determined by the smallest value then the function object greater should be used to indicate that pushes should apply the operator > rather than < in making decisions when inserting new elements to the priority queue. An example is shown in Figure 4.15. The priority_queue pq1 is defined as a vector based queue that uses the operation < to determine the priority of integers in the queue. The second queue, pq2, uses the operation > during insertion. Finally, the queue pq3 is of the same type as pq1 but it is also initialized with the values from the array a. The destructive loops show in which order the elements can be taken out of these queues and sequenced.

It is more interesting to see an application of the user defined objects. Consider the class Person defined in Section 1.6.

```
class Person {
public:
    . . . . . .
    bool operator<(const Person& p) const {
        return strcmp(name, p.name) < 0;
```

Figure 4.18 A program that uses member functions of the container priority_queue.

```
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

void main() {
    priority_queue<int> pq1; // plus <vector<int>> and less<int>
    priority_queue<int,vector<int>,greater<int> > pq2;
    pq1.push(3); pq1.push(1); pq1.push(2);
    pq2.push(3); pq2.push(1); pq2.push(2);
    int a[] = {4,6,5};
    priority_queue<int> pq3(a,a+3);
    while (!pq1.empty()) {
        cout << pq1.top() << " ";     // 3 2 1
        pq1.pop();
    }
    while (!pq2.empty()) {
        cout << pq2.top() << " ";     // 1 2 3
        pq2.pop();
    }
    while (!pq3.empty()) {
        cout << pq3.top() << " ";     // 6 5 4
        pq3.pop();
    }
}
```

```
    };
    bool operator<(const Person& p1, const Person& p2) {
        return p1.priority < p2.priority;
    }
private:
    char name[20];
    int age;
};
```

Our friend function is overloaded three times in the first two cases the priority is determined by less than and greater than respectively, in the third case by the age and in the second by the name. Furthermore, operator< is overloaded again above. The priority queue pq3 uses the overloaded operator< to create a priority queue of the form (most to least) Person:

```
Person p[] = {Person("Cheng",25), Person("Sun",30), Person("Bill",20)};
priority_queue<Person> pqNatural(p,p+3);
priority_queue<Person,vector<Person>,CompareYear> pqYear(p,p+3);
```

In these two declarations, the two priority queues are also initialized with objects from the array p.

In Section 4.3 there is also a Boolean function lessPerson used to determine the order of Person objects by age, not by name. How can we create a priority queue in which the highest priority is determined by age? One way to accomplish this is to define a function object

```
class lessPerson {
public:
bool operator()(const Person& p1, const Person &p2) const {
    return p1.age < p2.age;
}
};
```

and then declare a new priority queue

```
priority_queue<Person,vector<Person>,lessPerson> pqAge(p,p+3);
```

initialized with the same objects as pqNatural and pqName. Printing contents from the three queues indicates the different priorities of the objects in different queues:

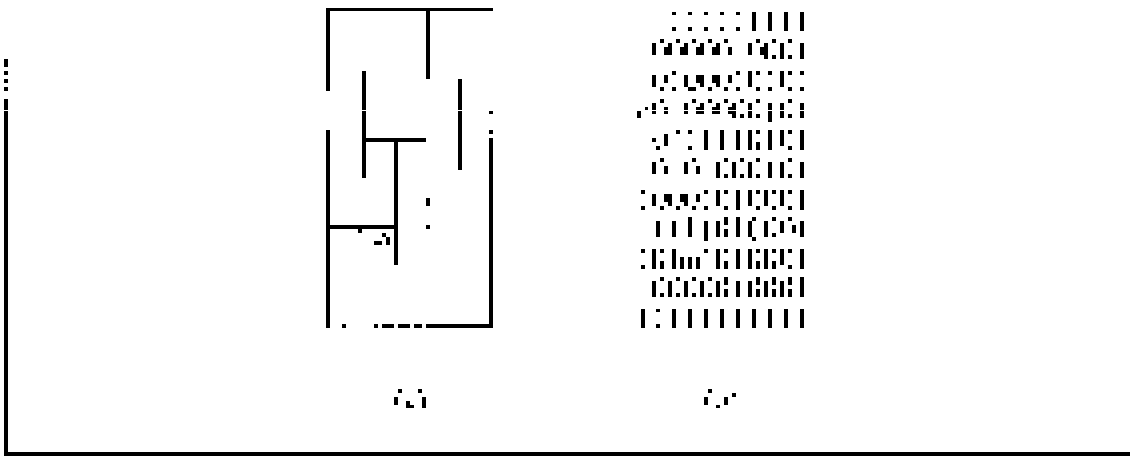
```
pqNatural:    (Cheng,25) (Bill,20) (Sun,30)
pqName:       (Sun,30) (Bill,20) (Cheng,25)
pqAge:        (Sun,30) (Cheng,25) (Bill,20)
```

4.7 Case Study: Exiting a Maze

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze (Figure 4-5). The mouse hopes to escape from the maze by systematically trying all the paths. If it reaches a dead end, it retraces its steps to the last position and begins a new, one more untried path. For each position, the mouse can go in one of four directions: right, left, down, up. Regardless of how clever it is, the mouse always tries the open paths in the order which may lead to some unnecessary detours. By recording information that allows for resuming the search after a dead end is reached, the mouse uses a method called backtracking. This method is discussed further in the next chapter.

The maze is implemented as a two-dimensional character array in which path segments are marked with the value ' ' and a position by the letter e and the initial position of the mouse by the letter m (Figure 4-6 shows this program, the maze problem is slightly generalized by allowing the exit to be in any position of the maze; picture the exit position as having an eleventh row and column). The mouse starts up and moving path segments to be on the underlining. To prevent itself from falling off the array by trying to continue its path when an open cell is reached on one of the borders, the maze is surrounded by walls. The mouse checks constantly whether it is on such a border line position or not. To avoid it, the program automatically puts a frame of walls around the maze entered by the user.

FIGURE 4.19 (a) A maze to traverse; (b) two-dimensional character array representing the situation.



The program uses two stacks: one to initialize the maze and another to implement backtracking.

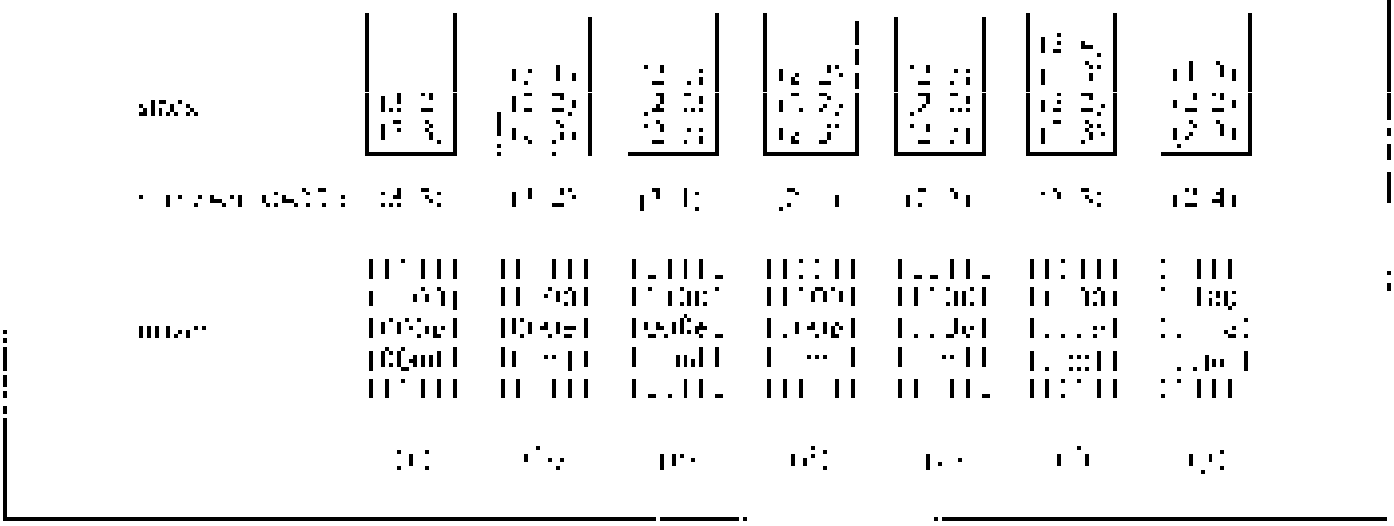
The user selects a maze outline at first. The file selected by the user can have any number of rows and any number of columns. The only assumption the program makes is that all rows are of the same length and that at least one blank character is in row number 0, column number one. The rows are pushed on the stack nonblanks in the order they are entered after attaching one blank at the beginning and one blank at the end. After all rows are entered, the size of the array store can be determined, and then the rows stored in the stack are transferred to the array.

A second stack, `pathStack`, is used in the process of escaping the maze. To remember unvisited paths for subsequent tries, the positions of the untried neighbors of the current position (if any) are saved on a stack and always in the same order: first upper neighbor, then lower, then left, and finally right. After stacking the neighbors on the stack, the cursor takes the topmost position and tries to move it by first storing its new neighbors and then moving to topmost position, and so forth, until it reaches the exit or exhausts all possibilities and finds itself trapped. To avoid falling into an infinite loop of trying paths which have already been investigated, each visited position of the maze is marked with a period.

Here is a pseudocode of an algorithm for escaping a maze.

```
tryMaze()
    Initialize stack, exitCell, entryCell, sourceCell = entryCell;
    while sourceCell is not exitCell
        mark sourceCell as visited;
        push onto the stack the unvisited neighbors of sourceCell;
        if stack is empty
            failure;
        else pop off a cell from the stack and make it sourceCell;
    success;
```

Figure 4.20 An example of processing a maze.



The stack stores coordinates of positions of cells. This could be done for instance by using two integer stacks for x and y coordinates. Another possibility is to use one integer stack with both coordinates stored in one integer variable with the help of the shifting operator. In the program in Figure 4.21 a class `coordCell` is used with two data fields x and y so that one `LinkedStack` is used for storing `coordCell` objects.

Consider an example shown in Figure 4.20. The program actually prints out the maze after each step marked by the numbers.

It first reads in the maze

```
1100
0000
0001
```

The maze is immediately surrounded with a frame of 1s

```
111111
110001
100001
100001
111 11
```

`entryCell` and `exitCell` are initialized to (1, 1) and `exitCell` to (2, 4) (see Figure 4.20(a)).

Because `entryCell` is not equal to `exitCell`, its four neighbors of the current cell (1, 1) are tested, and only two of them are candidates for processing, namely, (1, 2) and (2, 1). Therefore, they are pushed onto the stack. The stack is checked to see whether it contains any previous cell because it is not empty, the top most position, (1, 2) becomes current (Figure 4.20(b)).

2. `cell[3][2]` is still not equal to `exitCell`; therefore, the two viable options accessible from `[3][2]` are pushed onto the stack, namely, positions `[2][2]` and `[3][1]`. Note that the position holding the mouse is not included in the stack. After the current location is marked as visited, the situation in the maze is as in Figure 4.20c. Now, the topmost position, `[3][1]`, is popped off the stack and it becomes the value of `exitCell`. The process continues until the exit is reached, as shown step by step in Figure 4.20d–f.

Now, but in step four (Figure 4.20d), the position `[2][2]` is pushed onto the stack, although it is already there. However, this causes no danger, because when the second instance of this position is popped from the stack, all the cells leading from this position toward each boundary will be investigated using the first instance of the position on the stack. Note also that the mouse never crosses through the wall in the same position from its initial position to the exit.

Figure 4.21 contains a complete implementation of the maze-solving algorithm. Note that the program defines a class `Stack` for integer stacks. The code in lines 6–11 of `main` creates a stack `path` and then calls `push` to push the integer 1 onto `path`. The `while` loop in lines 12–17 pops the top element from the stack and returns it as the value

Program 4.21 Listing of main program for maze processing.

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

template<class T>
class Stack : public stack<T> {
public:
    T pop() {
        T tmp = top();
        stack<T>::pop();
        return tmp;
    }
};

class Cell {
public:
```

Figure 4.27 (continued)

```
    Cell(int x = 0, int y = 0) {
        x = x; y = y;
    }
    bool operator==(const Cell& c) const {
        return x == c.x && y == c.y;
    }
    bool operator<(const Cell& c) const { // to satisfy the compiler
        return true;
    }
private:
    int x, y;
    friend class Maze;
};

class Maze {
public:
    Maze();
    void exitMaze();
private:
    Cell currentCell, exitCell, entryCell;
    const char exitMarker, entryMarker, visited, passage, wall;
    Stack<Cell> mazeStack;
    char **store;          // array of strings
    void printVisited(int,int);
    friend ostream& operator<<(ostream&, const Maze&);
    int rows, cols;
};

Maze::Maze() : exitMarker('e'), entryMarker('n'), visited('.'),
               passage(' '), wall('1') {
    Stack<char*> mazeRows;
    char str[80], *s;
    int col, row = 0;
    cout << "Enter a rectangular maze using the following "
         << "characters:\n\n  entry\n  exit\n  wall\nG  passage\n"
         << "Enter one line at at time; end with Ctrl z\n";
    while (cin >> str) {
        row++;
        cols = strlen(str);
```

Listing 4.21 (continued)

```
        s   new char[cols*3];     // two more cells for boundary on
                                  // columns;
        mazeRows.add(s);
        Arrays.fill(s,' ');
        s[0]   = s[cols-1] = wall; // fill the boundaries with Is;
        s[cols*2  - 1] = '\0';
        if (exitChr[0].mx -MazeKey == 0) {
            exitCellx = cox;
            exitCelly = exitChr[0].mx -MazeKey = 0;
        }
        if (entryChr[0].entryMarker != 0) {
            entryCellx = cox;
            entryCelly = atoi(&s[0],entryMarker) - 0;
        }
    }
    rows = cox;
    block = new char[rows*3];          // create a 1d array of pointers:
    block 3   = new char[cols*3];      // a border is now;
    for ( : !mazeRows.empty(); rows--) {
        block row_ = mazeRows.pop();
    }
    block block 1] = new char[cols*3]; // another border is now;
    block 2 [cols* _ = exits[rows* ][cols* _ = '\0';
    for (col = 0; col <= cols* ; col++) {
        block  ][col] = wall;          // fill the borderline rows with Is;
        block rows- ][col_ = wall;
    }
}

void Maze::pushUnvisited(int row, int col) {
    if (mtrx[row][col  _ == passage || mtrx[row,col] == exitCell) {
        rowStack.push(cell(row,col));
    }
}
void Maze::exitMaze() {
    i = row, col;
    currentCell = entryCell;
    while (!currentCol  == exitCell) {
        row = currentCell.x;
```

Continues

Program 4.21 (continued)

```
        mz = new int[rows][cols];
        cout << "\n\n";          // print a snapshot
        for (int i=0; i<rows; i++)
            for (int j=0; j<cols; j++)
                mz[i][j] = maze[i][j];
        mz[entryRow][entryCol] = 2;
        mz[exitRow][exitCol] = 3;
        for (int i=0; i<rows; i++) {
            for (int j=0; j<cols; j++)
                cout << mz[i][j];
            cout << endl;
        }
        cout << endl;
    }
    cout << "Exit\n";
    cout << "Success\n";
}
ostream& operator<< (ostream& out, const Maze& maze) {
    out << maze.rows << " " << maze.cols << endl;
    for (int i = 0; i < maze.rows; i++) {
        for (int j = 0; j < maze.cols; j++)
            out << maze.maze[i][j] << " ";
        out << endl;
    }
    out << endl;
    return out;
}
void main() {
    Maze().exitMaze();
}
```

4.8 EXERCISES

1. How can you implement a stack S
   a. using two additional stacks
   b. using one additional queue
   c. using one additional stack and some additional variables
2. Put the elements on the stack S in ascending order using one additional stack and some additional variables.

3. Transfer elements from stack S_1 to stack S_2 so that the elements from S_1 are in the same order as they were in S_1.
   a. Using one additional stack
   b. Using no additional stack but only some additional variables

4. Suggest a way to allocate storage for stacks to hold elements of two different types, such as integers and floating-point numbers.

5. Using additional variables, order all elements in a circular singly linked list.
   a. Using two additional queues
   b. Using one additional queue

6. In this chapter, we described implementations of stacks based on singly linked lists and on arrays. Discuss the relative merits of these two implementations with respect to the following criteria: (1) ease of implementation, (2) space utilization in the presence of a wide variety of stack sizes, and (3) time efficiency of stack operations. Which implementation would you choose for a stack that grows and shrinks over the lifetime of the program? Justify your answer.

7. Implement a stack class for numbers in which the push operation checks for stack overflow. The class should provide a default error-handling function that is called when a push operation is attempted on a full stack. Also, provide a mechanism that allows a user program to install its own overflow-handling function. Discuss the relative merits of the following designs for installing error-handling functions. Consider, in particular, how easy it is to write programs that work correctly in the presence of errors.
   a. `void push(const Item& x, bool &success);`

      The boolean variable `success` would be set by the push operation to indicate whether the push was successful. The calling program would be responsible for testing `success` after every push operation.
   b. `bool push(const Item& x);`

      In this version, the return value indicates whether the push was successful. The calling program would be responsible for testing the return value after every push operation.
   c. `void push(const Item& x);`

      ```
      if (overflow) error();
      else ...;
      ```

      Here, the program would call `error` to handle the error condition. A default error-handling function would print a message and halt the program. Other error-handling functions could be installed by the user.

   What are the disadvantages of these designs?

8. Define a stack in terms of a private character array and a class

   ```
   template <class T>
   class Stack {
       T data[MAX_STACK];
       ...
       void push(const T& x) {
           push_helper(x);
           ...
   ```

9. Define a queue in terms of a stack.

10. A generic queue class by definition in terms of a vector:

```
template<class T,  int capacity = 30>
class queue {
    . . . . . .
public:
    . . . . . .
};
```

with a variable structure:

11. Modify the program from the case study to find out the exit without dead ends and, possibly with detours, for example for an input maze

```
1 1 1 1 1 1
1 e 0 0 0 1
1 1 1 0 1 1
1 0 0 0 0 1
1 0 1 0 0 1
1 1 1 1 1 1
```

The program from the case study outputs the processed maze

```
1 1 1 1 1 1
1 e . . . 1
1 1 1 1 1 1
1 . . . . 1
1 . . 0 . 1
1 1 1 1 1 1
success
```

The modified program should, in addition, generate the path from the exit to the mouse.

(1, 1) (1, 2) (1, 3) (2, 3) (3, 3) (3, 4) (3, 5) (4, 5) (4, 4) (4, 3)

which leaves out two dead ends, (1, 4) (1, 5) and (3, 2) (3, 1) (4, 1) (4, 2), but retains a detour, (3, 4) (3, 5) (4, 5) (4, 4).

12. Modify the program from the previous exercise so that it prints the maze with the path without dead ends. The path is indicated by dashes and vertical bars to indicate the changes of direction of the path. For the same maze from the previous exercise, the modified program should output

```
1 1 1 1 1 1
1 e - - - 1
1 1 | 1 1 1
1 . . | . 1
1 . . | - - 1
1 1 1 1 1 1
```

4.9 Programming Assignments

1. Write a program that reads a line of input text and converts it to uppercase. Then read the next line of input, print it backward (reversed), and then read the next line of input and print it in alternating uppercase and lowercase characters. Finally, print the number of characters in the first line of input. Use as few variables as possible. Avoid duplicating code.

2. Write a program to convert a non-negative decimal integer into a number in another number system whose base is between 2 and 9. The program should prompt the user for the integer and the base. Then it should print the digits of the number in the new number system. For example, if the user enters 10 and 2, the program should print 1010 (because $10 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$). If the user enters 100 and 8, the program should print 144 (because $100 = 1 \times 64 + 4 \times 8 + 4 \times 1$). Finally, if the user enters 29 and 9, the program should print 32 (because $29 = 3 \times 9 + 2 \times 1$). Use a loop that repeatedly computes the number and the digit that corresponds to the ones position (as the remainder after division by the base). Then print the digits in reverse order. Use the base to select the digit strings, for example, if the base is 2, select the digits "0" and "1".

   Modify your program so that it can convert numbers in bases greater than 10. In this case, the base is a number between 11 and 36. Number systems with bases greater than 10 use letters as digit symbols. For example, a hexadecimal (base 16) number system uses the digits 0, 1, ..., 9, A, B, C, D, E, F. In this system, the number 7F is equal to 127 (because $7 \times 16 + 15 = 127$). Use upper case letters for digits that are greater than 10. That is, use "A" and not "a" for the digit that corresponds to 10.

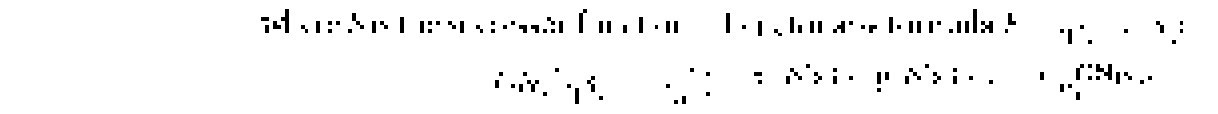
3. Write a program that implements the algorithm from Section 4.1.

4. Write a program that implements the algorithm from Section 4.2.

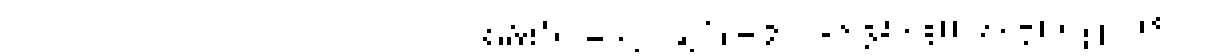
5. Write a program to perform calculations with very large integers. Implement the following operations: addition, subtraction, multiplication, and exponentiation. Use the program to find the first 50 digits of $1000! = 1 \times 2 \times 3 \times \cdots \times 1000$. Also apply the program to compute the exact value of 2^{1000} and other similar expressions.

   Use Gödel numbers to form a one-to-one correspondence between basic elements of language and numbers.

Symbol	Gödel Number
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
variable v_0	11
variable v_1	12



and



In this way, more and more expressions can be assigned to unique numbers. This method has been used by Gödel to prove theorems concerning the completeness and consistency of mathematics.

6. Write a program to evaluate postfix expressions involving large floating-point numbers. Extend this program to handle other arithmetic operations.

Bibliography

Queues

Some Unified Queue Algorithms, Communications of the ACM, October 1973, 16(10): 624–645.

Priority Queues

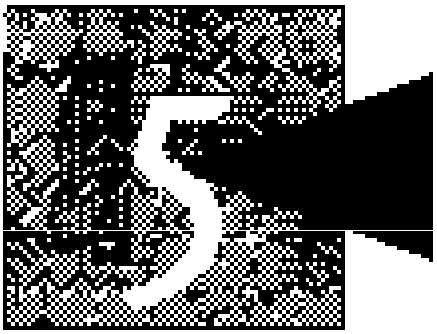
Brown, Mark R., Implementing Portable Priority Queues for Discrete Event Simulation, Report No. STAN-CS-79-770, Computer Science Department, Stanford University, Stanford, CA, 1979.

Cheriton, David R., and Robert E. Tarjan, Finding Minimum Spanning Trees, SIAM Journal on Computing, December 1976, 5(4): 724–742.

Horowitz, Ellis, and Sartaj Sahni, Fundamentals of Data Structures, Computer Science Press, Rockville, MD, 1983, pp. 351–418.

Jones, Douglas W., An Empirical Comparison of Priority-Queue and Event-Set Implementations, Communications of the ACM, April 1986, 29(4): 300–311.

Recursion



5.1 Recursive Definitions

One of the basic principles for developing correct programs is to understand the problem well enough to find a way to break it down into smaller problems that have already been solved. Frequently, these smaller problems are of the same type as the original problem—solving them requires breaking them down into even smaller problems of the same type. Eventually, the problems become so small that they can be solved directly without breaking them down further. The process of breaking a problem down into smaller problems that are of the same type as the original problem is called recursion. In this section, we will see how recursion can be used to define sets, functions, and algorithms. We will also see how recursion can be used to prove that algorithms work correctly.

When defining a set, we usually describe the elements that are in the set in terms of other elements in the set. For example, we might define the set of even natural numbers by saying that a number is even if it is the sum of two even numbers. Or we might define the set of natural numbers by saying that 1 is in the set and that each number in the set is one more than some number in the set. Definitions like these are called *recursive definitions*. Recursive definitions have two parts. The *basis* specifies that certain elementary objects are in the set. The *recursive part* specifies how to construct new objects in the set from objects already known to be in the set. Sometimes a recursive definition will also have an *exclusion* part, which specifies that no objects other than those constructed from the basis and recursive parts are in the set. For example, a recursive definition of the set N of natural numbers might be given as:

1. $0 \in N$.
2. If $n \in N$, then $n + 1 \in N$.
3. These are the only elements in the set N.

Clause 3 is understood in all definitions of this type, but it is usually omitted. Clauses like it will be omitted in all recursive definitions in this book.

According to these rules, the set of natural numbers $\mathbb{N}$ consists of the following items: 0, $0+1$, $0+1+1$, and so on. Although the set $\mathbb{N}$ contains objects (namely such objects), for we call natural numbers, the definition results in a procedure that repeatedly "adds" elements. Can you imagine how to generate the next large numbers using such a specification? Therefore, it is more convenient to use the following definition, which doesn't assume the whole range of the natural numbers to begin:

1. $0 \in \mathbb{N}$,
2. if $n \in \mathbb{N}$, then $n+1 \in \mathbb{N}$,
3. these are the only natural numbers.

Then the set $\mathbb{N}$ includes all possible combinations of the successor function $f(n) = n+1$.

Let's see definitions as a set of two processes: generating new elements, as we already learned, and testing whether an element belongs to the set. In the case of testing, the problem is solved by reduction to a simpler problem, and if the simple problem is still too complex it is reduced to a more simple problem, and so on, until it is reduced to a problem that can be solved without reduction. For instance, is 3 a natural number? According to the second condition of the definition, the set $\mathbb{N}$ contains 1 if $0 \in \mathbb{N}$, and the first condition already says that $0 \in \mathbb{N}$. Hence $1 \in \mathbb{N}$, $2 \in \mathbb{N}$, and finally $3 \in \mathbb{N}$.

The ability to describe a set and to solve a problem by reducing it to simpler problems of the same kind is characteristic of recursive definitions. We will see more examples of recursive definitions in later chapters, and we will learn more about it in the next chapter.

Recursive definitions are frequently used to define functions and sequences of numbers. For instance, the factorial function $n!$ can be defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (base clause)} \\ n \cdot (n-1)! & \text{if } n > 0 \text{ (inductive step)} \end{cases}$$

Let's apply this definition to evaluate $4!$ step by step:

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24$$

Here is another example of a definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1) + \dfrac{1}{f(n-1)} & \text{if } n > 0 \end{cases}$$

Let's generate the sequence of the first eight terms:

$$1, 2, \frac{5}{2}, \frac{29}{10}, \frac{941}{290}, \frac{969581}{272890}, \ldots$$

Recursive definitions of sequences are very convenient for computers. To compute the value of an element a_n of a sequence we first need to compute the values of some or all of the previous elements $a_0, \ldots, a_{n-1}$. For example, to apply the second definition above to find $f(2)$, we first compute the values of $f(1)$ and $f(0)$. Computationally, this is not always the

[illegible] it [illegible] to make [illegible] [illegible] [illegible]

The problem now seems to be more critical since it is far from clear how a function calling itself can possibly work for more than one call. This chapter shows that it is possible for such a function to work properly. Recursive definitions in most computers are eventually implemented using a run-time stack, although the actual work of implementing recursion is done by the operating system and the source code includes no indication of how it is performed. E. W. Dijkstra introduced this idea of using a stack to implement recursion. To better understand recursion and to see how it works, it is necessary to discuss the processing of function calls and to look at operations carried out by the system at function invocation and function exit.

5.2 Function Calls and Recursion Implementation

What happens when a function is called? If the function has formal parameters, their values must be initialized to the values passed as actual parameters in the call. The system has to know where to resume execution of the program after the function has finished. The function can be called by other functions or by the main program, and the function may itself call other functions directly or indirectly. This is an important point, as it has been argued from time to time that recursion is a special case and requires special treatment by the system. This can be shown by stating that recursion is unnecessary because any recursive function can be rewritten using only iteration and explicit stacks. Such a rewrite may be desirable in some cases, but we do not usually want to do it ourselves. Instead, we should let the system handle the stacking for us.

Let us first consider what information has to be saved when a function is called. To refresh your memory, recall that the run-time stack is a memory region set aside for such purposes. But what precisely must be saved when a function is called? Consider the following variable declarations in function f1():

```
int f1()
{
    int x, y;
    x = f2();
    y = f2();
    return x + y;
}
```

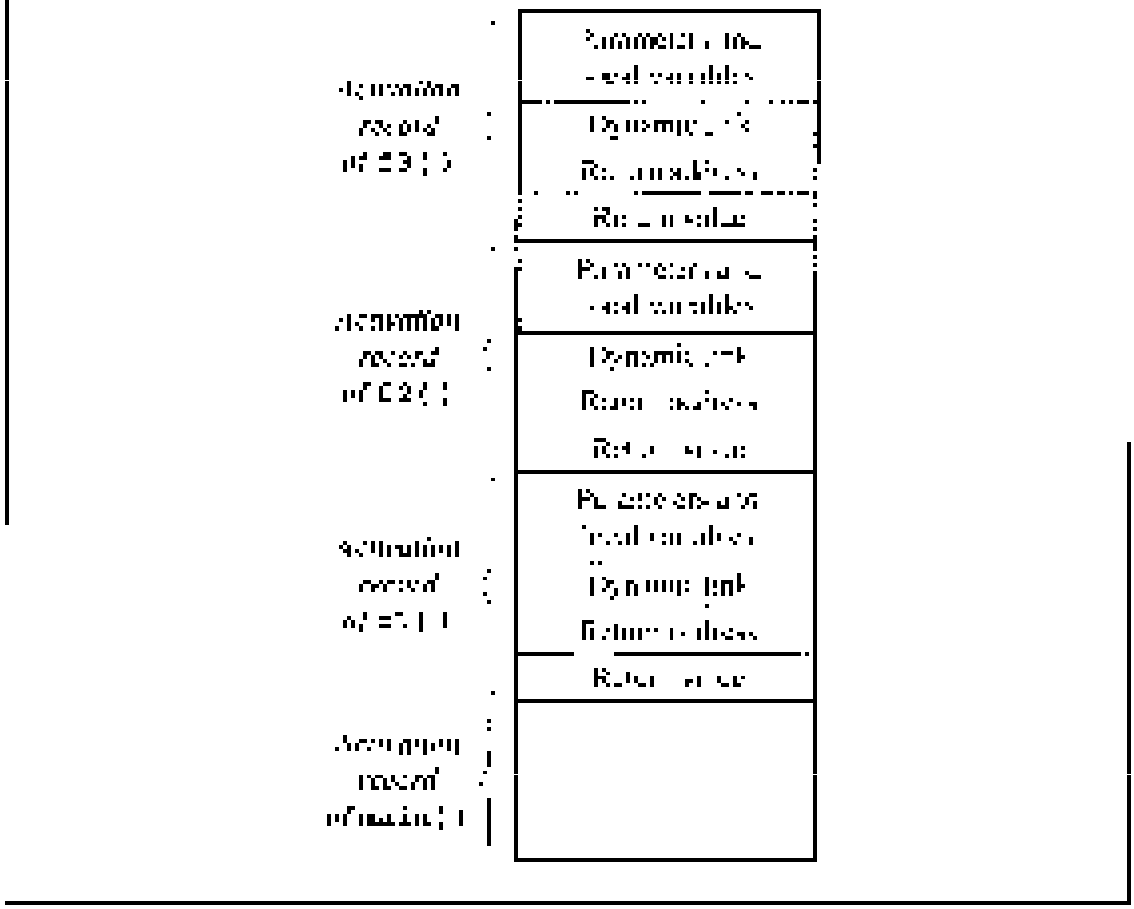
Function f1() calls function f2() twice. The variables x and y are local to f1(). The first call to f2() assigns a value to x, but when f2() is called the second time, the value assigned to x must be left unchanged. When f2() is terminated, f1() can use the value assigned to x to compute the return value. This is especially important in the context of the previous chapter, where f1() is the same as f2(), when a function calls itself recursively. How does the system maintain a distinction between these two variables x?

The state of each function, including main(), is described by the contents of its formal parameters, by the values of the function's parameters, and by the return address—that is, the location in the code where execution must resume after the function terminates. This information is called an activation record or stack frame and is allocated on the run-time stack. An activation record exists for as long as a function is executing. This record is a private pool of information for the function, a repository that stores all information necessary for its proper execution and how to return to where it was called from. Activation records usually have a short life span because they are dynamically allocated at function entry and deallocated upon exiting. Only the activation record at the top of the stack defines the current state of execution, since it contains all the information necessary to continue execution when the function terminates. All other activation records further down the stack are waiting for functions to return control to them. Figure 5.1 illustrates the run-time stack and the activation records for the following program:

```
int main()
{
    int a, b;
    a = 5;
    b = f1(a);
    return 0;
}

int f1(int x)
{
    int y;
    y = x + 2;
    return y * f2(x);
}

int f2(int x)
{
    return x * x;
}
```

An activation record usually contains the following information:

FIGURE 5.1 Control stack for a call sequence such as procedure A calling B, which then calls C and C calls B.



- Values for all parameters to the function declaration of the form (call by value) or pointers to actual parameters passed by reference, in the order of declaration of the parameters.
- Local (automatic) variables which can be stored elsewhere than in the activation record, containing only their descriptors and pointers to the locations where they are stored.
- Pointer to the instruction immediately following the call in the caller's code, to return to immediately following the call.
- A dynamic link, which is a pointer to the caller's activation record.
- Temporaries and bookkeeping information that is needed to restore the state of the machine to the point at which the procedure was called.

As mentioned above, the parameters are allocated by the caller in the caller's activation record, thus they are accessible to the called procedure through pointers stored in its activation record. The exact order in which the parameters are allocated is unimportant, but it must be consistent for all calls to the same procedure. Thus, in the call to B(1, 2) from A, the parameters 1 and 2 are pushed onto the stack in that order, and in the call to B(3, 4) from C, 3 is pushed and then 4. Before the call to B from A, the state of the stack is as shown in Figure 5.1. Before the call to B from C, the state of the stack is as shown in Figure 5.2.

the stack, if the return value for f() is stored by moving the stack pointer right below the return value of g(), then g() returns execution and moves its free access to the private pool of information necessary for invocation of its executor. On the other hand, if g() happens to call another function or f() itself, the run-time stack increases in height, and the activation record for g() is created on the stack and the memory of g() is safe and sound.

Creating an activation record for a function is called allocating the system stack frame for the function properly. Returning from a function call happens in the same order as the calls. The execution of a function usually involves calling itself but a different instantiation of a function may also be called, a variation of the same original. These invocations are expressed naturally by activation records and we thus defer to a later discussion.

5.3 Anatomy of a Recursive Call

The function that defines raising a real number to a non-negative integer power is a good example of a recursive function. The most natural definition of this function is given by

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

A C++ function for computing x^n is shown next directly from the definition of a power:

```
/* 001 */ double power (double x, unsigned int n)
/* 002 */ {
/* 003 */     if (n == 0)
/* 004 */         return 1.0;
              // else
/* 005 */     return x * power (x, n-1);
          }
```

Using this definition, the value of x^n is computed in the following way:

$$\begin{aligned} x^n &= x \cdot x^{n-1} = x \cdot (x \cdot x^{n-2}) = x \cdot (x \cdot (x \cdot x^{n-3})) \\ &= x \cdot (x \cdot (x \cdot (\cdots (x \cdot x^0)))) \\ &= x \cdot (x \cdot (x \cdot (\cdots (x \cdot 1)))) \end{aligned}$$

The repeated application of the definition steps over itself leads to the ancestor when the base step is reached and execution unwinds. The activation records for each of the n calls to the power function are stored on the system stack in the memory area of the run-time stack. Now, let us take a closer look at the activation record for the third call, which is where the argument x is equal to 2.0 and the argument n is equal to 1. At this point, the activation record for the first invocation of power() is already popped off the stack, and the activation record for the second invocation of power() is also popped off the stack. The activation record for the third invocation of power() is shown in Figure 5.1. The activation record for the third call contains the value of the parameter x, which is 2.0, and the value of the parameter n, which is 1. The return value is not yet computed. The activation record for the third call also contains the return address, which is the address of the statement in the function power() that will be executed after the third call returns. The return address is the address of the statement in line 005 that multiplies x and the return value of the function power(). This call

[illegible] [illegible] [illegible] [illegible]

```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

[illegible]

```
call 1          power(x,4)
call 2              power(x,3)
call 3                  power(x,2)
call 4                      power(x,1)
call 5                          power(x,0)
call 6                          1
call 7                      x
call 8                  x*x
call 9              x*x*x
call 10         x*x*x*x
```

What does the system do as the function is being executed? As we already know, the system keeps track of all calls in its run-time stack. Each line of code is assigned a number by the system, and it is that line of the function call that its successor is returned to. The address is used by the system to remember where to resume execution after the function has completed. For this example, assume that the lines in the function power() are assigned the numbers 100 through 105 and that it is called main() from the statement

```
main()
{ ...
/* 125 */  w = power(3,6,2);
  ...
}
```

A trace of the recursive calls is relatively simple, as indicated by the diagram

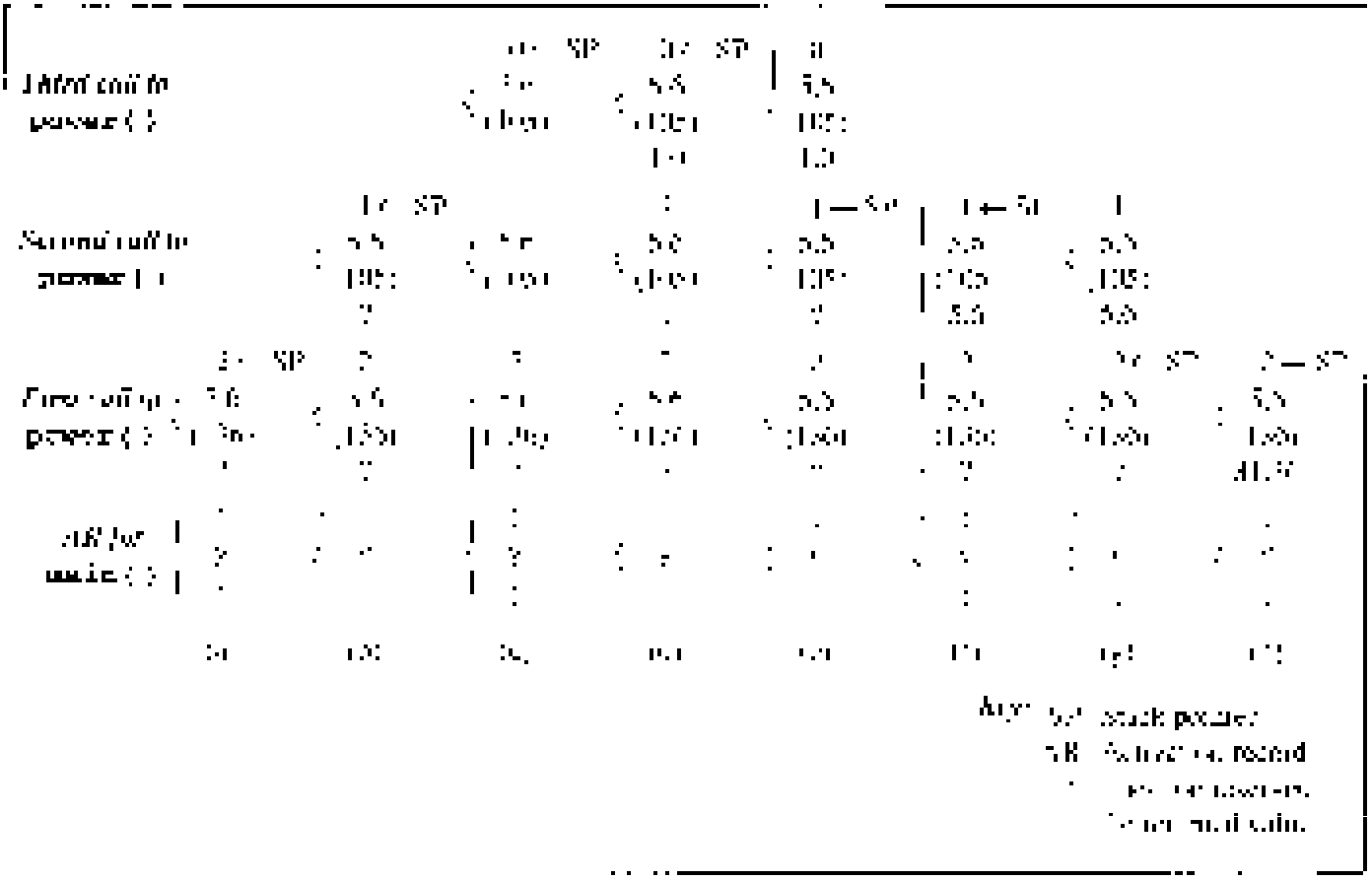
```
call 1          power(3,6,2)
call 2              power(3,6,1)
call 3                  power(3,6,0)
call 4                  1
call 2              3*1
call 1          3*3=9
```

Since most of the operations are performed on the run-time stack

---

[1] This is not quite precise, since the system uses machine code rather than source code to execute programs. The numbers in the run-time stack may, in fact, refer to machine-language instructions.

FIGURE 5.2 Changes to the run-time stack during execution of power(5.0, 3)



When the first call to power() is made, the parameters 5.0 and 3 are pushed onto the run-time stack (see Figure 5.2). Local variables result and i are also created. Control then passes to the body of power(), and the while loop begins. Since i is not equal to 0, result is set to 5.0 (5.0 * 1.0), and i is set to 2 (3 - 1). Next, the while loop is executed again, and result is set to 25.0 (5.0 * 5.0), and i is set to 1 (2 - 1). Finally, the while loop is executed a third time, and result is set to 125.0 (25.0 * 5.0), and i is set to 0 (1 - 1). Since i is now 0, control passes to the return statement, which returns the value of result (125.0) to the point in the program where power() was called—namely, to the assignment statement in the main() function. The value 125.0 is then assigned to the variable x, and the activation record for power() is popped off the run-time stack. Execution resumes with the output statement in the main() function, which displays the value of x (125.0) on the screen.

Now let's trace the execution of the second call to power(). This time, the parameters 5.0 and 0 are pushed onto the run-time stack (see Figure 5.2). Local variables result and i are also created. Control then passes to the body of power(), and the while loop begins. Since i is equal to 0, the while loop is skipped, and result is set to 1.0 (the value of any number raised to the power of 0 is 1). Next, the return statement is executed, and result is returned to the point in the program where power() was called—namely, to the assignment statement in the main() function. The value 1.0 is then assigned to the variable y, and the activation record for power() is popped off the run-time stack. Execution resumes with the output statement in the main() function, which displays the value of y (1.0) on the screen.

Finally, the third call to power() is made. This time, the parameters 0.0 and 3 are pushed onto the run-time stack (see Figure 5.2). Local variables result and i are also created. Control then passes to the body of power(), and the while loop begins. Since i is not equal to 0, result is set to 0.0 (0.0 * 0.0), and i is set to 2 (3 - 1). Next, the while loop is executed again, and result is set to 0.0 (0.0 * 0.0), and i is set to 1 (2 - 1). Finally, the while loop is executed a third time, and result is set to 0.0 (0.0 * 0.0), and i is set to 0 (1 - 1). Since i is now 0, control passes to the return statement, which returns the value of result (0.0) to the point in the program where power() was called—namely, to the assignment statement in the main() function. The value 0.0 is then assigned to the variable z, and the activation record for power() is popped off the run-time stack. Execution resumes with the output statement in the main() function, which displays the value of z (0.0) on the screen.

Again, the second argument is checked to see if it is 0. Since it is not, power() is called for the third time, this time with the arguments 5.0 and 0. Before the function is executed, the system remembers the arguments and local variables by pushing them onto the stack (see Figure 5.2). Then execution begins with the first statement in the function.

Again, the question arises: Is the second argument ever used? Because it finally has a concrete value—namely, 1—it can be returned and popped off the stack, and the function is finished without making any additional calls. At this point, there are two pending calls on the run-time stack—the calls to power(x, n) that have not been completed. How is this done? The system first evaluates the activation record of power(x, n) that has just finished. This is performed logically by popping all its fields (the result, two arguments, and the return address) off the stack. We say "logically" since physically, on these relocatable machines, only the SP is decremented appropriately. This is important because we do not want the result to be destroyed once it has been used yet. Before and after completion of the last call of power(x, n), the stack looks the same, but the SP's value is changed (see Figures 5.3d and 5.3e).

Now the second call to power(x, n) can complete since it waited for the result of the call power(x, n-1). This result, 1.0, is multiplied by x and stored in the field allocated for the result. After that, the system can pop the current activation record off the stack by decrementing the SP and it can finish the execution of the first call to power(x, n) that needed the result for the second call. Figure 5.3f shows the contents of the stack before overlaying the old values, and Figure 5.3g shows the contents of the stack after this change. At this moment, power(x, n) can finish its first call by multiplying the result of its second call, 1.0, by x (its first argument). Control now returns to the function that invoked power(x, n), and the final value, 2.0, is computed. Right before the assignment is executed, the contents of the stack look like Figure 5.3h.

The tail-recursive power(x, n) can be implemented differently, without using the result fields, as in the following code:

```
double tail_power(double x, unsigned int n) {
  double res = 1;
  for ( ; n > 0; n--)
      res = res * x;
  return res;
}
```

Do not gain anything by using recursion instead of iteration. The recursive version seems to be much more difficult to understand than the original definition of the power function. The definition is simply expressed in terms of n without having the original structure of the definition. The recursive version of power(x, n) is definitely inferior to the iterative version in terms of execution speed. Even though the code for the recursive version is not substantially longer than the iterative version, but for most functions of the natural numbers, the recursive solution is much more concise and elegant.

5.4 Tail Recursion

A recursive definition contains a reference to itself or a function being defined. There are, however, a variety of ways such a reference can be implemented. This reference

[illegible] [illegible] [illegible] [

5.5 Nontail Recursion

Consider a problem that can be solved by a recursive procedure that is not tail recursive. Here is such a procedure that reverses the characters it reads:

```
/* 201 */ void reverse(){
/* 202 */     int c;
/* 203 */     if ((c = getchar()) != '\n') {
/* 204 */         reverse();
/* 205 */         putchar(c);
/* 206 */     }
/* 207 */ }
```

When this program is called, the procedure reverse() is called recursively until a newline character is read. Each call has a different character stored in its local variable c. When the newline character is finally read, reverse() returns, printing that character. Then the preceding call returns, printing the character before it, and so on. Figure 5.3 shows the arguments of the most recent stack frame before each call to reverse() and during the execution of the return statement.

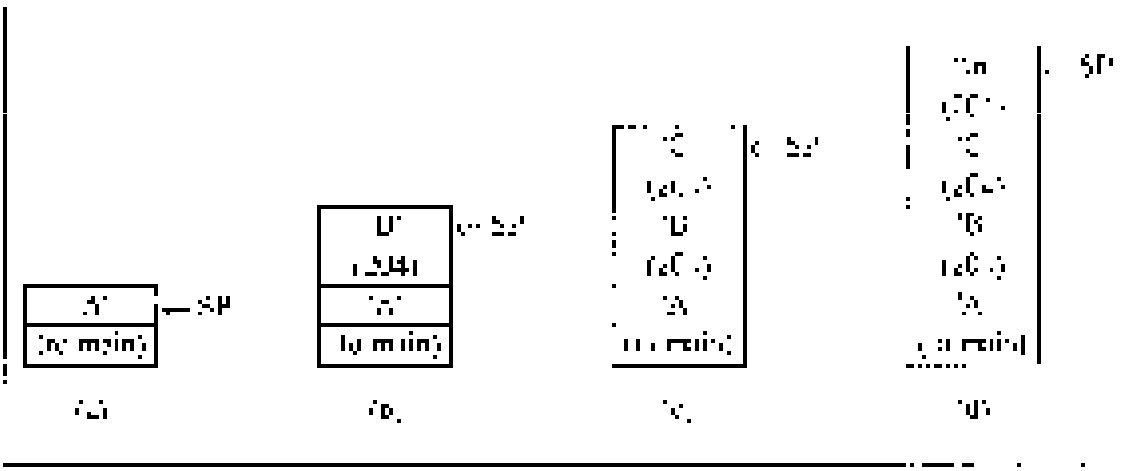


FIGURE 5.3 Changes in the run-time stack during the execution of reverse()

On the fourth call `getc()` fetches the end-of-line character and `reverse()` executes its own instructions. The system retrieves the return address from the activation record and discards that record by decrementing SP by the proper number of bytes. Execution resumes from line 201, which is a `while` statement. Because the activation record of the third call is now active, the value of `c`, the letter "C", is compared to the first character. Next, the activation record of the third call is discarded and now SP points to where "B" is stored. The second call is about to be finished, but the first "B" is assigned to `c` and then the machine instructions are executed, which results in printing "B" on the screen right after "C". Finally the activation record of the first call is reactivated, then "A" is printed, and what can be seen on the screen is the string "CBA." The first call is finished and the program continues execution in `main()`.

Compare the recursive implementation with the standard version of the same function:

```
void simpleIterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.getline(stack,80);
  for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

The function is quite short and, perhaps a bit more cryptic than its recursive counterpart. What is the difference there, keep in mind that the brevity and relative simplicity of the second version are due mainly to the fact that we want to reverse a string or array of characters. The functions that `iostream` like `strlen()` and `getline()` from the standard C++ library will be used. If we are not surprised with such functions then our recursive function has to be implemented first and only.

```
void iterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.getline(stack,80);
  while(stack[top] != '\0')
    top++;
  for (top--; top >= 0; cout.put(stack[top--]));
}
```

The `while` loop replaces `getline()` and the auxiliary variable `top` replaces `strlen()`. The `for` loop is about the same as before. This function is not quite symmetrical because reversing an input line can end up with integers used the same implementation as `iterativeReverse()` after changing the data type of `stack` from `char` to `int` and modifying one `while` loop.

Note that the variable name `stack` used for the array is not accidental. We are just making explicit what is done implicitly by the system. Our stack lives over the run-time stack's dynamic area but is managed by two `for` loops, one ascending as in the case of tail recursion. In addition, the statement `push` from the recursive version has to be accounted for. Note also that the variable `stack` is local to the

[illegible] ; however, if it were a mutable type such as a stack object, then this implementation would be wrong:

```
void myMethod() {
    Stack st;
    st.push(1);
    while (st != null) {
        st.pop();
        st.push(1);
    }
    while (!st.empty()) {
        count += st.pop();
    }
}
```

will [illegible] Stack.Overflow exception outside the function.

Although changing the [illegible] to [illegible] does not solve the problem, it does make it much clearer that the loop is not terminating. Moreover, it is better, as a practical matter, to avoid the [illegible] function altogether, replacing it with [illegible] calls. Finally, a function that [illegible] is not very [illegible] to write.

One way to [illegible] the [illegible] is to [illegible] it. [illegible] is [illegible] to [illegible] the [illegible] stack. [illegible] [illegible] [illegible]

Figure 5.4 Examples of von Koch snowflakes

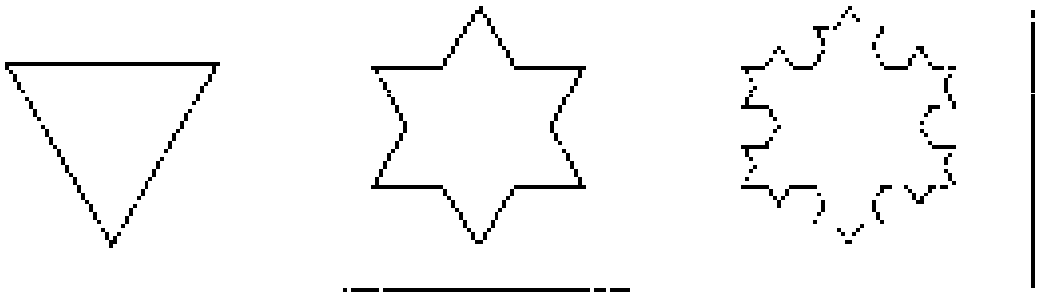
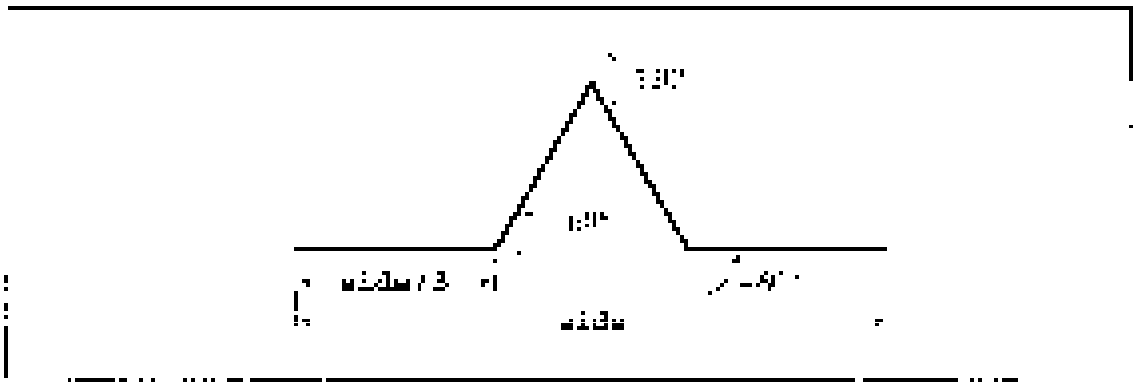


Figure 5.5 An improved drawing for a side of one segment of the von Koch snowflake



Complete the function for the case of the drawing level being greater than 0. Use a statement like the following to draw the first segment of the triangle:

```
drawTo( side/3, level-1 );
```

Then turn the turtle 60 degrees to the left, draw the second segment, turn the turtle 120 degrees to the right, draw the third segment, turn the turtle 60 degrees to the left, and draw the last segment. Each segment is one-third the length of the original side.

This program can be compiled and run without changing the C++ code in Figure 5.6.

Program 5.6

```
//                          Turbo C++ program

#include <iostream.h>
#include <graphics.h>
#include <math.h>

class RecursiveCurve {
public:
    RecursiveCurve();
    void readInitValues();
    void showLake();
private:
    double side, angle;
    int level;
    void right(double x) {
        angle -= x;
    }
    void left (double x) {
        angle    =;
    }
    void drawFourLines(double side, int level);
};
void RecursiveCurve::RecursiveCurve() {
    setcolor(WHITE);
    setbkcolor(BLUE);
    moveto (200,150);
    angle = 0.0;
    cout << "Enter side and level: ";
    cin  >> side >> level;
}
void RecursiveCurve::drawFourLines(double side, int level) {
                                          // arguments to sin();
   if (level    0;                        // and cos(); are angles
       lineto(); cos(angle-x_(2/180);side);  // provided in radians,
              (sin(angle-x_(2/180)*side);  // i.e., use coefficient
   else {                                 // x_(2/180 is necessary;
       drawFourLines(side/3.0,level-1);
       left (60);
       drawFourLines(side/3.0,level-1);

                    ... - ___-
```

Figure 5.6 (continued)

```
        right(120);
        drawFourLines(size/3.0,level-1);
        left(60);
        drawFourLines(size/3.0,level-1);
    }
}
void RecursiveSquare::drawSquare() {
    for (int i = 0; i < 4; i++) {
        drawFourLines(size,level);
        right(90);
    }
}

void main() {
    int gdriver = DETECT, grMode, grResult, grError;
    initgraph(&gdriver,&grMode,"c:\\tc\\bgi");
    grError = graphresult();
    if (grError != grOk) {
        cerr << "Error: " << grapherrormsg(grError) << endl;
        return;
    }
    RecursiveSquare square;
    square.drawSquare();
}
```

5.6 Indirect Recursion

The preceding examples have all been examples of direct recursion. A function is called direct recursive if it calls itself. However, it is possible that a function calls another function, which in turn calls another function, and so on, until finally the original function is called again. This is known as indirect recursion.

For example, consider a set of functions that call each other in a circular fashion:

```
f1 -> f2 -> f3 -> f4 -> ... -> fn-1 -> fn -> f1
```

Figure 5.7 shows a static snapshot of this call graph. You can also think of it as a dynamic chain. Suppose f1 is called from main(). Then f1 calls f2, f2 calls f3, and so on, until finally fn calls f1 again. The corresponding chain of calls is

```
f1 -> f2 -> f3 -> f4 -> ... -> fn-1 -> fn -> f1
```

This situation can be complicated by the fact that functions such as decoding information that is received might store the incoming information in a buffer, decode it, convert it into legible form, and then send it to a file. Assembly fills the buffer and calls decoding, which in turn, after finishing, must call assembly to refill the buffer with decoded information to be assembled. After assembly accomplishes its tasks, it calls receive to get more encoded information from the same buffer. Therefore, we have the chain of calls

```
receive() -> decode() -> assemble() -> receive() -> decode() -> ...
```

which is fine should we know when to stop. Pseudocode for these functions would look as follows:

```
receive(buffer)
    while the buffer is not yet filled up
        if not enough bytes have arrived
            go to sleep and wait for more bytes in buffer
        else read bytes;
        decode(buffer);

decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    empty information from buffer and store it;
    assemble(buffer);
```

A more mathematically oriented example involves functions calculating the trigonometric functions sine and cosine. Recall the Taylor series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

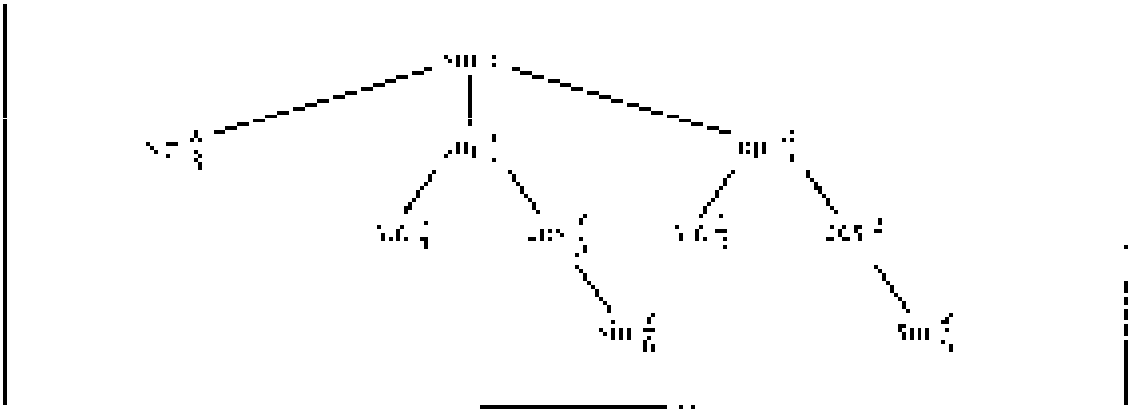
$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

As usual in the case of recursion, there has to be an exit condition to avoid falling into an infinite loop of recursive calls. In the case of sine, we use the following approximation:

$$\sin(x) \approx x - \frac{x^3}{3!}$$

which works for values of x given in radian approximations. To compute the sine of a number x such that its absolute value is greater than an assumed tolerance, we have to compute $\sin\left(\frac{x}{3}\right)$ directly. $\sin\left(\frac{x}{3}\right)$ in turn through usage is also indirectly $\sin\left(\frac{x}{9}\right)$

FIGURE 5.7 A tree of recursive calls for sin(x)



Through tangent and cosine. Local square roots of the values will also be used when computing the cosine. The tree below can represent all the calls as in Figure 5.7

5.7 Nested Recursion

A more complicated use of recursion is found in definitions in which the function is used as an argument to itself. Both the Ackermann function and the following definition are examples of such nesting:

$$m(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ m(2 + m(2n)) & \text{if } n > 1 \end{cases}$$

Function m is well defined for $n \geq 0$. This fact is obvious for $n = 0$ and $n = 1$. One has to be convinced for $n = 2, 3$, and 4. Thus, $m(2) = m(2 + m(4)) = m(2 + 2 + m(8)) = 2$. Other values of $m(n)$, for $n = 1, 3$, and 4?

Another example of nested recursion is a very famous function, which was originally proposed by Wilhelm Ackermann in 1928 and later modified by Rózsa Péter:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{otherwise} \end{cases}$$

This function is interesting because of its remarkably rapid growth. It grows so fast that it is guaranteed not to have a representation by a formula that uses only familiar operations such as addition, multiplication, and exponentiation. To illustrate the rate of growth of the Ackermann function, we need only show that

$$A(3, n) = 2^{n+3} - 3$$

$$A(4,m) = 2^{2^{\cdots^{2}}} - 3$$

with a stack of $m+2$ 2s in the exponent. $A(4,2) = 2^{2^{16}} - 3 = 2^{65536} - 3$, which is vastly more than the number of atoms in the universe (which is 10^{80}, according to current estimates).

The definition translates very nicely into C++, but the task of computing even the second row of the table is truly formidable.

5.8 Excessive Recursion

Logical simplicity and readability are useful criteria for organizing and supporting the process of recursion. The price for using recursion is usually longer execution time and storage on the function stack more things than sequential or iterative approaches. Recursion is not too deep for example, computing $5!^{[\text{FACTR}]}$ takes very little space on the stack and our program crashes. But usually the number of recursive calls is small, say less than 100,000, so the danger of overflowing the stack may not be important.[4] However, if simple recursive functions repeat the computations for some of their parameters, the run time can be prohibitively long even for very simple cases.

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined recursively over

$$F(n) = \begin{cases} n & \text{if } n < 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

The definition is modular. The first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But those predecessors are in turn sums of their predecessors, and so on, back to the beginning of the sequence. The sequence produced by the definition is

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

How can this definition be implemented in C++? It takes almost no effort to translate the definition into a recursive version, which is

```
unsigned long Fibonacci(unsigned long n)
{
  if (n < 2)
    return n;
  else
    return Fibonacci(n-2) + Fibonacci(n-1);
}
```

This function is easy to understand but extremely inefficient. To see, compute `Fibonacci(6)`, the seventh number of the sequence which is 8. Based on the definition, the computation requires

---

[4] Recursively to compute the value of $5!$ [FACTR] using our recursive algorithm, we'd need to multiply 5 times the value of $4!$, which requires multiplying 4 times the value of $3!$, and so on, down to $1!$, which is 1. Each of these steps requires a bit of storage on the stack, though. Even though the program is small, the computer still would have to store 5, 4, 3, 2, and 1 on the stack, which may be even more dangerous than your program is.

```
Fib(6) = Fib(5) + Fib(4)
       = Fib(4) + Fib(3) + Fib(3) + Fib(2)
       = Fib(3) + Fib(2) + Fib(2) + Fib(1) + Fib(2) + Fib(1) + Fib(0)
       = Fib(2) + Fib(1) + Fib(1) + Fib(0) + Fib(1) + Fib(0) + Fib(1) + Fib(0) + 1
       = Fib(1) + Fib(0) + 1 + 1 + 0 + 1 + 0 + 1 + 0 + 1
       = 1 + 1 + 1 + 0 + 1 + 0 + 1 + 0 + 1
       = 6
```

This is just the beginning of our calculation process, and even here there are many shortcuts. A more complete calculation can be expressed more concisely in the form of the table shown in Figure 5.8. The tremendous inefficiency results because `Fib(6)` is called to determine the seventh element of the Fibonacci sequence. The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, `Fib(3)` is called eight times with arguments of 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once. To find `Fib(6) = 8`, it computes `Fib(5)`, `Fib(4)`, `Fib(3)`, `Fib(2)`, `Fib(1)`, and `Fib(0)`. First, to determine these values, `Fib(4)`, ..., `Fib(0)` have to be computed to know the value of `Fib(5)`. Independently of this, the chain of computations `Fib(3)`, ..., `Fib(0)` is executed to find `Fib(4)`.

We can prove that the number of additions required to find `Fib(n)` using a recursive definition is equal to `Fib(n + 1) - 1`. Counting two calls per one addition plus the very first one means that `Fib(n)` is called `2 * Fib(n + 1) - 1` times to compute `Fib(n)`. This number can be exceedingly large for fairly small `n`, as the table in Figure 5.8 indicates.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number and nearly a million calls to determine the thirty-first. This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with `n`, the algorithm has to be abandoned except for very small numbers.

Figure 5.8 The tree of calls for `Fib(6)`

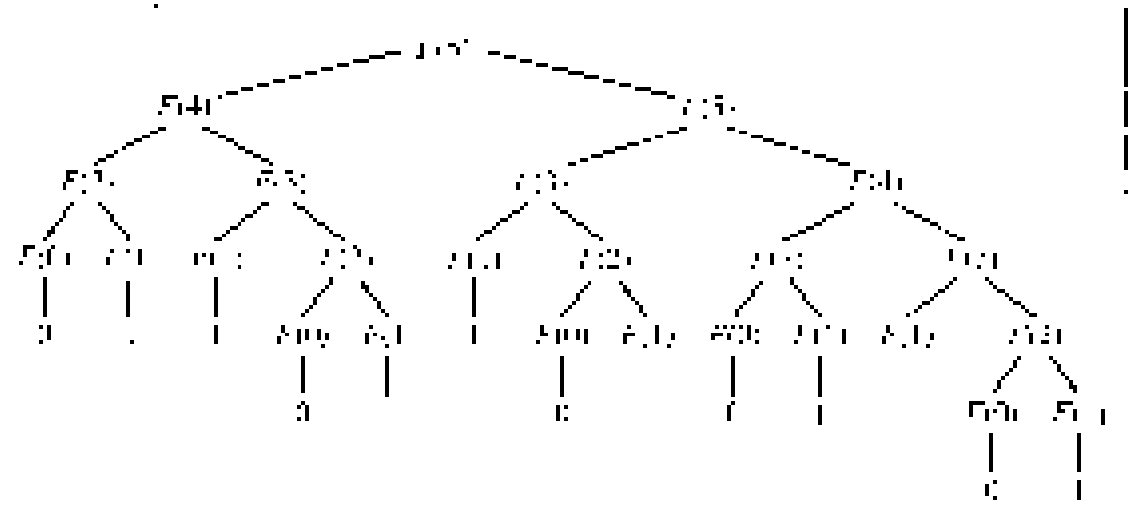


TABLE 5.9 Number of additions and number of recursive calls to count the Fibonacci numbers.

n	Fib(n)	Number of Additions	Number of Calls
5	5	4	9
10	55	54	109
15	610	609	1219
20	6765	6764	13529
25	75025	75024	150049
30	832040	832039	1664079

An iterative algorithm, implemented after exam, as follows:

```
unsigned int FastFib(unsigned int n)
{
    if (n < 2)
        return n;
    else {
        register unsigned int i = 2, temp, current = 1, last = 0;
        for (; i <= n; i++) {
            temp = current;
            current += last;
            last = temp;
        }
        return current;
    }
}
```

For $n = 1$, the `for` loop executes zero times, making these assignments per call. Each iteration executes the `for` loop body, the same as in Figure 5.13.

However, there is another, more efficient method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\mathrm{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Since $|\hat{\phi}| < 1$, $\hat{\phi}^n$ becomes smaller as n increases. For sufficiently large n, $\hat{\phi}^n$ is so small that it can be ignored and

$$\mathrm{Fib}(n) = \frac{\phi^n}{\sqrt{5}}$$

TABLE 5.10 Comparison of iterative and recursive algorithms for calculating Fibonacci numbers

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	7	12	25
10	9	22	177
15	14	37	1973
20	19	52	21891
25	24	67	242785
30	29	82	2692537

[illegible] approximately the same number of additions for the iterative and recursive algorithms. However, the number of assignments is quite different. The iterative algorithm requires only a few assignments, whereas the number of assignments for the recursive algorithm grows rapidly. In fact, the number of assignments for the recursive algorithm is exactly the same as the number of additions. The reason for this is that each addition is preceded by an assignment. The number of assignments for the recursive algorithm is

$$\text{assignments} = \text{additions} = \text{fib}(n+1) + \text{fib}(n) - 1 = \text{fib}(n+2) - 1 = \text{fib}(n+2) - 1 = 1.618^{n+2} - 1$$

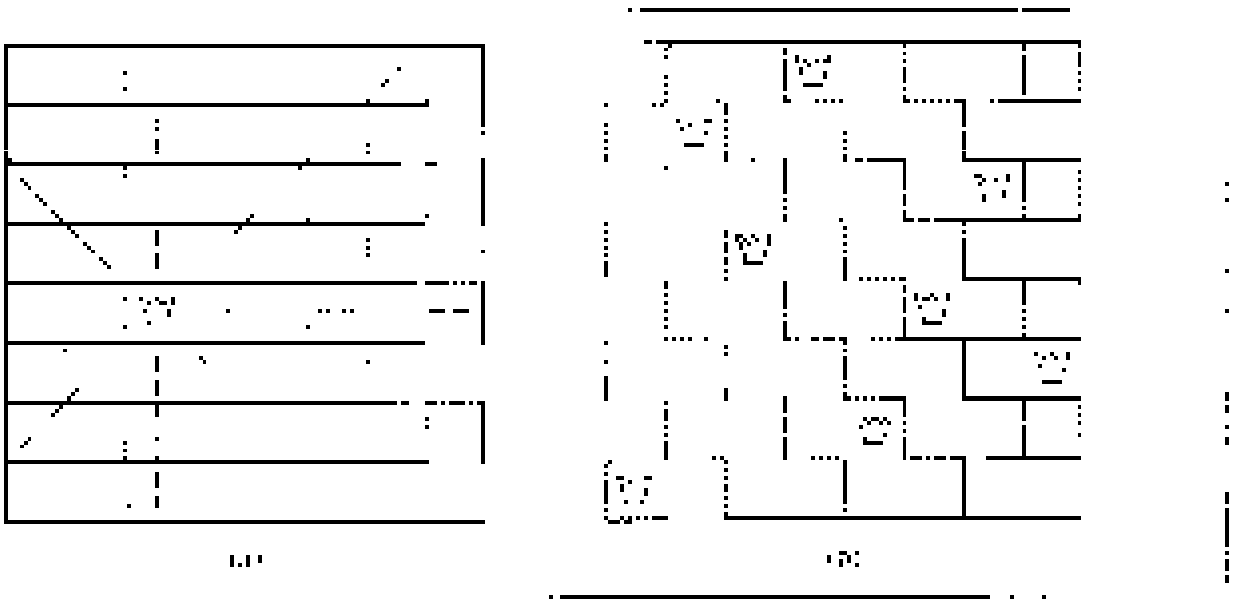
Try to prove by induction that this formula is correct.

5.9 BACKTRACKING

In solving some problems, a situation arises in which we have a choice between several possibilities. We don't know which choice to make, but after we make a choice, we can tell whether it was a wrong choice. If it was a wrong choice, we backtrack to the point where we made the choice and try another. If we have tried all possibilities and none has led to a solution, we backtrack to the previous choice point and try another. Eventually we either find a solution or backtrack to the first choice point and find that all possibilities from that point have been exhausted. Backtracking algorithms are often recursive because backtracking to a choice point is similar to backtracking from a dead end.

The eight queens problem illustrates backtracking. The problem is to place eight queens on a chessboard so that no queen can take another piece. That is, no two queens can be in the same row, column, or diagonal as the queen (see Figure 5.11). To solve this problem, we start with the first column and place a queen in the first row. Then we move to the second column and place a queen in the first row where she is not in conflict with the first queen. If we find a row in the second column where the queen is not in conflict with the first queen, we move on to all the remaining columns, placing a queen in the first row that does not conflict with any queen already on the board. Figure 5.12 illustrates this algorithm. Figure 5.12(a) shows the first queen placed in the upper left corner of the board. Figure 5.12(b) shows the second queen placed in the third row of the second column. Figure 5.12(c) shows the third queen placed in the fifth row of the third column. Figure 5.12(d) shows that there is no row in the fourth column where a queen can be placed. We backtrack to the third column and move the third queen to the sixth row, as shown in Figure 5.12(e). Figure 5.12(f) shows the fourth queen placed in the second row of the fourth column. Figure 5.12(g) shows that there is no row in the fifth column where a queen can be placed. We backtrack to the fourth column and move the fourth queen to the third row, as shown in Figure 5.12(h). Figure 5.12(i) shows the fifth queen placed in the second row of the fifth column. Figure 5.12(j) shows that there is no row in the sixth column where a queen can be placed. We backtrack to the fifth column and move the fifth queen to the fourth row, as shown in Figure 5.12(k). Figure 5.12(l) shows the sixth queen placed in the second row of the sixth column. Figure 5.12(m) shows that there is no row in the seventh column where a queen can be placed. We backtrack to the sixth column and move the sixth queen to the fifth row, as shown in Figure 5.12(n). Figure 5.12(o) shows the seventh queen placed in the third row of the seventh column. Figure 5.12(p) shows that there is no row in the eighth column where a queen can be placed. We backtrack to the seventh column and move the seventh queen to the fourth row, as shown in Figure 5.12(q). Figure 5.12(r) shows the eighth queen placed in the second row of the eighth column. Figure 5.12(s) shows the solution to the eight queens problem.

FIGURE 5.11 The eight queens problem



(a) (b)

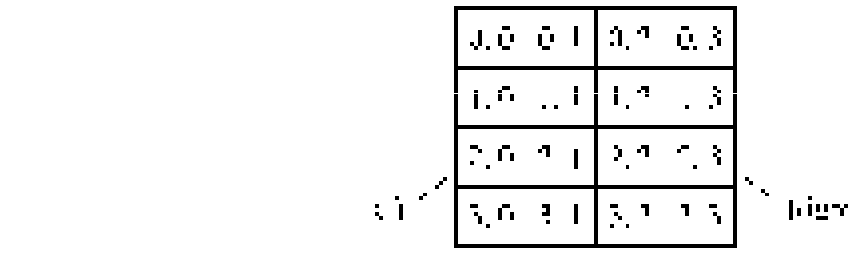
conflicting position. We then move another queen back for the fifth queen and try again with the sixth. If no solution can be found, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and the process is started. This process requires a great deal of effort, most of which is spent backtracking to previously considered but ultimately unsuccessful avenues. Because of this, a process that proceeds in this manner is a direct application of recursion and is a natural application for backtracking. Pseudocode for this backtracking algorithm is as follows (the exit in procedure backtracking):

```
put8queens(now)
    try every position on the chessboard
      if position is available
          remember this queen's position and;
          if row < 8
            put8queens(row + 1)
          else success
          remove the queen from position and;
```

This algorithm tries all possible positions for the eight queens. Is this the best that can be said of this algorithm?

The most natural approach for implementing this algorithm is to declare an 8 × 8 array board of boolean components to represent the chessboard. The array is initialized to false, and each time a queen is put in a position (i, j), board[i][j] is set to true. Also, a function must be written to determine whether a position (i, j) is under attack and to set both diagonals that cross at that position (i, j). When backtracking, the same positions (that is, positions in corresponding rows, columns, and diagonals) have to be set back to false again. Under the worst circumstances, hundreds of queens will need

Figure 5.12 4 × 4 chessboard.



available positions for queens. The scoring and resetting process is the most time-consuming part of the implementation. For each queen, between 22 and 28 positions have to be scored and reset. This involves calculations between 7 and 13 diagonals.

In this approach, the board is viewed from the perspective of the player who sees the entire board along with all the queens at the same time. However, if we focus solely on the queens, we can consider the chessboard from their perspective. For the queens, the board is not divided into squares, but into rows, columns, and diagonals. Each queen is placed on a single square, it resides not only on this square, but on the entire row, column, and diagonal. Besides, these lines have reciprocity properties. A different data structure can be utilized to represent this.

To simplify the problem for the first solution we use a 4 × 4 chessboard instead of the regular 8 × 8 board. Later we can make the necessary changes for this program to accommodate a regular board.

Figure 5.12 contains the 4 × 4 chessboard. Notice that indexes in the left-to-right indexed left diagonals add up to the same number (row + col = k); this number is associated with this diagonal. There are seven left diagonals, 0 through 6. Indexes in the left-to-right indexed right diagonals have the same difference (row − col = k), and this number is unique among all right diagonals. Therefore, right diagonals are assigned numbers −3 through 3. The data structure used for marking diagonals is dependent on how it is denoted by numbers 0 through 6. For right diagonals, it is used as an array, but it cannot be indexed by negative numbers. Therefore, it is an array of seven cells, but to account for negative values obtained from the formula row − col, the same number is always added to it so as not to cross the boundaries of this array.

An analogous array is also needed for columns, but now rows need not be traversed along with the queens (we have already accomplished this in row 5). Figure 5.13 contains the code to implement these arrays. The program is short due to recursion, which hides some of the goings on from the user's sight.

Figures 5.14 through 5.17 document the steps taken by backtracking to place four queens on the chessboard. Figure 5.14 contains the move number, queen number, row, and column number for each attempt to place a queen. Figure 5.15 shows the changes in the arrays positiveDiag, column, leftDiagonal, and rightDiagonal. Figure 5.16 shows the changes in the run-time stack during the

Figure 5.13 Eight queens problem implementation.

```
class ChessBoard {
public:
    ChessBoard();     // 8 x 8 chessboard
    ChessBoard(int);  // n x n chessboard
    void SolveIt();
private:
    const bool available;
    const int rowSize, rowMax;
    bool *colAvail, *diagAvail, *revDiagAvail;
    int *queenLocation, boxNum;
    void putQueen(int);
    void printBoard();
    void initialize();
};

ChessBoard::ChessBoard() : available(true), rowSize(8), rowMax(8-1)
{
    initialize();
}
ChessBoard::ChessBoard(int n) : available(true), rowSize(n),
rowMax(rowSize-1) {
    initialize();
}
void ChessBoard::initialize() {
    register int i;
    column = new bool[rowSize];
    queenLocation = new int[rowSize];
    diagAvail = new bool[rowSize*2 - 1];
    revDiagAvail = new bool[rowSize*2 - 1];
    for (i = 0; i < rowSize; i++)
        queenLocation[i] = -1;
    for (i = 0; i < rowSize; i++)
        column[i] = available;
    for (i = 0; i < rowSize*2 - 1; i++)
        diagAvail[i] = revDiagAvail[i] = available;
    boxNum = 0;
}
void ChessBoard::putQueen(int row) {
    for (int col = 0; col < rowSize; col++)
```

FIGURE 5.13 (continued)

```
        if (column[col] == available &&
            leftDiagonal[row+col]    == available &&
            rightDiagonal[row-col+norm]  == available) {
            positionInRow[row] = col;
            column[col] = !available;
            leftDiagonal[row+col]   = !available;
            rightDiagonal[row-col+norm] = !available;
            if (row < squares-1)
                putQueen(row+1);
            else printBoard();
            column[col]   = available;
            leftDiagonal[row+col]   = available;
            rightDiagonal[row-col+norm] = available;
        }
    }
    private void findSolutions() {
        putQueen(0);
        cout << howMany << " solutions found.\n";
    }
```

right sides. All the spaces in the matrix are initialized to true by the statement in the for loop, which makes way for the initial invocation of putQueen(). Each successive recursive call to putQueen() will result in a row and col. Figure 5.13 illustrates the changes to the chessboard. A detailed description of each step follows.

1. We start by trying to put the first queen, the one in upper left corner (0,0). Since this is the very first move, the condition of the if statement is certain to be true. After the queen is placed, the diagonal to the upper right slope is marked by setting rightDiagonal[0] to false. Figure 5.13, (1) shows the result of these assignments.

2. Since row < 8, putQueen() calls itself with row+1, but before its execution, an activation record is created on the run time stack (see Figure 5.11c). Now we check the availability of a field on the second row (i.e., row = 1). For col = 0, column[0] is queried. For col = 1, main right diagonal is checked, and for col = 2, all three parts of the if statement condition are true. Therefore the second queen is placed in position (1,2), and this fact is immediately reflected in the private data arrays. Again, row < 8, so putQueen() is called trying to locate the third queen in row 2. After all the positions in this row, 0 through 7, are checked, no available position is found, the for loop is exited without executing the body of the if statement, and this call to

FIGURE 5.14 Single-stepping the first recursive configuration of four queens as traced by the function putQueen().

Move	Queen	row	col	
[1]	1	1	1	
[2]	2	2	3	illegal
[3]	1	2	2	
[4]	2	3	4	illegal
[5]		2		
[6]	2		4	
[7]	3	3	1	
[8]	4	4	3	

FIGURE 5.15 Changes in the four arrays used by function putQueen().

position number	column	leftDiagonal	rightDiagonal	row
[0], [1], ...	[0], [1], [2], [3]	[0], [1], [2], [3], [4], [5], [6]	[-3], [-2], [-1], [0], [1], [2], [3]	0, 1
[1] [2]	[1] [3]	[2] [3]	[-1] [1]	[2] [3]
[2], [3], ...	[0], [1], [2], [3]	[0], [1], [2], [3], [4], [5], [6]	[-3], [-2], [-1], [0], [1], [2], [3]	2, 3
[3] [2] [1]	[1] [3] [0] [2]	[3] [4] [2] [3]	[-2] [0] [-1] [1]	[2] [3] [1]
[1], [2], [3], [4]	[1], [3], [0], [2]	[3], [4], [2], [3], [1], [2], [0]	[-2], [0], [-1], [1], [-3], [-1], [2]	0, 1, 2, 3
[2] [0] [3] [1]	[2] [0] [3] [1]	[4] [2] [5] [3]	[-1] [1] [-2] [0]	[3] [0] [2] [1]

putQueen() is now done. But it will be executed by putQueen() after the second row is tried in the call of a new activation.

[3] Values of col and row are stored, and the execution of the second call of putQueen() continues by restoring some fields in the new activation record, and since col = 2, the for loop can continue executing. The test in the if statement allows the second queen to be placed on the board, this time in position (1, 2).

[4] After each putQueen() is called (push a new activation record on the run-time stack), and after the test in the putQueen() for loop succeeds, a recursive call is made. (See Figure 5.16.) No calls are made to call from step [3] is returned,

FIGURE 5.16 Changes to the run-time stack during the first successful completion of putQueen(1).

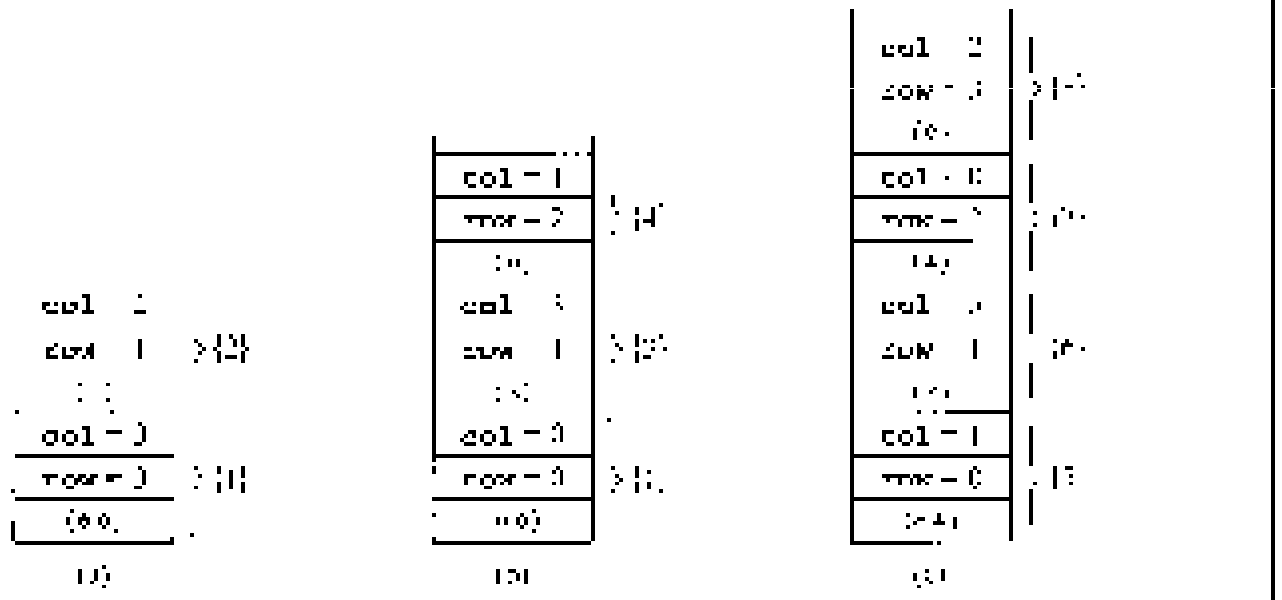
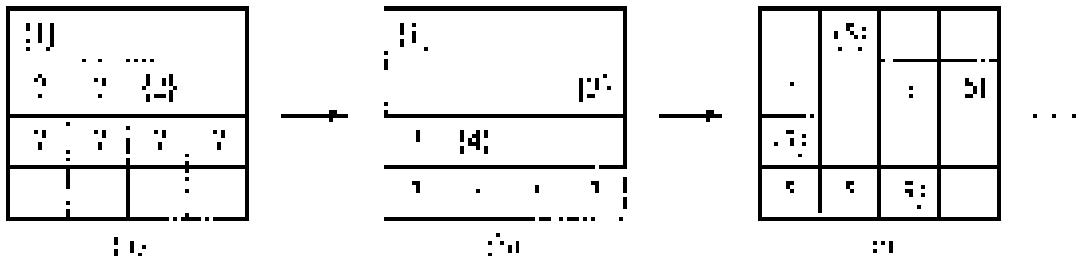


FIGURE 5.17 Changes to the chessboard heading for the first successful configuration.



...of the third queen is attempted, but no position can be found for it. At the same time, each invocation of the for loop is finished.

(5) As a result, the first call of putQueen(1) resumes execution by placing the first queen in position (1, 2).

(6–8) This time, each invocation of the for loop finds a complete solution.

Figure 5.18 contains a trace of the execution up to the first successful placement of the queens on the chessboard.

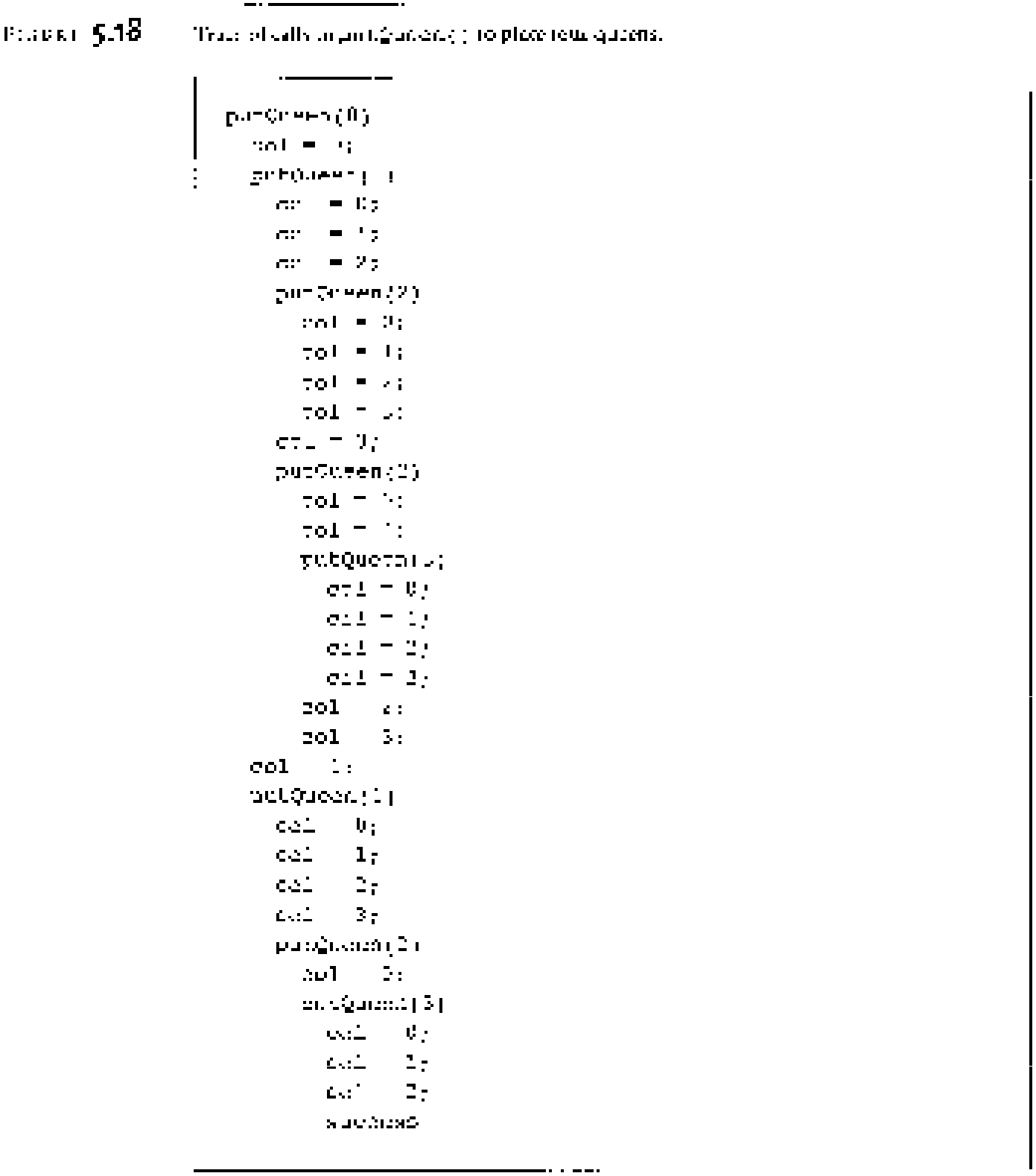


FIGURE 5.18 Trace of call to putQueen(0) to place four queens.

5.10 Concluding Remarks

After having studied all these examples (and one more to come), what can be said about recursion as a programming tool? Like any other tool in data structures, it should be used with good judgment. There are no general rules for when to use it and when not to use it. Each particular problem decides. Recursion is usually less efficient than its iterative equivalent. But if a recursive program takes .01 milliseconds (ms) for execution, for example, and the iterative version only 10 ms, then although the latter is ten times faster, the difference is hardly perceivable. If there is an advantage in the clarity, readability, and simplicity of the recursive solution, then the difference in the execution time between these two versions can be disregarded. Recursion is often simpler than the iterative solution, and more often than not will be right on the original algorithm. The factorial and power functions are such examples, and we will see more interesting cases in the chapters to follow.

Although every recursive procedure can be converted into an iterative version, the conversion is not always a trivial task. In particular, it may involve explicitly manipulating a stack. That's where the time-space trade-off comes into play. Using iteration often necessitates the introduction of new data structures to implement a stack, whereas recursion relieves the programmer of this task by handing it over to the system. One way or the other, if normal recursion is avoided, very often a stack has to be maintained by the programmer or by the system. But the programmer decides who carries the load.

Two situations can be presented in which a nonrecursive implementation is preferable even if recursion is more natural or simpler. First, iteration should be used in the so-called real-time systems where an immediate response is vital for proper functioning of the program. For example, in military environments, in the space station, or in certain types of scientific experiments, it may matter whether the response time is 10 ms or 100 ms. Second, the programmer is encouraged to avoid recursion in programs that are executed hundreds of times. The best example of this kind of program is a compiler.

But these remarks should not be treated too stringently, because sometimes the recursive version is better than a nonrecursive implementation. Hardware may have built-in stack operations that considerably speed up functions operating on the run-time stack, such as recursive functions. Running a simple routine implemented recursively and iteratively and comparing the two can help us make a decision. Recursion is advisable when we know that recursion can execute for only a few terminations. Such a test is especially important if tail recursion comes into play. However, when a stack cannot be eliminated from the iterative version, the use of recursion is usually recommended, since the execution time for both versions does not differ substantially—certainly not by a factor of 10.

Recursion should be eliminated if some part of the work is unnecessarily repeated, as in our naive Fibonacci example. There are, however, some computations for which a similar structure is found that the ease of using recursion can sometimes be deceptive, and this is where recursion can be applied for only a few times. Inversions and merge sort are examples. Whether a recursive implementation leads to unnecessary repetition may not be immediately apparent, but often drawing a tree of calls similar to Figure 5.9 can be very helpful. If it shows that a subproblem is called many times with the same

argument. A recursive function for computing a factorial function is reduced to a linked list with no restrictions on it. Such a tree is very deep—that is, it has many levels—but the program won't be vulnerable to run-time stack overflow. A tree that is shallow and bushy, with many nodes on the same level, does not seem to be a good approach—only if the number of operations is very moderate.

5.11 Case Study: A Recursive Descent Interpreter

All programs we have run so far on computers have been translated into machine language and executed directly by the computer hardware. However, this is not always necessary. Alternatively, we can write a program that reads another program and executes each statement in it. This technique requires a program called an **interpreter** that reads the source program and executes it directly, without translating it into machine language. Interpreters usually execute programs more slowly than translators do. However, because interpreters do not require the translation step, they usually start executing the source program more quickly. Interpreters are often used in situations where execution speed is not crucial, such as in command interpreters for operating systems like UNIX or DOS.

Writing an interpreter is by no means a trivial task. As an example, in this section we study a simple interpreter for a limited programming language. Our language, which we call SIMPLE, consists only of assignment statements and if statements. There are no declarations, loops, functions, etc. But in this limited language we can illustrate the principal ideas used in compilers and interpreters for more powerful languages such as C++.

Our program is an interpreter in that it not only checks whether the assignment statements are syntactically correct, but also executes the assignments.

The program is to work in the following way. If we enter the assignment statements

```
count = 5;
total = count;
total = (4/2.0 - 1) * total + count;
```

then the system should prompt for the value of each variable as it is encountered in the statement. For example,

```
print: count
```

the system should respond by printing

```
count = 999
```

Evaluation of all variables stored so far can be performed by entering

```
status
```

and the following values should be printed for our example:

```
var3    223
var2    3
var1    3
```

All current values are to be stored on file `data` and updated if necessary. Thus, if

```
var2    var2 + 1;
```

is entered, then

```
print var2
```

should return

```
var2 = 25
```

The interpreter prints a message if any undefined variable is used in an assignment and expressions do not conform to the required syntax (such as unmatched parentheses).

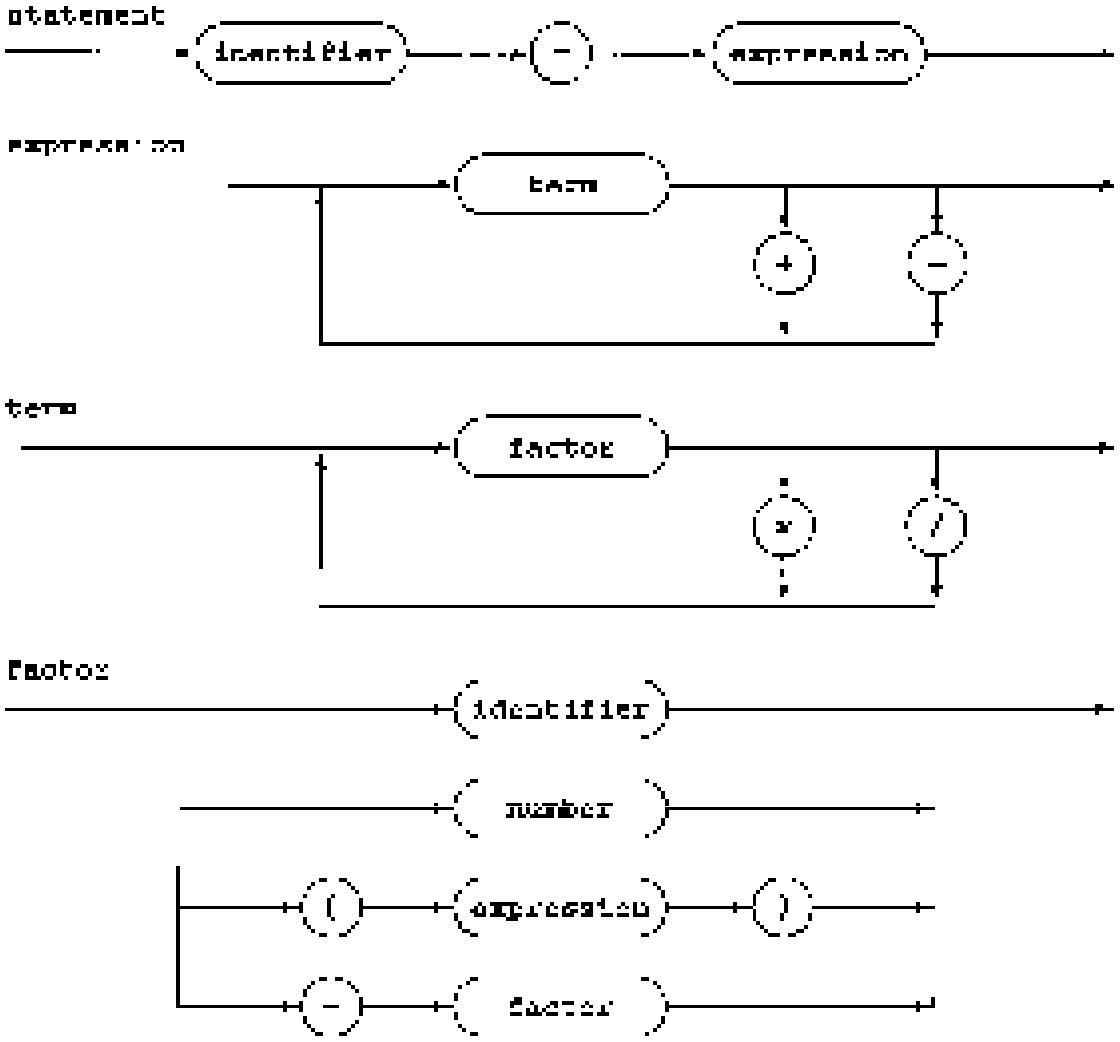
The program can be written in a style that is not dependent on the number of variables. We choose a method known as *recursive descent*. This consists of several mutually recursive functions (recursive, as they appear in Fig. 3.19).

Each diagram shows a syntax graph, or part of it. For example, to parse a `factor` we follow the arrow labeled `factor` until we reach a circle. If the circle contains a symbol (`'('` and `')'` in this case), we must find that symbol in the input. If the circle contains a word, we call the function with that name. Thus, to parse a `factor` we look for a left parenthesis, an `expression`, and a right parenthesis, or we look for a `number` or a `variable`. Parsing a `term` is similar to parsing a `factor`, except that we look for `factor`s instead of `term`s. To parse an `expression`, we look for `term`s instead of `factor`s.

The diagram in Figure 3.19 is slightly more complicated than the others. It shows that an `expression` is a `term` optionally followed by a `'+'` or `'-'` and another `term`, optionally followed by a `'+'` or `'-'` and another `term`, and so on. Thus, to parse an `expression`, we look for a `term` and then zero or more occurrences of `'+'` or `'-'` followed by a `term`. Parsing a `term` is similar to parsing an `expression`, except that we look for `factor`s instead of `term`s.

How can the recursive descent interpreter be implemented? The simplest approach is to write a function for each diagram in Figure 3.19. For instance, `term()` is a function returning a double number. This function always reads a `factor` and then looks at the next character to see if it is either `'*'` or `'/'`. If not, `term()` returns the `factor`. Otherwise, it calls `factor()` again. Then it multiplies the value already accumulated for `term` by the value returned by the second call to `factor` or divides it by the value returned by the second call to `factor`. `term()` then looks for more `factor`s. Each call to `factor` can initiate another call to `factor` indirectly through

FIGURE 5.19 Diagrams of functions used by the recursive descent interpreter.



the call `term()` in `factor()` or `expression()` in `term()`. Similarly, the function `term()` looks like the following:

```
term()
    f1 = factor();
    while (token is * or /)
        f2 = factor();
        f1 = f1 * f2 or f1 / f2;
    return f1;
```

In addition, `expression()` is very similar to `term()`, and the function `factor()` is

```
    }
    return value;
}
```

We have nearly examined the class's global variable that is used for scanning an input character by character.

However, in the meantime, we assume that the only valid statements are entered for evaluation. What happens if a mistake is made, such as entering two equal signs, misspelling a variable name, or forgetting a semicolon? In the next project, you'll simply discuss and attempt some of these mistakes before we continue the complete code that we've developed.

FIGURE 5.20 Implementation of multiple language interpreters

```
// ... interpreters ...

#include <iostream>
#include <map>

#include <cctype>
#include <cmath>
#include <cstdlib> // for exit()

using namespace std;

class Calculator {
public:
    Calculator(char *s = "", double v = 0) {
        id = new char[strlen(s) + 1];
        strcpy(id, s);
        value = v;
    }
```

Figure 5.20 (continued)

```
    bool operator== (const Idnode& node) const {
        return strcmp(id,node.id) == 0;
    }
    bool operator!= (const Idnode& node) const {
        return !(*this == node);
    }
    bool operator< (const Idnode& node) const {
        return strcmp(id,node.id) < 0;
    }
    bool operator> (const Idnode& node) const {
        return strcmp(id,node.id) > 0;
    }
private:
    char *id;
    double value;
    friend class Statement;
    friend ostream& operator<< (ostream&, const Idnode&);
};

class Statement {
public:
    Statement() {
    }
    void setStatement();
private:
    Idnode *node;
    char  ch;
    double Factor();
    double Term();
    double Expression();
    void readid(char);
    void lookup(char *id) {
        Idnode key(id); node=
    }
    double findValue(char *);
    void  putvalue(Idnode*, double);
    friend ostream& operator<< (ostream&, Statement&);
};

#end
```

Figure 5.20 (continued)

```
//////////////////////////////////////////////////////////////////////////
// Statement.cpp
//////////////////////////////////////////////////////////////////////////

#include "Interpreter.h"

double Statement::doEvaluation() {
    IdNode tmpId;
    list<IdNode>::iterator i = find(idList.begin(),idList.end(),tmpId);
    if (i != idList.end())
        return i->value;
    else cannotProceed("Unknown variable");
    return 0;  // this statement will never be reached
}

void Statement::processAssignment(const Id &id,double v) {
    IdNode tmpId(id,v);
    list<IdNode>::iterator i = find(idList.begin(),idList.end(),tmpId);
    if (i != idList.end())
        i->value = v;
    else idList.push_front(tmpId);
}

// readId() reads a string of letters and digits that starts with
// a letter, and stores that in array passed to it as an actual
// parameter.
// Examples of identifiers are: var , x, pqr12xyz, aMath, etc.

void Statement::readId(char *id) {
    int i = 0;
    if (isspace(ch))
        nextCh();       // skip a white;
    if (isalpha(ch)) {
        while (isalnum(ch)) {
            id[i++] = ch;
            nextCh(); // don't skip blanks;
        }
        id[i] = '\0';
    }
    else invokeError("Identifier expected");
}
```

Listing 5.20 (continued)

```
double Statement::factor() {
    double var, minus = 1.0;
    string str[200];
    char ch;
    while (ch == '+' || ch == '-') {          // read all + and - signs.
        if (ch == '-')
            minus *= -1.0;
        cin >> ch;
    }
    if (isalpha(ch) || ch == '.') {           // factor can be a number
        cin.putback(ch);                      // or a variable.
        cin >> var >> ch;
    }
    else if (ch == '(') {                     // or a parenthesized
                                              // expression.
        var = expression();
        if (ch == ')')
            cin >> ch;
        else IOSInitError("Ack! paren left out");
    }
    else {
        readIn(id);                           // or an identifier.
        if (isspace(ch))
            cin >> ch;
        var = findValue(id);
    }
    return minus * var;
}
```

```
double
Statement::term() {
    double f = factor();
    while (true) {
        switch (ch) {
            case '*' : f *= factor(); break;
            case '/' : f /= factor(); break;
            default  : return f;
        }
    }
}
```

(continued)

Program 5.20 (continued)

```
double StatementInterpreter::expression() {
    double x = term();
    while (true) {
        switch (ch) {
            case '+': ch = getch(); x += term(); break;
            case '-': ch = getch(); x -= term(); break;
            default: return x;
        }
    }
}

void StatementInterpreter::executeStatement() {
    char id[20], command[20];
    double x;
    cout << "Enter a statement: ";
    cin >> ch;
    readId(id);
    strcpy(stepCommand,id);
    if (strcmp(command,"print") == 0) {
        cout << value;
    else if (strcmp(command,"read") == 0) {
        readId(id);
        cout << id << " = " << findValue(id) << endl;
    }
    else if (strcmp(command,"set") == 0)
        exit(0);
    else {
        if (ch != '=')
            cin >> ch;
        if (ch == '=') {
            x = expression();
            if (ch != ';')
                throwError("There are more tokens in the statement");
            else processCode(id,x);
        }
        else throwError("'=' is missing");
    }
}

ostream &operator<< (ostream &out, const Instruction &i) {
```

FIGURE 5.20 main.cpp

```
    //InterfaceNodeVector::reverse_i = v.rbegin();
    for ( ; i != v.rend(); i++)
        out << *i;
    out << endl;
    return out;
}

ostream& operator<<(ostream& out, const TcNode& n) {
    out << n.kind() << " = " << n.value() << endl;
    return out;
}

//////////////////////// useInterpreter.cpp ////////////////////////

#include "Interpreter.h"

using namespace std;

void main() {
    Statement statement;
    cout << "The program processes statements of the following form:"
         << "\nnode = <expression>\nprint <node>\nstatements\nexecution";
    while (true)                    // This routine  can be broken by exit()
        statement.getStatement();   // in getStatement() or upon hitting an
                                    // error.
}
```

5.12 EXERCISES

1. [redacted]

2. [redacted]

3. What is the output of the following version of reverse()?

```
void reverse() {
  int c;
  cin.get(c);
  if (c != '\n')
      reverse();
  cout.put(c);
}
```

4. What is the output of the same function if c is declared as

```
static char c;
```

5. An early application of recursion can be found in the mathematical theory of John Napier's method of finding logarithms. The method was as follows:

```
Given two numbers a1 and a2 and their logarithms log1 and log2, where log1 is the
known,
while not done
    find a geometric mean of a1 and a2 and its arithmetic mean log3, where log3 is
    the arithmetic mean of the earlier logarithms log1 and log2
    if (log3 is sufficiently close to log1)
    process the pair a2 and log3 else process the pair a1 and log3;
```

For example, the 10-based logarithms of 10 and 1000 are numbers 1 and 3. The geometric mean of 10 and 1000 is 100, and the arithmetic mean of their logarithms 1 and 3 is 2. Thus, the logarithm of 100 is equal to 2. The process can be continued. The geometric mean of 100 and 1000 is about 316, whose logarithm is equal to 2.5. The process can be continued.

a. Write a recursive function logarithm() that computes the logarithm of a number between 10 and 1000. Use approximation 1 if your result is within 0.01 of the correct value. Otherwise, use approximation 2.

b. Modify this function so that it computes the logarithm of a specific number between 10 and 1000. Also provide a function that finds the logarithm of any positive number. Hint: Keep dividing by 10 until the result is between 10 and 1000.

c. Add a function call to the main() function that prints the logarithm of 1, 10, 100, 1000, and 10000. Explain why the logarithm function is not called recursively for some of these values.

6. Two algorithms for both versions of the power function are given in this chapter. A third algorithm is based on the observation that $x^n = (x^{n/2})^2$ if n is even and $x^n = x(x^{(n-1)/2})^2$ if n is odd. For example, $x^{13} = x(x^6)^2 = x(x(x^3)^2)^2 = x(x(x(x)^2)^2)^2$. Only four multiplications are needed to find the value of x^{13}. Using this observation, implement both algorithms for computing x^n. Hint: A special case is needed for odd exponents.

7. Execute the bound box functions test1() and test2() for the parameter values 0.1, 0.2, and 0.3. Explain why these functions are given in Section 5.6.

8. Write a function that converts a string of numeric digits into an integer. For instance, `convert("2345")` would return the integer 2345.

9. Write a recursive function that computes the binomial coefficient, defined as

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$$

10. Write a recursive function that computes the first terms of the series

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \cdots$$

11. Write a recursive function that computes the greatest common divisor of two integers, using the following definition:

$$GCD(m,n) = \begin{cases} m & \text{if } n \text{ is evenly divisible by } m \\ GCD(n, m \bmod n) & \text{if } n > m \\ GCD(\text{the smaller number}, \text{the larger number}) & \text{otherwise} \end{cases}$$

12. Give a recursive version of the following function:

```
void reverseLines()
{
  for (int i = 1; i <= 10; i++)
    cout << readLine() << endl;
}
```

13. Consider a recursive class that represents a word. Its private data consists of:

   a. a word

   b. a student ID, ignoring all the owner's other personal information. (Assume that no two students have the same ID.)

14. Implement the following functions recursively, without using any loops or local variables:

   a. Check if a string is a palindrome.

   b. Count all occurrences of a character in a string.

   c. Remove all occurrences of a character from a string.

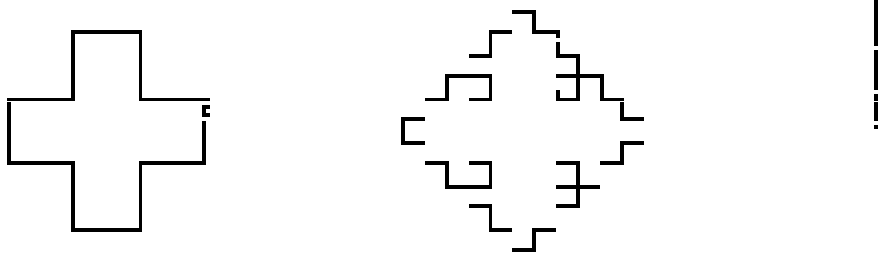
15. Write equivalents of the last three functions from Exercise 14 iteratively.

16. What changes would you have to make in the `Scanner` class to support a `hasNextInt` method? Would you have to add any private helper methods to the class?

17. Create a graphical display of the function $\frac{x}{15}$ that calls itself recursively for values of x greater than 15.

18. Write a recursive function that accepts a file name as a parameter and prints to the console all of the lines in the file that contain a semicolon (;) character.

Figure 5.21 Figures to be drawn with modified program in Figure 5.20



19. The recursive Fibonacci function of Section 5.5 is very inefficient because it computes the same Fibonacci numbers many times. For example, to compute fib(6), the function computes fib(5) and fib(4). But to compute fib(5), it also computes fib(4). Write a program that counts how often the Fibonacci function computes each value. Then explain why the program takes so long to compute fib(30). Can you improve the efficiency?

20. Write a recursive function `void print_digits(int n)` that prints the digits of an integer. For example, `print_digits(1729)` should print `1 7 2 9`. Then write a program that prompts the user for an integer and calls your function.

21. Write the function prototype of the function in Exercise 20 as a C++ function.

22. Suppose the following program is executed. Trace the execution and explain what it does.

```
int f(int n)
{
   if (n == 0) return 1;
   else return n * f(n - 1);
}

int g(int n)
{
   if (n == 0) return 1;
   else return n * g(n - 1) + g(n - 1);
}
```

Indicate clearly which functions are called in which stage of parsing these sentences.

23. Extend our interpreter so that it can also process exponentiation, ^. Remember that exponentiation has precedence over all other operations so that `2 * 3 ^ 4 + 5` is the same as `2 * (3 ^ 4) + 5`. Notice also that exponentiation is a right associative operator, unlike addition or multiplication, so that `2 ^ 3 ^ 4` is the same as `2 ^ (3 ^ 4)` and not as `(2 ^ 3) ^ 4`.

24. In C++, the division operator, `/`, returns an integer result when it is applied to two integers. For instance, `1729 / 100` equals `17`. However, most users interpret the result as `17.29`. Modify the interpreter so that division works the same way as in C++.

25. Our interpreter is unforgiving when a mistake is made by the user, since it finishes execution as soon as a problem is detected. For example, when the name of a variable is mistyped after requesting its value, the program notifies the user of the error and destroys the list of identifiers. Modify the program so that it continues execution after finding an error.

26. Write a recursive program to print the Towers of Hanoi.

5.13 Programming Assignments

1. Given n pairs of data points (x_i, y_i), the covariance of x and y is defined by

$$c = s_x s_y$$

where s_x is the standard deviation of x and s_y is the standard deviation of y. The standard deviation s_x is defined by

$$s_x = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

and the mean $\bar{x}$ by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Write a program that reads a set of data points and computes the standard deviation using both versions of the mean and variance. Run your program for $n = 1000$, 10000, 100000, and 1000000 and compare the running times.

2. Write a program to implement the differentiation rules in the following table.

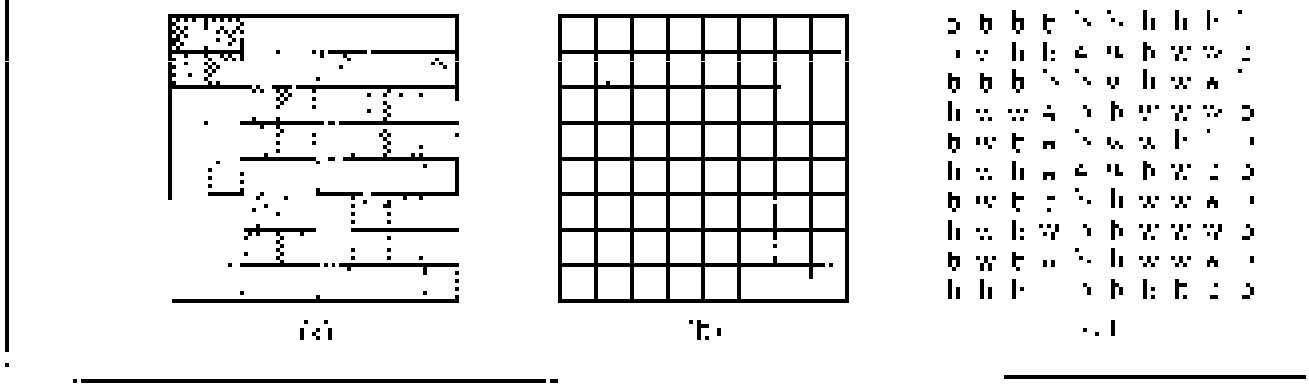
Rule 1: $(cf)' = cf'$	c is a constant
Rule 2: $(f \pm g)' = f' \pm g'$	
Rule 3: $(fg)' = f'g + fg'$	
Rule 4: $(f^n)' = nf^{n-1}f'$	n is a positive integer

As a simple application of these rules, find the derivative of $f(x) = 3x^2 + 5x + 1$ with respect to x.

$$\begin{aligned}
f'(x) &= \frac{d}{dx}(3x^2) + \frac{d}{dx}(5x) + \frac{d}{dx}(1) && \text{by Rule 2}\\
&= 3\frac{d}{dx}(x^2) + 5\frac{d}{dx}(x) + 0 && \text{by Rule 1}\\
&= 3(2x^1)\frac{dx}{dx} + 5(1)x^0 + 0 && \text{by Rule 3}\\
&= 6x^1 + 5x^0 && \text{by Rule 4}\\
&= 6x + 5 && \text{by Rule 1}
\end{aligned}$$

First, run your program for polynomials only, and then add formulas for derivatives for trigonometric functions, logarithms, etc. and extend the range of functions handled by your program.

FIGURE 5.22 (a) Two boxes of squares of blocks and (b) the cells and (c) the $2^n \times 2^n$ array implementing square (b).



3. An $n \times n$ square consists of n^2 unit cells arranged in an $n \times n$ array. The problem is to count the number of squares and the number of cells in each square. For example, Figure 5.22(a) consists of one square of size 8, one square of size 7, four squares of size 6, nine squares of size 5, and so on, for a total of 204 squares and 1,496 cells. Write a program that, for a given $n \times n$ square, outputs the number of squares and their sizes. Design your program so that it can process all properly marked cells. You will find it useful to write a function that processes a frame of black cells surrounding the enclosed square to simplify your implementation. For instance, the square in Figure 5.22(b) is stored as the square in Figure 5.22(c).

   Traverse the square row by row and, for the first unvisited cell encountered, invoke a function that processes one square. The function is invoking four recursive calls in this function to search for the adjacent cells and marking it with a special symbol as visited (counted).

4. Write a program that pretty-prints C++ programs, that is, a filter for printing programs with consistent use of indentation. Insert a number of spaces between tokens such as keywords, proper braces, operators, and so on, and a number of blank lines between blocks of code as specified in the input. Delete unnecessary spaces with the exception of those separating identifiers and keywords. Replace multiple-line comments with line comments. Delete empty statements and so on. The program takes as input a C++ file and produces an identical file according to the rules established for the pretty-printing program. Consider the code

```
if (n == 1) { n = 2 * n;
if (n < 10)
f(n,m-1); else f(n-2,m); } else n = 3 * n;
```

   should be transformed into

```
if (n == 1) {
    n = 2 * n;
    if (n < 10)
        f(n,m-1);
```

```
        else if (n == 1)
    }
    else n = 3 * n;
```

5. An excellent example of a program that can be greatly simplified by the use of recursion is the Chapter 1 case study escaping a maze. As already explained, in each maze cell the mouse stores a direction to the mouse standing in the cell in which it is currently located. The cells put on the stack are those that should be investigated later reading a cell, it does the same for each visited cell. Write a program that uses recursion to solve the maze problem. Use the following pseudocode

```
exitCell(currentCell)
    if currentCell is marked
        return
    else exitCell(the passage above currentCell)
        exitCell(the passage below currentCell)
        exitCell(the passage left to currentCell)
        exitCell(the passage right to currentCell)
```

Bibliography

Recursion and Applications of Recursion

Barron, David W., *Recursive Techniques in Programming*, New York: Elsevier, 1975.

Burton, Warren and Brock, Philippe, *Algorithms: The Construction, Proof and Analysis of Programs*, New York: Wiley, 1986, Chapter 5.

Gill, Richard C., *Programs and Computers*, New York: Wiley, 1976.

Sedgewick, William H., *Recursion: Programming Techniques*, Reading, MA: Addison-Wesley, 1983.

Stevens, Richard, *Advanced Algorithms*, Second Edition, Chicago: Science Research Associates, 1984.

Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Englewood Cliffs, NJ: Prentice-Hall, 1976.

Transformations Between Recursion and Iteration

Arsac, Jacques, and Kodratoff, Yves, "Some Techniques for Recursion Removal from Recursive Functions," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, April 1982, pp. 295–322.

Barron, David W., "Recursion + Iteration = Structure," *Proceedings of Structure in Computer Science and Engineering*, 1975, pp. 1–13.

Darlington, John, "A Synthesis of Several Sorting Algorithms," *Acta Informatica*, Vol. 11, No. 1, 1978, pp. 1–30.

Algorithms to Solve the Eight Queens Problem

Wirth, Niklaus, *Algorithms and Data Structures*, Englewood Cliffs, NJ: Prentice-Hall, 1986.

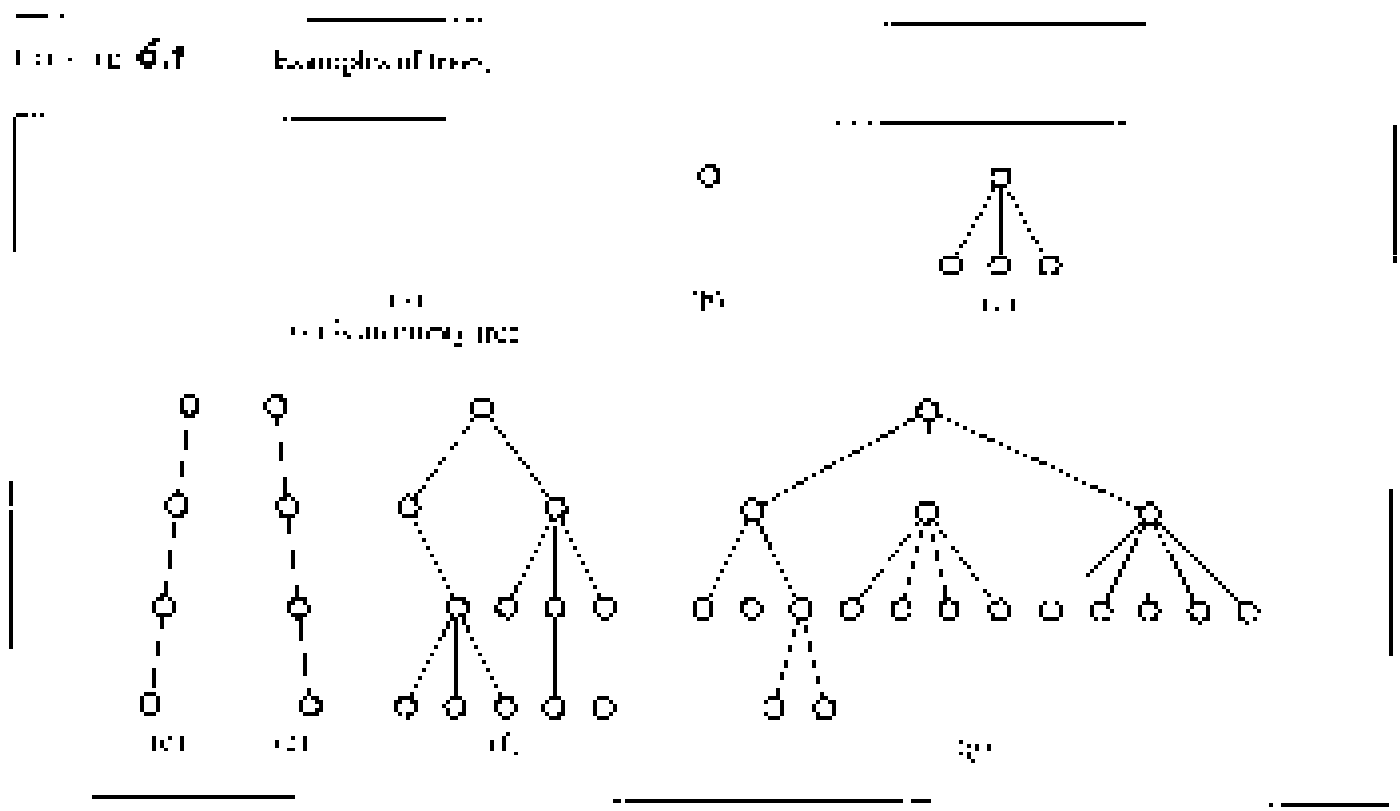
Binary Trees

6.1 Trees, Binary Trees, and Binary Search Trees

Linked lists usually provide great flexibility but they are linear structures and it is difficult to use them to organize a hierarchical ordering of objects. Although stacks and queues relax this restriction slightly, they are limited to only one dimension. To overcome this limitation, we create a new data type called a tree that consists of nodes and arcs. While trees of most any type can be described, we will concentrate on binary trees. A binary tree is a tree that has a parent that can have only two children. Trees can be described recursively as follows. Note that their children are empty trees. A tree can be defined recursively as the following:

1. An empty structure is an empty tree.
2. If [illegible] is a binary tree, then the structure made of a root node and two subtrees, called the left and right subtrees, is a tree.
3. Only structures generated by rules 1 and 2 are trees.

Figure 6.1 shows an example of a tree. Nodes can be reached by following a unique sequence of arcs, called a path. The number of arcs in a path is called the length of the path. The level of a node is the length of the path from the root to the node plus 1 (which is the number of nodes in the path). The height of a tree is the maximum level of any node in the tree. The empty tree is the tree with height 0. The height of a leaf is 1. The level of the root node is 1. The height of a tree is the maximum level of any node in the tree



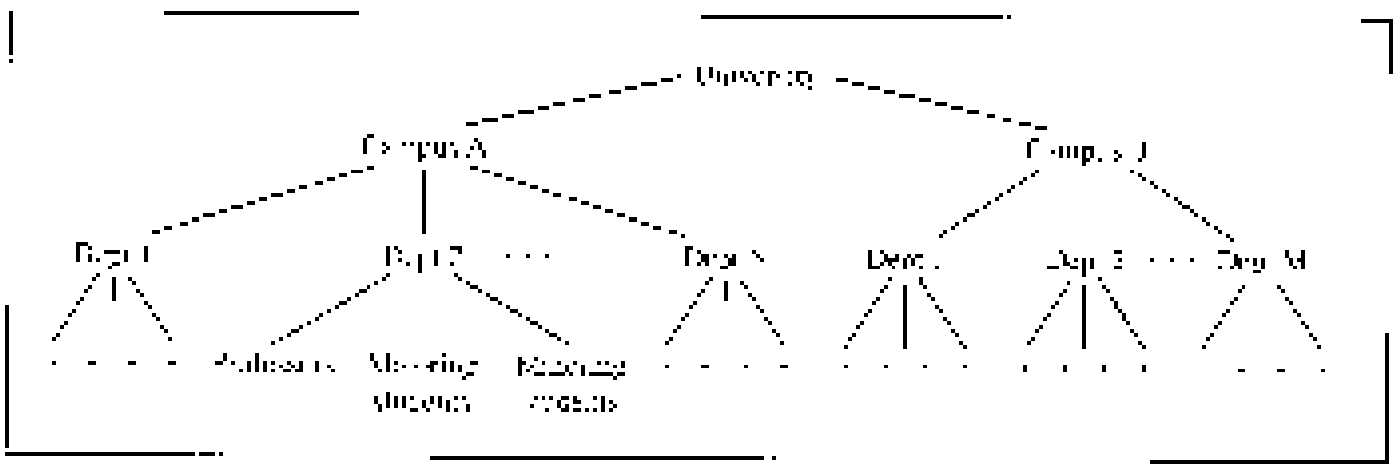
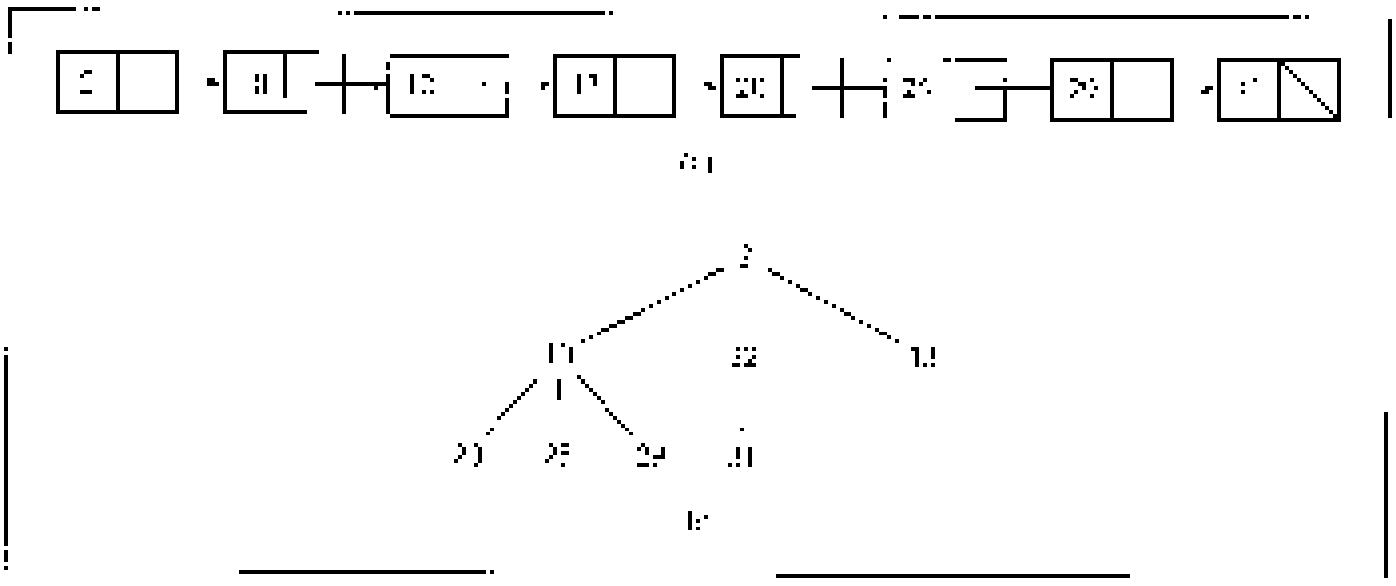


Figure 6.2 contains an example of a tree that arises in the study of the economy. Other examples are genealogical trees, trees of the type that grammarians study to analyze sentences, trees showing the hierarchical structure of organizations, and trees representing the possible outcomes of various sequences of events. Virtually all areas of human endeavor use trees to represent hierarchical relationships.

The definition of a tree does not impose any condition on the number of nodes at a given node. This number can vary from 0 to any integer. In binary search trees, this is a welcome property, but cases do arise in which we only use two branches, but such ranges can have a different number of operations. Such trees are used in databases.

FIGURE 6.3 Transforming a linked list into a tree.



management systems, especially in the hierarchical model. In this section, we discuss the only reason for this optimization. In fact, in the discussion of balanced binary trees in Chapter 10, we will see that this optimization is the equivalent of expression trees. Thus, this section serves as a preview of more powerful techniques to accelerate the search process.

Consider a linked list with elements. To locate an element, the search must start from the beginning of the list and the list must be scanned until the element is found or the end of the list is reached. Even if the list is ordered, the search algorithm has to start from the first node. Thus, if the list has 10,000 nodes and the desired item is the last node, 9,999 of its predecessors have to be traversed unnecessarily. Even worse, if all the elements are stored in an ordered manner, all elements are examined sequentially, so many predecessors of the node with the largest number of tests can be very time consuming. Even when the element is the greatest in the node for which we are searching, the linked list in Figure 6.3(a) can be transformed into the more efficient tree in Figure 6.3(b).

What is a reasonable criterion for ordering nodes in a tree? To test for equality, whether or not it is in the tree, only one test has to be performed. Obviously, the number of tests can be reduced further if these nodes are ordered from smaller and smaller on the left to larger on the right. What would an algorithm be like that moves us to make a decision to go one way or the other, left or right, at each node? Suppose we start at the root, 25, and are looking for 31. Since 31 is greater than 25, we go to its right child, 32, and are finished with the left subtree of 25. The search for 31 could be ignored on the same level as 32, as well as the left child of 32. With this ordering of the tree, nothing really interesting was involved in the search process. This is discussed later in this chapter when we speak of searching. Other orderings could be chosen.

Again, note that nodes can have any number of children. In fact, there are applications where each node has a definite number of children. We will discuss this in the next chapter. There we will study binary trees. A binary tree is a tree where nodes have no children (possibly), or one child is designated as either a left child or

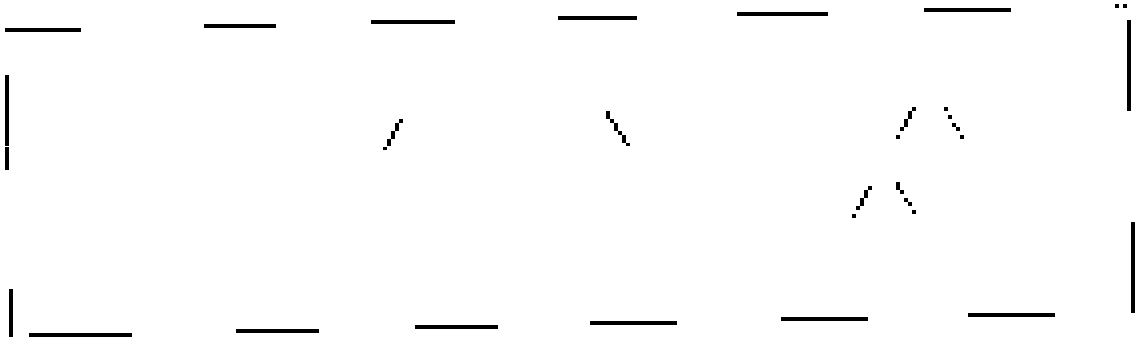


FIGURE 6.4 Examples of binary trees.

...g(n) children. For example, the trees in T_{6a} in Fig. 6.4 are binary trees, whereas the trees in T_{6b} in Fig. 6.4 are not. An important characteristic of binary trees, which is used later in this section, in the analysis of searching algorithms, is the number of leaves.

As already defined, the level of a node is the number of arcs traversed from the root to the node. Alternatively, according to this definition, the root is at level 0, its children are at level 1, and so on. If all nodes at all levels except the last level have two children, then there would be $1 = 2^0$ node at level 0, $2 = 2^1$ nodes at level 1, $4 = 2^2$ nodes at level 2, and generally 2^k nodes at level k. A tree satisfying this condition is referred to as a complete binary tree. In this tree, all nodes have both their children, and all leaves are at the same level. Consequently, in all binary trees, there are at most 2^k nodes at level k. In Chapter 9, we calculate the number of leaves in a complete binary tree in which all nodes have two descendants unless they are leaves. Because leaves can be anywhere in the tree and appear at any level except level 1, no general application formula can be given to calculate the number of nodes because it may vary from one tree to another. But this formula can be approximated by the following theorem.

> **Theorem 6.1** In all nonempty binary trees where nonterminal nodes have exactly two nonempty children, the number of leaves n_0 is one more than the number of nonterminal nodes; that is, $n_0 = n_2 + 1$.

For a tree having one node, this observation holds trivially. If it holds for a certain tree, then attaching two leaves to one of the already existing leaves, thereby converting a nonterminal node, increases n_0 by 1 and n_2 by 1. Hence, the new tree has been produced. The tree is incremented by 2. After these two increments, the equation $n_0 = n_2 + 1$ is left unchanged. That is, $n_0 = n_2 + 1$, which is exactly the original situation. See Exercise 6.3(a). It implies that an n-level complete binary tree has 2^n leaves, and an account of the preceding observation, it also has $2^n - 1$ nonterminal nodes, which makes $2^n + 2^n - 1 = 2^{n+1} - 1$ nodes in total. See also Figure 6.5.

In this chapter, the binary search trees are called ordered binary trees, a set of properties are imposed. A binary search tree is one that satisfies the following property: for each node of the tree, if its value is stored in it, and all values stored in the right subtree are greater than its

FIGURE 6.5 Adding a leaf to the tree in Figure 6.4 preserving the relationship between the number of leaves and the number of nonterminal nodes in (a).

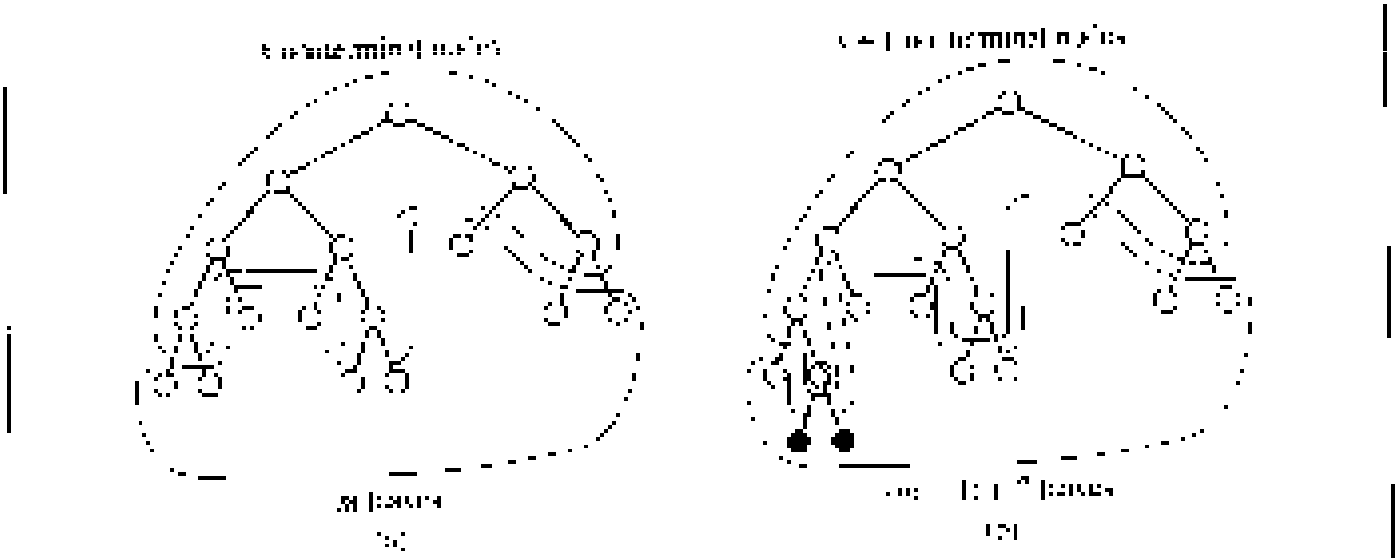
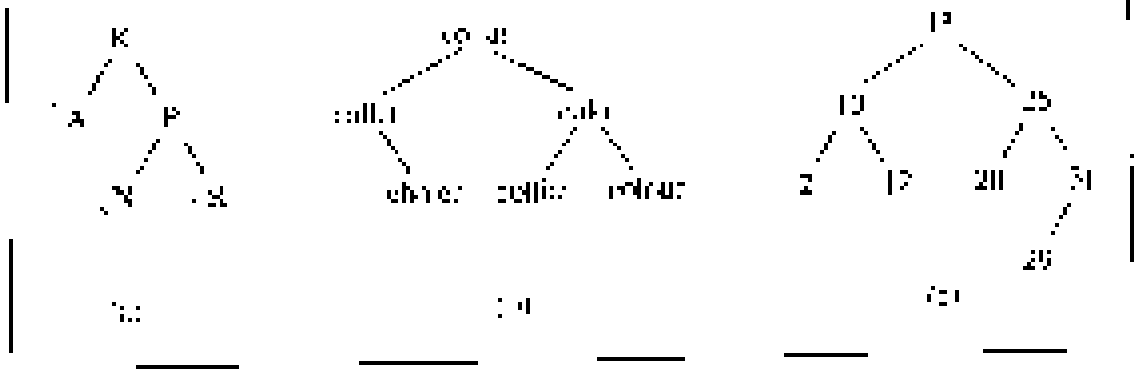


FIGURE 6.6 Some invalid binary search trees.



For positions to be balanced, each stack must contain the same value in the same position. An attempt to do so on the left side would result in the string "great" being stored in the stack, while on the right side, the string "great" would be stored. Depending on the contents of the stacks, the "great" on the left side may or may not be popped off. Dependency on the contents of the stacks is also the case for strings of the form $a^n b^n$. The reason is that a finite-state machine cannot count to an arbitrary number. Similarly, Figure 6.6(c) is not a binary search tree because the value 25 is in the left subtree of the root, 20. Therefore, searching would not be optimized.

6.2 Implementing Binary Trees

Binary trees can be implemented in many ways. Two ways are arrays and linked structures. The implementation of a tree as an array, above, is declared as a structure with an information field and two "pointer" fields. These pointer fields contain the array cells in which the left and right children are stored, if there are any. For example, the tree from Figure 6.5 can be represented as the array in Figure 6.7. The root is always located in the first cell, cell 0, and -1 indicates a null child. In this representation, the two children of node 0 are located in positions 1 and 2, and the right child of node 31 is null.

However, this implementation may be inconvenient. It is often the case with array allocation, since the size of the array also becomes part of the program and must be known in advance. It is a problem because the data may overflow the array if not enough space is allocated, or memory space may be wasted if too much space is allocated. This is important because trees often change and it may be hard to predict how many nodes will be created during program execution. However, sometimes an array implementation of a tree is convenient and desirable. It is used when allocating the heap sort, although an array of ints rather than an array of structures is used there. Usually, a dynamic data structure is a more efficient way to represent a tree. The compiler's examples use dynamic data structures.

In the new implementation, a node is an instance of a class containing an information member and two pointer members. This class is used and operated on by member functions in another class that pertains to the tree as a whole (see Figure 6.8). The two pointer members of BSTNode are declared public because they can be accessible only from non-public members of objects of type BST so that the information hiding principle will remain. It is important to have members of BSTNode be public because otherwise they are not accessible to the classes derived from BST.

FIGURE 6.7 Array representation of the tree in Figure 6.5.

Index	Info	Left	Right
0	[illegible]	[illegible]	[illegible]
1	[illegible]	[illegible]	[illegible]
2	[illegible]	[illegible]	[illegible]
3	[illegible]	[illegible]	[illegible]
4	[illegible]	[illegible]	[illegible]
5	[illegible]	[illegible]	[illegible]
6	[illegible]	[illegible]	[illegible]
7	[illegible]	[illegible]	[illegible]

Figure 6.8 Implementation of a generic binary search tree.

```
// ... generic binary search tree

#include <queue>
#include <stack>

using namespace std;

#ifndef BINARY_SEARCH_TREE
#define BINARY_SEARCH_TREE

template <class T>
class Stack : public stack<T> { ... } // as in Figure 4.2

template <class T>
class Queue : public queue<T> {
public:
    T dequeue() {
        T top = front();
        queue<T>::pop();
        return top;
    }
    void enqueue(const T& e) {
        push(e);
    }
};

template <class T>
class BSTNode {
public:
    BSTNode() {
        left = right = 0;
    }
    BSTNode(const T& e, BSTNode *l = 0, BSTNode *r = 0) {
        key = e; left = l; right = r;
    }
    T key;
    BSTNode *left, *right;
};
```

Program 6.8 (continued)

```
template<class T>
class BST {
public:
    BST() {
        root = 0;
    }
    ~BST() {
        clear();
    }
    void clear() {
        clear(root); root = 0;
    }
    bool isEmpty() const {
        return root == 0;
    }
    void preorder() {
        preorder(root);                    // Figure 6.11
    }
    void inorder() {
        inorder(root);                     // Figure 6.11
    }
    void postorder() {
        postorder(root);                   // Figure 6.11
    }
    T* search(const T& el) const {
        return search(root,el);            // Figure 6.9
    }
    void breadthFirst();                   // Figure 6.10
    void iterativePreorder();              // Figure 6.13
    void iterativeInorder();               // Figure 6.17
    void iterativePostorder();             // Figure 6.15
    void MorrisInorder();                  // Figure 6.22
    void insert(const T&);                 // Figure 6.23
    void deleteByMerging(BSTNode<T>*&);    // Figure 6.24
    void findAndDeleteByMerging(const T&); // Figure 6.25
    void deleteByCopying(BSTNode<T>*&);    // Figure 6.32
    int balanced(int,int);                 // Section 6.3
    . . . . . .
protected:
    BSTNode<T>* root;
};
```

Figure 6.8 (continued)

```
    void inorder(BSTNode<T>*);
    bool search(BSTNode<T>*, const T&) const;  // Figure 6.9
    void preorder(BSTNode<T>*);                // Figure 6.10
    void inorder(BSTNode<T>*);                 // Figure 6.10
    void postorder(BSTNode<T>*);               // Figure 6.10
    virtual void visit(BSTNode<T>* p) {
        cout << p->key << " ";
    }
    // ... rest of the class ...
};

// ...
```

6.3 Searching a Binary Search Tree

An algorithm for finding a given key in a binary search tree might proceed as follows. Starting at the root, the key in the root node is compared with the search key. If the search key is less than the root key, the left subtree of the root node is searched; if the search key is greater than the root key, the right subtree is searched. If the search key is equal to the root key, the search is successful. If the subtree to be searched is empty, the search is unsuccessful. For example, if the search key is 25 in the tree in Figure 6.5, initially it is compared with 15, the root key. Since $25 > 15$, the search proceeds to the right child of the root, 20. Since $25 > 20$, the search proceeds to the right child of 20, 22. Since $25 > 22$, the search proceeds to the right child of 22, 25. Since $25 = 25$, the search is successful.

In general, searching a binary search tree requires four tests (why?). In the worst case, the number of tests is fewer than four. Depending on the search key, one to four tests are necessary to find a node or to determine that the node is not in the tree. Two approaches are possible. One approach locates the node and returns a pointer to it. The other approach determines whether the node is in the tree and returns a bool value. If the search key is in the tree, both approaches require the same amount of work. If the search key is not in the tree, the first approach requires more work than the second approach. For example, if the search key is 24, the first approach returns a pointer to node 25, the last node visited before the search fails. The second approach returns a value of false.

Figure 6.9 A function for searching a binary search tree

```
template<class T>
T* BST<T>::treeSearch(BSTNode<T>* p, const T& el) const {
    while (p != 0)
        if (el == p->key)
            return &p->key;
        else if (el < p->key)
            p = p->left;
        else p = p->right;
    return 0;
}
```

The worst case scenario when the leaf level has to be reached is experienced when one of the most remote leaves of the desired element is encountered.

The complexity of searching is measured by the number of comparisons performed during the searching process. This number depends on the number of nodes encountered on the way to a path leading from the root to the node being searched for. Therefore, the worst case is the length of the path leading to the node. But the worst case depends on the shape of the tree and the position of the node in the tree.

The *internal path length* (IPL) is the sum of all path lengths of all nodes of a tree. It can be calculated by summing $\sum_{i=1}^{h} i \cdot n_i$ over all levels i, where h is the number of levels and n_i is the number of nodes on level i. A position of a node in the tree is determined by the path length. An average search path, called an *average path length*, is given by the formula IPL/n, where n depends on the shape of the tree. In the worst case, when the tree resembles a linked list, $\text{path}_{\text{avg}} = \frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$, and a search takes n time units.

For best cases, when all leaves on the rightmost two levels are in the most favorable positions, and only one of the nodes on level $h-1$ has one child, the sum of the path lengths, or internal path length, can be expressed by the formula path_{avg} for the corresponding path of a single element on level h and the rest on level $h-1$:

$$\text{IPL} = 2\text{IPL} - \text{IPL} = n(h-1) - \sum_{i=1}^{h-1} 2^i = n(h-1) - 2^h + 2$$

As was shown by Knuth, the number of nodes in a complete binary tree of height h is $2^h - 1$, so

$$\text{path}_{\text{avg}} = \text{IPL}/n = h - 1 - (2^h - 2)/n = h - 1 - (2^h - 2)/(2^h - 1)$$

A worst-case search for a tree with the height $h = 10$ requires 10 comparisons, but the height of a tree with 1,023 nodes can be as low as 9. The average path length for such a tree is $\text{path}_{\text{avg}} = 9 - 1 - (2^9 - 2)/(2^9 - 1) = 7 - 510/511 \approx 7$. The average path length for a perfectly balanced tree with 1,023 nodes is $\log_2(n+1) - 1 = \log_2 1024 - 1 = 10 - 1 = 9$. Clearly, a perfectly balanced tree is a very good tree.

The average case for an average tree is somewhere between $\frac{1}{2}n$ and $\frac{1}{4}n(n-1)$. It is possible for a node in an average position in a tree of average shape to require $O(n)$ or only $O(\lg n)$. First, the average shape of the tree has to be represented computationally.

The root of a binary tree can have an empty left subtree and a right subtree with $n-1$ nodes. It also can have one node in the left subtree and $n-2$ nodes in the right and so on. Finally, it can have an empty right subtree with all remaining nodes in the left. The same possibility can be applied to both subtrees of the root, to the subtrees of those subtrees, down to the leaves. The average internal path length is the average over all these different shaped trees.

Assume that the tree contains nodes 1 through n. If 1 is the root, then its left subtree has $i-1$ nodes and its right subtree has $n-i$ nodes. If $path_{i-1}$ and $path_{n-i}$ are the average paths in those subtrees, then the average path of this tree is

$$path_n(i) = ((i-1)(path_{i-1}+1) + (n-i)(path_{n-i}+1))/n$$

Assuming that elements are coming randomly to the tree, the root of the tree can be any number 1 through n. Therefore, the average path of an average tree is obtained by averaging i values of $path_n(i)$ over all values of i. This gives the formula

$$path_n = \frac{1}{n}\sum_{i=1}^{n} path_n(i) = \frac{1}{n}\sum_{i=1}^{n}((i-1)(path_{i-1}+1) + (n-i)(path_{n-i}+1))$$

$$= \frac{2}{n}\sum_{i=0}^{n-1} path_i + n - 1$$

From this, and from $path_1 = 0$, we obtain $path_n \approx 2n\ln n - 1.386n$. By mathematical induction or part c (see section A.1 in Appendix A). This is an approximation for the average number of comparisons in an average tree. This number is $O(\lg n)$, which is closer to the best case than to the worst case. This number also indicates that there is little room for improvement, since $path_{n-1}(path_n) \approx 0.15$, and the average path length in the best case is different by only 0.5 from the expected path length in the average case. Searching in a binary tree is therefore very efficient in most cases, even without balancing the tree. However, this is true only for randomly constructed trees because in highly unbalanced and elongated trees, which are almost reversed linked lists, search time is $O(n)$, which is unacceptable considering that $O(\lg n)$ efficiency can be achieved.

6.4 Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly one time. Traversal may be interpreted as putting all nodes on one line or linearizing a tree.

The definition of traversal specifies only one condition—visiting each node only one time—but it does not specify the order in which the nodes are visited. Hence, there are as many tree traversals as there are permutations of nodes; for a tree with n nodes, there are $n!$ different traversals. Most of them, however, are rather chaotic and do not indicate much regularity so that implementing such traversals lacks generality. For each, a separate set of traversal procedures must be implemented, and only a few of them can be used for a different number of uses. For example, two possible traver-

[illegible] [illegible] [illegible] [

Program 6.10 Top-down, left-to-right breadth-first traversal implementation.

```
template<class T>
void BinaryTree<T>::breadthFirst() const {
    Queue<BinaryTreeNode<T>*> queue;
    BinaryTreeNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.isEmpty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                queue.enqueue(p->left);
            if (p->right != 0)
                queue.enqueue(p->right);
        }
    }
}
```

In the implementation, code still seems like a bit of a mystery. It can be used to perform traversals where the move is always from left to right and attention is focused on the first column. The three traversals are given their standard names:

VLR—preorder tree traversal

LVR—inorder tree traversal

LRV—postorder tree traversal

Error and exception functions can be implemented directly from the symbolic descriptions of these three traversals, as shown in Figure 6.11.

These functions may seem too simplistic, but their real powers lie in recursion, in fact, circular recursion. The real job is done by the system on the run-time stack. This is simpleness coding that lays a heavy burden upon the system. To better understand this process, binary tree traversal is discussed in some detail.

In inorder traversal, the left subtree of the current node is visited first, then the node itself, and finally, the right subtree. All of this, obviously, holds if the tree is not empty. Before drawing aside the run-time curtain by examining the run-time stack, the output given by the inorder traversal is determined by referring to Figure 6.12. The following steps correspond to the letters in that figure:

(a) Node D is the root, so `visit(D)` is called for the first time. The function calls itself for node D's left child, node B.

(b) Node B is not null, so `inorder(p)` is called for node B. Because node B has both a left and a right subtree, `inorder(p)` is called for node B's left child, node A.

FIGURE 6.11 Inorder traversal implementation.

```
template<class T>
void BSTree<T>::inorder(BSTreeNode<T> *p) {
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}

template<class T>
void BSTree<T>::preorder(BSTreeNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}

template<class T>
void BSTree<T>::postorder(BSTreeNode<T> *p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

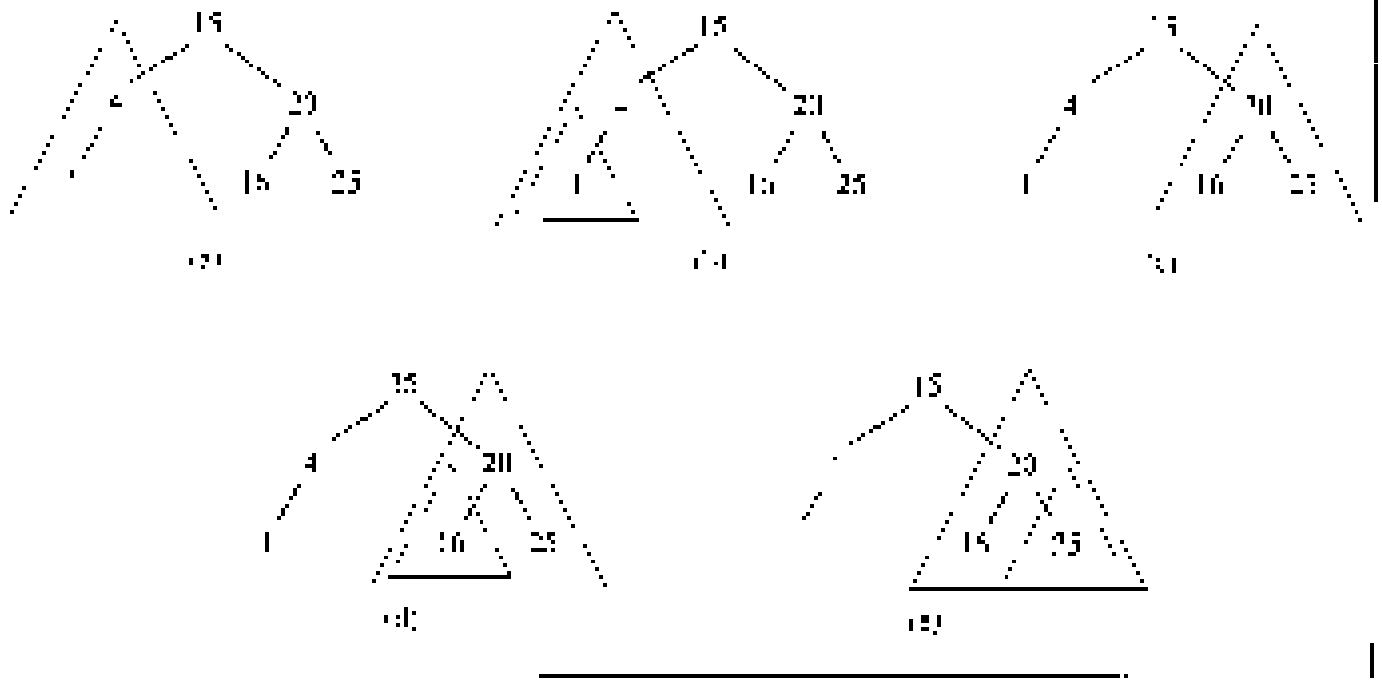
inorder recursive calls of `inorder(p)` passes the execution to the `if` statement in which `inorder(p->left)` is called. When the left subtree is finished, node 1 is visited and then a quick call to `inorder(p)` is executed for the null right subtree of node 1. After returning the call for node 1, node 3 is visited. Node 3 has a null left subtree, so `inorder(p)` is called once to check that, and upon returning the call for node 15, node 15 is visited.

(c) Node 15 has a right child, so `inorder(p)` is called for node 20.

(d) `inorder(p)` is called for node 16, the node is visited, and then for its null left subtree, which is followed by visiting node 15. After a quick call to `inorder(p)` on the null right subtree of node 16 and returning the call on node 20, node 20 is also visited.

(e) `inorder(p)` was called on node 30, then on its empty left subtree, node 35 is visited, and finally `inorder(p)` was called on node 35's empty right subtree.

FIGURE 6.12 In-order traversal.



If the visit includes printing the value stored in a node, then the output is:

1 4 5 16 20 25

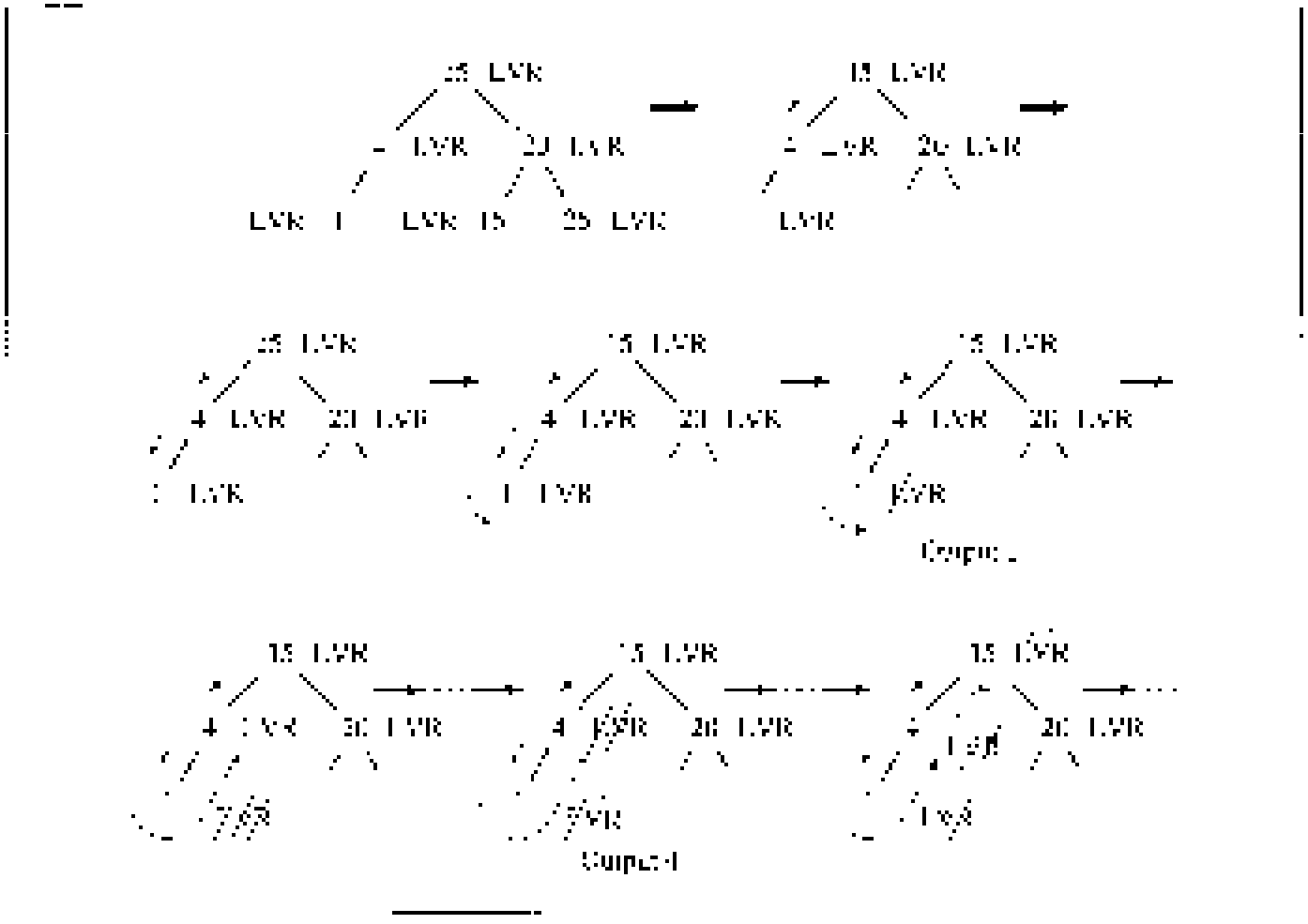
The way the traversal is carried out is the following. Three tasks L, V, and R are performed for each node encountered. The traversal of the right subtree of a node is held until the first two tasks, L and V, are accomplished. If the latter two are finished, the node can be printed, as in Figure 6.13.

To see how the `inorder()` works, the behavior of the run-time stack is observed. The numbers in parentheses in Figure 6.14 indicate return addresses shown on the left-hand side of the code in Listing 6.11.

```
template <class T>
void BinaryTree<T>::inorder(BSTNode<T> *node) {
    if (node != 0) {
/* 1 */     inorder(node->left);
/* 2 */     visit(node);
/* 3 */     inorder(node->right);
/* 4 */ }
}
```

A circled address with an up arrow and a number indicates the return value of a node pushed onto the stack. For example, ↑2 means that a node points to the node of the return value is the number 2. Figure 6.15 shows the changes of the run-time stack when `inorder()` is executed for the tree in Figure 6.12.

FIGURE 6.13 Details of execution of the first steps of inorder traversal



(a) Initially, the run-time stack is empty (or we don't assume that the stack is empty by disregarding what has been stored on it before the call to inorder).

(b) Upon the first call, the return address (the address of the instruction following the call to inorder) and the current value of parameter node (the address of the root) are pushed onto the stack. The execution of the body of inorder begins with node's left child (node.left) assigned to node.

(c) Before the recursive call is executed, the return address (2) and current value of node (↑1) are pushed onto the stack. Since node is not null, inorder is about to be invoked for node's left child (↑1).

(d) Before the second recursive call, the return address (3) and the node value (↑1) are pushed onto the stack.

(e) Inorder is called with node's left child (null), the address (4) and the current value of parameter node (null) are stored on the stack. Since node is null, inorder is exited immediately; upon exit, the activation record is removed from the stack.

(f) The execution proceeds in this manner, pushing activation records onto the stack for each recursive call (and popping them off the stack upon exit from each call) until the node with value 1 is processed, at which point no more recursive calls are made and the stack is empty.

FIGURE 6.14 Changes in the run time stack during inorder traversal.

Program 6.15 A nonrecursive implementation of preorder tree traversal

```
template<class T>
void BinaryTree<T>::iterativePreorder() {
    LinkedStack<BinaryTreeNode<T>*> travelStack;
    BinaryTreeNode<T> *p = root;
    if (p != 0) {
        travelStack.push(p);
        while (!travelStack.empty()) {
            p = travelStack.pop();
            visit(p);
            if (p->rightChild != 0)
                travelStack.push(p->rightChild);
            if (p->leftChild != 0)   // left child pushed after right
                travelStack.push(p->leftChild); // to be on the top of
        }                           // the stack;
    }
}
```

(g) With the right child of node 1 on the stack, the statement under (f) is executed, which finds node 2 as the left child of node 1. This time, however, the address of node 2 is not null, so it is pushed onto the stack. Because node 2 has no left child, nothing is pushed; upon exit, the stack is cleaned up.

(h) Node 2 is visited, and its right child (node 4) is pushed onto the stack. Then it is popped, leaving node 4 on the stack.

(i) Since this is node 4, which has no children, the system removes the current activation record and reflects again on the stack, restores the node's value, 1, and returns execution from statement (f). This prints the number 1 and then calls `nonrecursive()` for the right child of node 1, which is null.

These steps are just the beginning. All of the steps are shown in Figure 6.16.

At this point, consider the problem of a nonrecursive implementation of the three traversal algorithms. As indicated in Chapter 5, a recursive implementation has a tendency to be less efficient than a nonrecursive counterpart. If two recursive calls are used in a function, then the problem of possible inefficiency doubles. Can recursion be eliminated from the implementation? The answer has to be positive because recursion is not eliminated in the source code, the system does it for us anyway. So the question should be rephrased: Is it expedient or desired?

Look first at a nonrecursive version of the preorder traversal shown in Figure 6.15. The function `iterativePreorder()` is not as elegant as `preorder()`, but it is still short and legible. However, it uses a stack heavily. If stack operations are necessary to process the stack, and the overall implementation is not as clear. Although two recursive calls are omitted, there are now up to four calls (not counting c

[illegible] The standard recursive implementation is different.

It is easy to implement algorithms for predicate types such as `isLeaf` for the binary tree ADT. The implementation of `isLeaf` is given in Section 6.2. Other predicates, such as `isLeftChild`, require access to the parent of the node in question. Because the parent node is not explicitly stored in the node, the only way to find it is to start at the root and search for the node in question. Thus, visiting a node's parent requires a traversal. Because we have to determine whether the node is a left or right child, we can use `inorderTraversal` to implement the predicate. Function `isLeftChild` traverses the tree until it finds the node in question. If this node is not the root, `isLeftChild` checks whether the node is the left child of its parent. Function `isRightChild` is implemented in a similar way. What makes these functions inefficient is that `visit` has to traverse the tree to find the parent before it can check whether the node is a left or right child. Therefore, nonrecursive implementations of predicates for postorder traversals have to be developed independently.

A nonrecursive version of postorder traversal can be obtained relatively easily if we observe that the sequence generated by a left-to-right postorder traversal (LRV order) is the mirror image of a reversed sequence generated by a right-to-left preorder traversal (VRL order). In this case, the implementation of `iterativePreorder()` can be adapted to provide `iterativePostorder`. This means that two stacks have to be used to visit each node in the reverse order. Left-to-right preorder traversal is included. It is, however, possible to develop a function for postorder traversal that pushes onto the stack a node that has two descendants once before traversing its left subtree and once before traversing its right subtree. An auxiliary pointer is used to distinguish between these two cases. Nodes with one descendant are pushed only once, and leaves do not need to be pushed at all (Figure 6.6).

A nonrecursive inorder tree traversal is also a complicated matter. One possible implementation is given in Figure 6.7. In cases like this, we can clearly see the power of recursion. `iterativeInorder` is almost unreadable and without thorough explanation, it is not easy to determine the purpose of this function. On the other hand, recursive `inorder` immediately demonstrates a purposeful logic. Therefore, `iterativeInorder` can be defended in one case only, if it is known that there is a substantial gain in execution time and that the function is called often in a program. Otherwise, `inorder` is preferable over its iterative counterpart.

6.4.3 Stackless Depth-First Traversal

Threaded Trees

The traversal functions analyzed in the preceding section, whether recursive or nonrecursive, use both time and space implicitly or explicitly to store information about nodes that are processed at a later time. In short, in the case of recursive functions, the run-time stack was utilized. In the case of nonrecursive variants, an explicitly defined and user-maintained stack was used. The common idea was that some additional space had to be spent to maintain the structure of some tree-space had to be set aside for the stack itself. In the worst case, which occurs frequently, the stack may hold information about almost every node of the tree, requiring [illegible]

FIGURE 6.16 A nonrecursive implementation of postorder traversal

```
template <class I>
void PBTCtrl::iterativePostorder() {
    Stack<PBTBlock<I>*> travStack;
    PBTBlock<I>* p = root, *q = root;
    while (p != 0) {
        for ( ; p->left != 0; p = p->left)
            travStack.push(p);
        while (p != 0 && (p->right == 0 || p->right == q)) {
            visit(p);
            q = p;
            if (travStack.empty())
                return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

It is more difficult to implement iterative postorder traversal than it is to implement iterative preorder or inorder traversal. The reason is that a node may be visited only after the traversal of its right subtree is completed, and traversal does not return to the node after the traversal of its left subtree is completed. Therefore, the node must be pushed onto the stack before the traversal of its left subtree, which explains why it is pushed onto the stack twice.

The program in Figure 6.16 uses a variable `q` to store the address of the node that was last visited. This variable is used to determine whether the traversal of a node's right subtree has been completed and the node can be visited. Variable `p` stores the address of the node that is currently being processed. Initially, `p` points to the root node. The program uses a nested loop. The inner loop pushes the nodes on the path from the root node to the leftmost node onto the stack. The outer loop pops a node off the stack and visits it if it has no right subtree or if the traversal of its right subtree has been completed. Otherwise, the node is pushed back onto the stack, and the traversal of its right subtree is initiated.

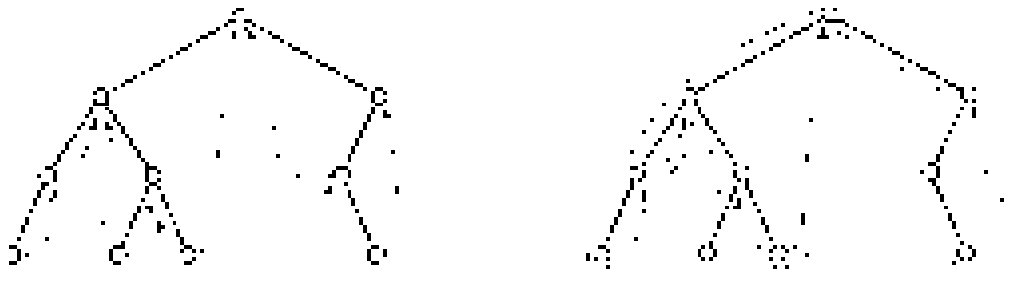
Figure 6.17 suggests that the program in Figure 6.16 visits each node twice. This is not true. Each node is visited only once. Figure 6.18 illustrates the traversal of a binary tree, which corresponds to the expression $(a + b) * (c - d)$.

The function is relatively simple. The `for` loop in Figure 6.16 directs the order in which processed nodes in the tree. Note that only one variable, `p`, is needed to traverse the tree. No stack is needed to store the addresses of the visited nodes. Instead, nodes contain data to direct the traversal. The `right` pointer is temporarily used in the implementation of `traverseInorder()`. The Boolean data member `visited` plays this role as shown in Figure 6.19. However, it requires additional memory to

FIGURE 6.17 A C++ nonrecursive implementation of inorder tree traversal

```
template <class T>
void BSTree<T>::iterInorder() {
    LinkedStack<BSTreeNode<T>*> travStack;
    BSTreeNode *p = root;
    while (p != 0) {
        while (p != 0) {            // stack the right child (if any);
            if (p->right)           // and then the node itself when going
                travStack.push(p->right); // to the left;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop();        // pop a node with no left child
        while (!travStack.empty() && p->right == 0) { // visit it
            visit(p);               // and all nodes with no right
            p = travStack.pop();    // child;
        }
        visit(p);                   // visit also the first node with
        if (!travStack.empty())     // a right child (if any);
            p = travStack.pop();
        else p = 0;
    }
}
```

FIGURE 6.18 (a) A threaded tree and (b) an inorder traversal's path; (c) a threaded tree with right successors only



(a) (b)

Figure 6.19 Implementation of the generic threaded tree and the inorder traversal of a threaded tree.

```
// ... Header : A Binary Search Threaded Tree

#include <iostream.h>
#define Threaded 1

template<class T>
class ThreadedNode {
public:
    ThreadedNode() {
        left = right = 0;
    }
    ThreadedNode(const T& el, ThreadedNode *l = 0, ThreadedNode *r = 0) {
        key = el; left = l; right = r; successor = 0;
    }
    T key;
    ThreadedNode *left, *right;
    unsigned int successor : 1;
};

template<class T>
class ThreadedTree {
public:
    ThreadedTree() {
        root = 0;
    }
    void insert(const T&);                    // Figure 6.20
    void inorder();
protected:
    ThreadedNode<T> *root;
};

template<class T>
void ThreadedTree<T>::inorder() {
```

FIGURE 6.19 (continued)

```
    ThreadedNode<T> *prev, *p = root;
    if (p != 0) {                    // process only nonempty trees;
        while (p->left != 0)         // go to the leftmost node;
            p = p->left;
        while (p != 0) {
            visit(p);
            prev = p;
            p = p->right;            // go to the right node and only
            if (p != 0 && prev->successor == 0) // if it is a
                while (p->left != 0) // descendant to the
                    p = p->left;     // leftmost node, otherwise
        }                            // visit the successor;
    }
}
```

computer memory, might be significant in comparison to other fields. However, the exact details are highly dependent on the implementation. The operating system almost certainly pads the structure with additional bits for proper alignment of machine words. If so, a successful method is one byte, four or more words, defeating the argument about saving space by using threaded trees.

Threaded trees can also be used for preorder and postorder traversals. In preorder traversal, the current node is visited first and then traversal continues with its left descendant if any, or right descendant, if any. If the current node is a leaf, threads are used to go through the chain of ancestors to find another ancestor to continue traversal with the right descendant of the last ancestor.

Postorder traversal is only slightly more complicated. First, a dummy node is created that has the root as its left descendant. In the traversal process, a variable can be used to check the type of the current node. If the current node is left traversal and the current node has a right descendant, then the right descendant is traversed. Otherwise, the action is changed to right traversal. If the action is right traversal and the current node has a right ancestor descendant, then the descendant is traversed and the action is changed to left traversal. If the action is right traversal and the current node does not have a right ancestor descendant, then the current node is visited. Here is the list of the eight possible combinations of cases. First, the beginning point is reached through the thread of the current node, then left, left descendant is reached, and then traversal continues with the left descendant. Second, the beginning point is reached through the thread of the current node, then left, left descendant is reached, and then traversal continues with the right descendant. Third, the beginning point is reached through the thread of the current node, then left, left descendant is not reached, and then traversal continues with the right descendant. Fourth, the beginning point is reached through the thread of the current node, then left, left descendant is not reached, and then traversal continues with the right descendant. Fifth, the beginning point is reached through the right descendant of the current node, then left, left descendant is reached, and then traversal continues with the left descendant. Sixth, the beginning point is reached through the right descendant of the current node, then left, left descendant is reached, and then traversal continues with the right descendant. Seventh, the beginning point is reached through the right descendant of the current node, then left, left descendant is not reached, and then traversal continues with the right descendant. Eighth, the beginning point is reached through the right descendant of the current node, then left, left descendant is not reached, and then traversal continues with the right descendant.

Traversal Through Tree Transformation

The last set of traversal algorithms discussed earlier in this chapter used a stack to retain some information necessary for successful processing. The stack was implemented either explicitly as part of the traversal algorithm or implicitly by the recursion mechanism. Both methods require additional space proportional to the height of the tree. The algorithms discussed in this section do not require a stack or any other data structure to retain information about the traversal. Instead, they temporarily modify the tree structure itself, using pointers that normally point to a node's children to point to other nodes that will help the traversal proceed correctly. Once the traversal is complete, the tree is restored to its original structure. Algorithms that use this approach are sometimes called *threaded* algorithms, because they create a temporary "thread" connecting a tree's nodes in a special order that depends on the order in which the tree is being traversed.

Before we discuss any threaded algorithms, we must first consider how it is possible to traverse a tree without using a stack or recursion. Obviously, if a node's children are destroyed during the traversal, some other method must be found to return to the node's parent after the node has been processed. Unless the node is the root, its parent is unknown. Even if the node is the root, its right child is unknown. Clearly, some additional information must be stored in the tree nodes if we are to traverse the tree without using a stack.

Consider the problem of traversing a tree in preorder. Once we have processed a node's left subtree, we normally move to its right subtree. But how do we find the right subtree? We can't store the address of the right subtree in the node's right child pointer, because that pointer is needed to point to the node's right child. Instead, we can store the address of the right subtree in the rightmost node of the left subtree. For example, in the tree in Figure 6.4.1, node 5 is the rightmost node in node 2's left subtree. Therefore, node 5's right child pointer can be used to point to node 2's right child, node 3. Figure 6.4.2 shows how this tree looks after node 2's left subtree has been traversed.

Here is the preorder algorithm:

```
Move to first node in tree
While the node is not NULL
    If node has no left descendants
        Visit it
        Go to its right child
    Else find the rightmost node in left subtree that points to this node
        Go to the right child of this node
```

This algorithm successfully traverses the tree but only once, since it destroys its original structure. Before the tree is destroyed, however, it can be restored by following the right pointers from the rightmost nodes. This is accomplished by retaining the left pointer of the node moved down to the right and using it to locate the rightmost nodes. The code is in Figure 6.4.3.

A complete version of the algorithm is shown in Figure 6.4.4, and the details of the operation are illustrated in Figure 6.4.5. The following description is divided into numbered parts that correspond to the comments in the code in Figure 6.4.4.

1. Initially, p points to the root node, which has a left child. So we locate the rightmost node in the root's left subtree, which is the right child of node 11, pointed to by pred. Since node 5 has no right child, we make p the right child of pred. Now node 11's right child points to the root, node 6. Node 6's left child pointer points to node 5. Its left child is now the root of the left subtree, node 11. The tree now looks like Figure 6.4.5b.
2. Node 6 has a left child, node 11, which also has a left child. First we locate the rightmost node in this subtree, node 8 (Figure 6.4.5c), and then the current root, node 6,

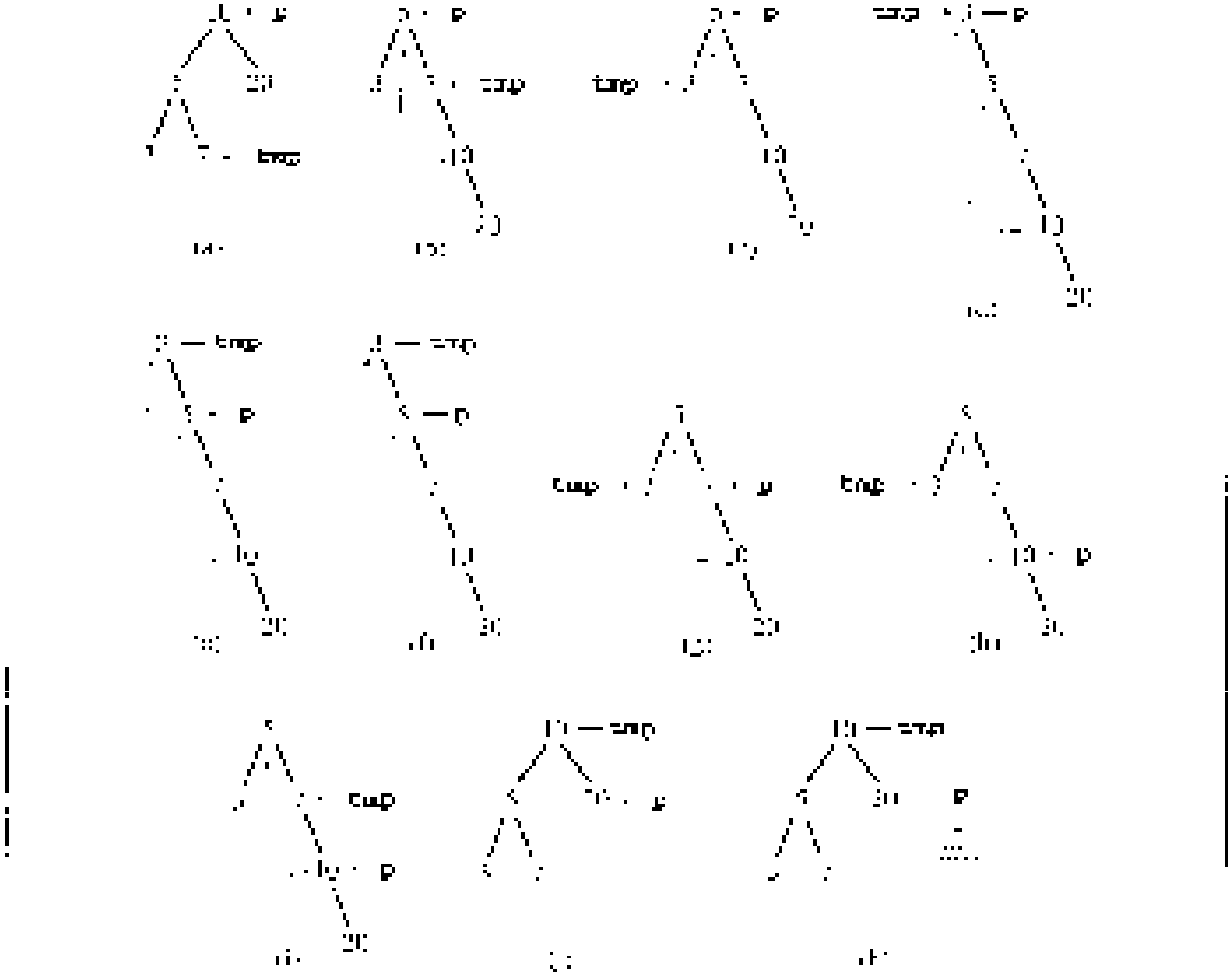
Figure 6.20 Implementation of the Morris algorithm for inorder traversal.

```
template<class T>
void BSTree<T>::Morris_inorder() {
    BSTNode<T> *p = root, *temp;
    while (p != 0)
        if (p->left == 0) {
            visit(p);
            p = p->right;
        }
        else {
            temp = p->left;
            while (temp->right != 0 &&  // go to the rightmost node
                   temp->right != p)    // of the left subtree or
                temp = temp->right;     // to the temporary parent
            if (temp->right == 0) {     // if go if first time
                temp->right = p;        // rightmost node was
                p = p->left;            // reached. Make it a
            }                           // temporary parent of its
            else {                      // rightmost node, else
                                        // a temporary parent was
                visit(p);               // found; visit node p
                temp->right = 0;        // and then cut the right
                                        // pointer of the rightmost
                p = p->right;           // node, whereby it
            }                           // ceases to be a parent;
        }
}
```

is assigned the right child of node 4 which now may contact with node 3 through its left pointer (Figure 6.21d).

3. Because node 3 pointed to by p has no left child, it is therefore visited, and p is assigned to its right child node 5 (Figure 6.21e).
4. Node 5 has a non-null left pointer, so temp finds a temporary parent of node 5, which is the rightmost node currently pointed to by temp (Figure 6.21f). Next node 5 is visited and reconfiguration of the tree in Figure 6.21f is established by setting the right pointer of node 4 to null (Figure 6.21g).
5. Node 5, pointed to by p, is visited, and p moves down to its right child (6.21h).
6. temp again points to the temporary parent of node 10 (Figure 6.21i). Next, node 10 is visited and then reestablished to its state 3 of step 1 by setting up the left pointer of node 5 (Figure 6.21j).
7. Finally, node 20 is visited without further adjustments as it has no left child nor has its position been altered.

FIGURE 6.21 Tree traversal with the Morris method.



This is quite close to the original Morris algorithm. Note that we never need to store the stack explicitly because it is implicit in the recursion [Question 6.2]. This is due to the fact that there are two left children for which we call traverseInOrder recursively: the left child of the root and the left child of the right child of the root. The algorithm performs exactly two passes over each edge in the tree.

Postorder traversal is easily obtained from inorder traversal by using a similar technique: we store the nodes in the inner while loop in reverse order [Question 6.3]. This is also the basis for a traversal algorithm that uses only a constant amount of space [Question 6.4].

Preorder traversal can also be obtained from inorder traversal by processing a node only when it is encountered for the first time. This way it resembles a depth-first traversal [Question 6.5].

traversal stack that in the interactive algorithm, pushing and popping pointers to nodes as we go left and right. An explicit stack is not necessary in the recursive algorithm because the system stack is used implicitly. Each node is visited once, so the time is linear in the number of nodes in the tree. Since no additional memory is required for the recursive algorithm, it is clearly superior in space to the interactive one. The only additional space used is the temporary stack space used by the recursive calls, which could be at most the depth of the tree. Even in a badly unbalanced tree, this space should not cause any problems. In a well-balanced tree, containing thousands of nodes, only a few levels of recursion would be required.

Morris (1979) invented a traversal algorithm that avoids the stack altogether. Instead of using the stack to keep track of where we are in the tree, this algorithm temporarily changes some of the right pointers in the tree, so they point back up the tree. With these changes in place, it is relatively easy to move around the tree as required. Before the algorithm terminates, all of the changes to the tree are undone, so the tree is unaltered when the algorithm is finished. Morris's algorithm is rather complex, and we refer you to the original paper for a description of how it works. Suffice it to say that it is relatively efficient, although it is not as fast as the preorder, inorder, and postorder traversals discussed in this section. All of these require time proportional to the number of nodes in the tree, whereas Morris's algorithm requires time proportional to the number of nodes times the height of the tree. Thus, for a well-balanced tree containing n nodes, Morris's algorithm requires time proportional to $n \log n$, whereas the others require time proportional to n. Morris's algorithm is also somewhat slower by a constant factor than the others. Thus, for trees of moderate size, Morris's algorithm is probably not worthwhile. For very large trees, however, it may be the only practical solution.

6.5 Insertion

Searching a binary tree does not modify the tree. It sometimes is useful to write a procedure that searches for a node and, if it is not found, inserts a new node with the specified key. Such a procedure must be careful to preserve the ordering property of the tree. Thus, it must follow the same path that a search would follow, and the new node must be inserted in the place where a search would look for it.

To insert a new node, called n, into a tree, we search the tree for n's key. If a node with the same key is found, then we have the option of either replacing the node or leaving the tree unchanged. We will leave the tree unchanged. If we reach a leaf node without finding a node with the same key, then n must be inserted as a child of that leaf node. Figure 6.22 illustrates the insertion process.

Figure 6.22 Inserting a node into a binary tree

Implementing this algorithm requires a little more thought than searching does. We must keep track of the parent of the current node, so that we can set its left or right pointer to point to the new node. We must also be careful to handle correctly the case in which the tree is initially empty. In this case, the new node must be installed as the root of the tree. The following procedure, which is based on one in Wirth (1976), takes care of these details:

FIGURE 6.22 Inserting nodes into binary search trees.

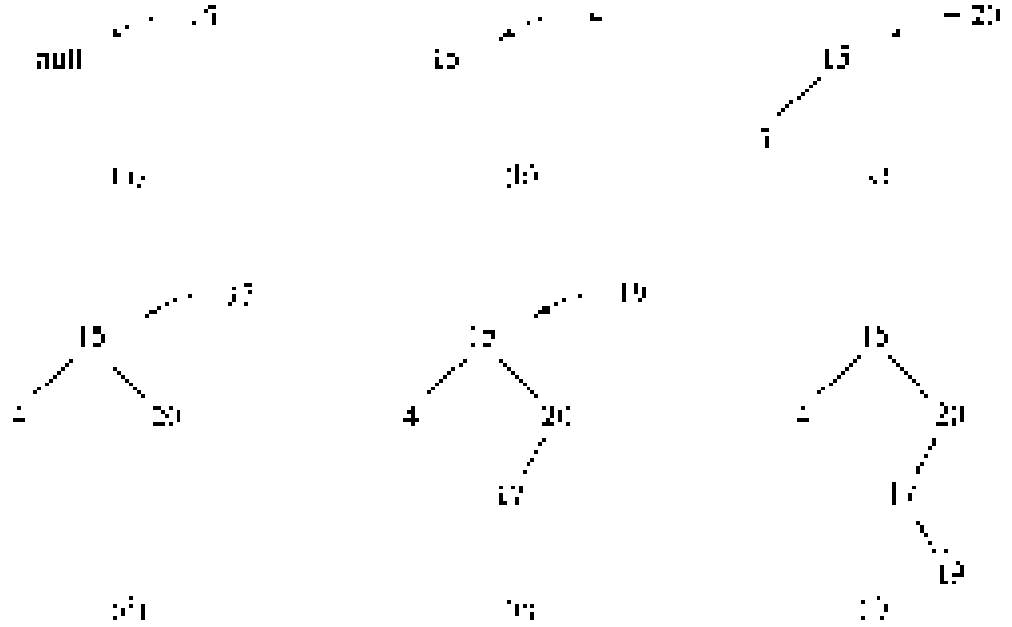


FIGURE 6.23 Implementation of the insertion algorithm.

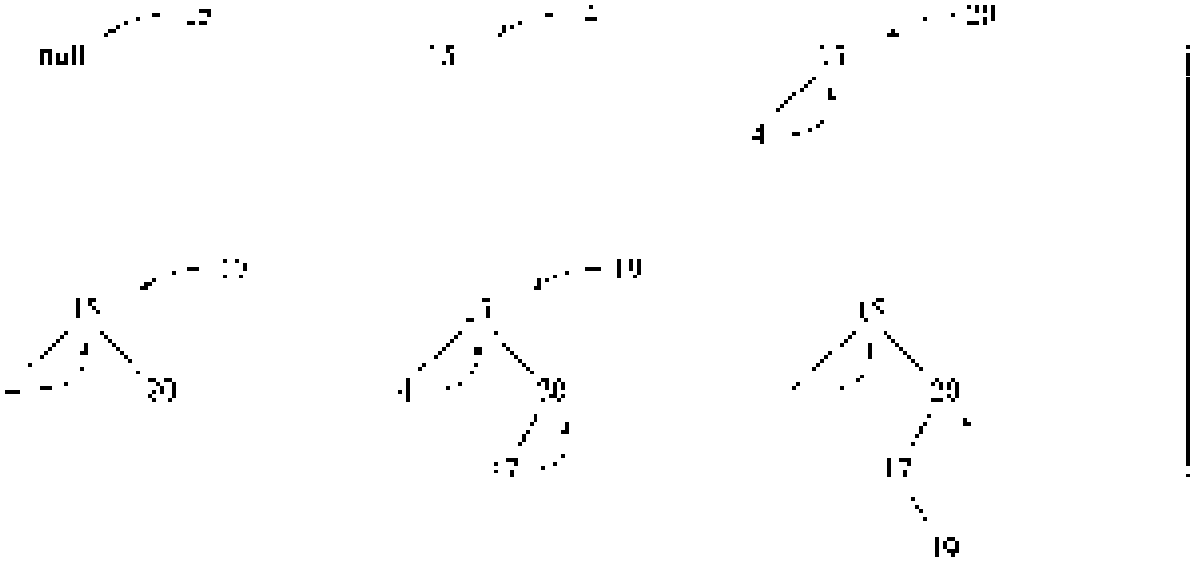
```
template<...>
void BST<...>::insert(const T& el) {
    BSTNode<T> *p = root, *prev = 0;
    while (p != 0) {          // find a place for inserting new node;
        prev = p;
        if (p->key < el)
            p = p->right;
        else p = p->left;
    }
    if (root == 0)            // tree is empty;
        root = new BSTNode<T>(el);
    else if (prev->key < el)
        prev->right = new BSTNode<T>(el);
    else prev->left = new BSTNode<T>(el);
}
```

Figure 6.24 Implementation of the algorithm to insert nodes into a threaded tree

```
template<class T>
void ThreadedTree<T>::insert(const T& x) {
    ThreadedNode<T> *p, *prev = 0, *newNode;
    newNode = new ThreadedNode<T>(x);
    if (root == 0) {                    // tree is empty;
        root = newNode;
        return;
    }
    p = root;                           // find a place to insert newNode;
    while (p != 0) {
        prev = p;
        if (p->key > x)
            p = p->left;
        else if (p->rightThread == 0)   // go to the right node only if it
            p = p->right;               // is a descendant, not a successor;
        else break;                     // avoid to the inorder successor node;
    }
    if (prev->key > x) {                // newNode is left son of
        prev->leftThread = newNode;     // the parent, the parent
        newNode->rightThread = 1;       // might become the predecessor;
        newNode->right = prev;
    }
    else if (prev->rightThread == 1) {  // if the parent of newNode
        newNode->rightThread = 1;       // is not the rightmost node,
        prev->rightThread = 0;          // make parent's successor
        newNode->right = prev->right;   // newNode's successor,
        prev->right = newNode;
    }
    else prev->right = newNode;         // otherwise it has no successor;
}
```

traversing through the consideration. The first approach does not change the threading the process. The third approach changes it, but restores it to the same condition as before it started. Only the second approach needs some preparatory operations that have to become feasible: it requires threads. These threads may be created each time before the traversal procedure starts its task and removed each time it is finished. If the traversal is performed infrequently, this becomes a viable option. Another approach is to maintain the threads in all operations on the tree when inserting a new element in the binary search tree.

FIGURE 6.25 Inserting nodes into a threaded tree.



The function for inserting a node in a threaded tree is a simple extension of `tsucc()` for regular binary search trees to adjust threads whenever applicable. This function is for inorder traversal and it only takes care of successors, not predecessors.

A node with a right child has a successor somewhere in its right subtree. Therefore, it does not need a successor thread. Such threads are needed to allow climbing the tree, not going down. If a node with no right child has its successor somewhere above it. Except for one node, all nodes with no right children will have threads to their successors. If a node becomes a new right child of another node, it inherits the successor from its new parent. If a node becomes a left child of another node, this parent becomes its successor. Figure 6.26 contains the implementation of this algorithm. The first few insertions are shown in Figure 6.25.

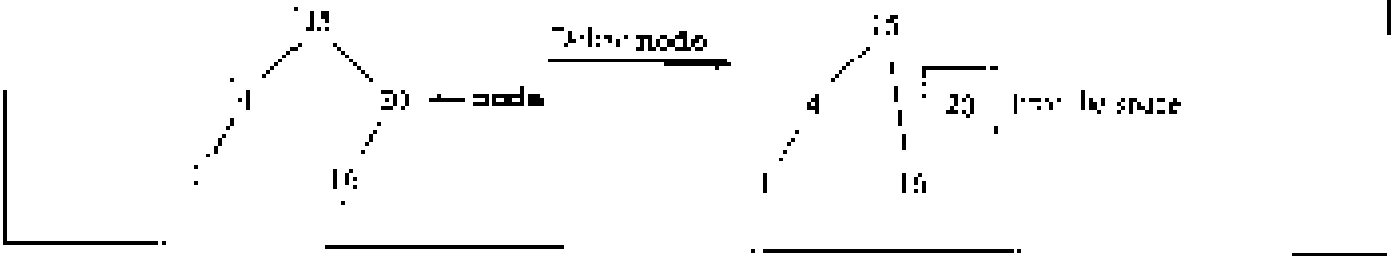
6.6 Deletion

Deleting a node is a little operation necessary to maintain a binary search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. Here are the cases to consider when deleting a node from a binary search tree:

1. The node is a leaf or has one child. This is the easiest case to deal with. The appropriate pointer of its parent is set to null and the node is disposed of by the `free` operator. Figure 6.27a.

FIGURE 6.26 Deleting a leaf.

FIGURE 6.27 Deleting a node with one child.



2. The node has one child. This case is not too complicated. The parent's pointer to the node is reset to point to the node's child, in essence, the node's children are lifted up by one level and the grandparent "grandchild" relationship is established. For example, the node containing 20 in Figure 6.27 is deleted by setting the right pointer of the parent, containing 15, to point to 20's only child, which is 16.
3. The node has two children. In this case, no one step operation can be performed since the parent's right or left pointer cannot point to both of the node's children. This section discusses two different solutions to this problem.

6.6.1 Deletion by Merging

This solution makes one of the two subtrees of the node to be deleted a subtree of the node's parent. This technique is called deletion by merging. But how can we merge these subtrees? By the nature of binary search trees, every value in the right subtree is greater than every value in the left subtree, so the best thing to do is to find in the left subtree the node with the greatest value and make it the parent of the right subtree. Symmetrically, the node with the lowest value could be found in the right subtree and made a parent of the left subtree.

The desired node is the rightmost node of the left subtree. It can be located by moving along this subtree and taking right pointers until one is encountered. This

means that the node will not have a right child, and there is no danger of violating the property of binary search trees in the operation by setting that node's right pointer to be the right subtree. It is useful to draw a picture to help you understand the logic behind the following code. Figure 6.28 depicts this operation. Figure 6.29 contains the implementation of the algorithm.

FIGURE 6.28 Summary of deleting by merging.

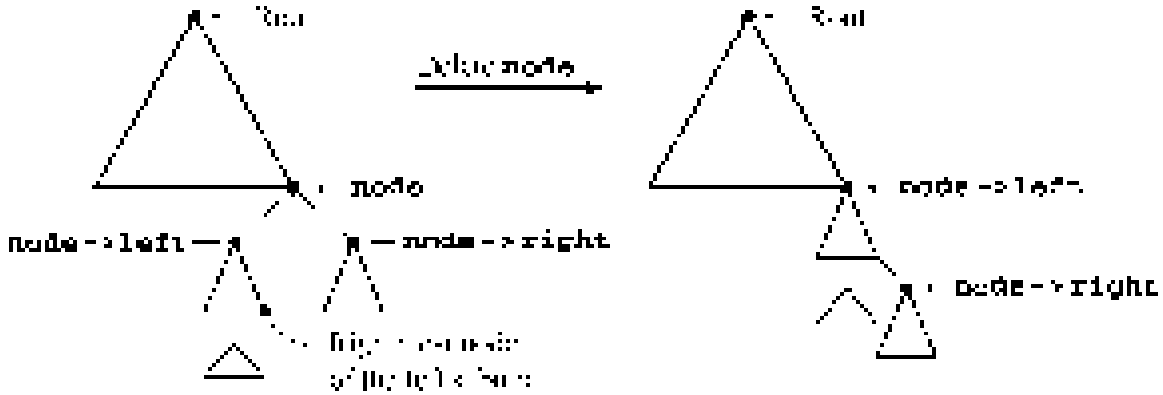


FIGURE 6.29 Implementation of algorithm for deleting by merging.

```
template<class T>
void BinaryTree<T>::deleteByMerging(BinaryTreeNode<T>* &node) {
    BinaryTreeNode<T> *tmp = node;
    if (node != 0) {
        if (node->right)                // node has a right child: its left
            node = node->right;         // child is attached to the
                                        // parent;
        else if (node->left == 0)       // node has no left child either so nothing
            node = node->right;         // is attached to its parent;
        else {                          // both left and right exist;
            tmp = node->left;           //   a node to left;
            while (tmp->right != 0)     // go down until right is free;
                tmp = tmp->right;
            tmp->right =                // it's rightmost node of the left
                node->right;            //   subtree of the node;
```

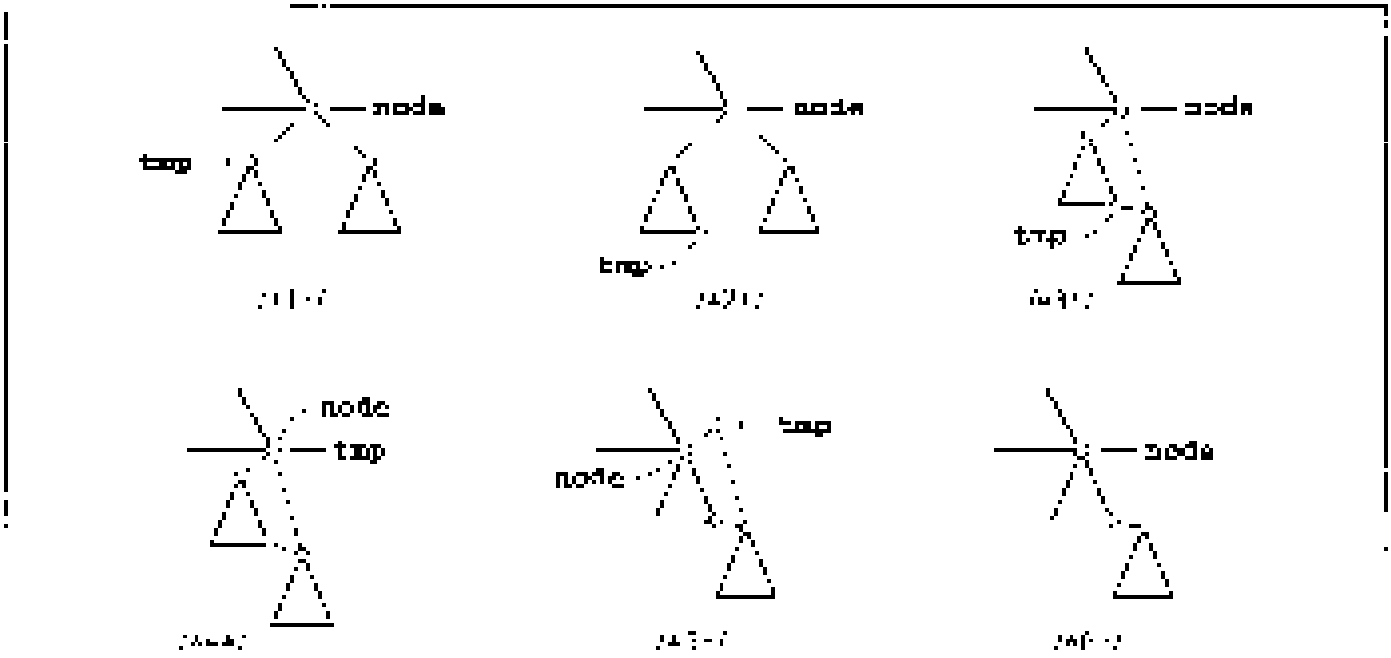
Continued

FIGURE 6.29 continued

```
                                        //    Deletion and the right subtree;
            tmp = node;                 // 4.
            node = node->left;          // 5.
        }
        delete tmp;                     // 6.
    }
}

template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0 &&
           node->key != el) {
        prev = node;
        if (node->key < el)
            node = node->right;
        else node = node->left;
    }
    if (node != 0 && node->key == el)
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    else if (root != 0)
        cout << "key " << el << " is not in the tree\n";
    else cout << "The tree is empty\n";
}
```

In my approach that findAndDeleteByMerging() function calls the auxiliary function deleteByMerging(), before returning, where findAndDeleteByMerging() assumes responsibility for deallocating the node. But using deleteByMerging() in findAndDeleteByMerging() is not necessary; it could be done directly, as in the case of deleteByCopying(), but then the code would be longer and specifically in need of keeping track of the node's parent through a pointer variable; it would also require a traversal of the tree to find the parent. Using deleteByMerging() in findAndDeleteByMerging() seems to be a good idea because the code of deleteByMerging() is used twice: Once to access the end procedure, when the node is

FIGURE 6.30 Details of deleting by merging



tool would be lost. One reason for this is the fact that search() focuses on the node's key, and findAndDelete() focuses on the node itself as an element of a larger structure, namely, a tree.

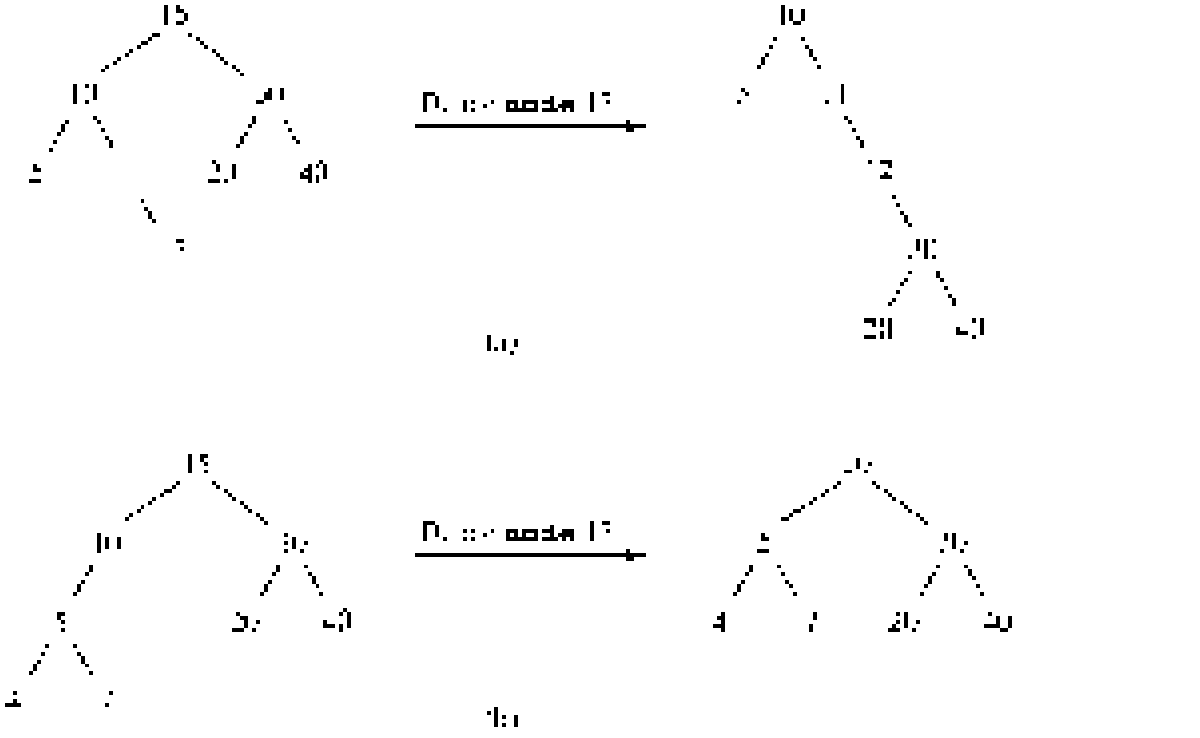
Figure 6.30 shows each step of this operation. It shows what changes are made when findAndDelete(x) is executed. The numbers in this figure correspond to numbers put in comments in the code in Figure 6.29.

The algorithm for deletion by merging may result in increasing the height of the tree. In some cases, the new tree may be highly unbalanced, as Figure 6.31a illustrates. Sometimes the height may be reduced (see Figure 6.31b). This algorithm is not necessarily inefficient, but it is certainly far from perfect. There is a need for an algorithm that does not give the tree the chance to increase its height when deleting one of its nodes.

6.6.2 Deletion by Copying

Another solution, called *deletion by copying*, was proposed by Thomas Hibbard and Donald Knuth. If the node has two children, it can be reduced to one of two simple cases: The node is a leaf or the node has only one nonempty child. This can be done by replacing the key being deleted with its immediate predecessor or successor. As already indicated in the algorithm deletion by merging, a key's predecessor is the key in the rightmost node in the left subtree (and analogically, its immediate successor is the key in the leftmost node in the right subtree). First, the predecessor has to be located. This is done, again, by moving one step to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible. Next, the key of the located node replaces the key to be deleted. And this is where one of new simple cases

FIGURE 6.31 The height of a tree can be both increased or decreased after deletion by merging



(a)

(b)

ranges are disjoint by the symmetry property of the binary search tree. If this is the case, then the successor node is guaranteed to be the leftmost node in the right subtree of the node to be deleted. Figure 6.32 illustrates this process.

Deletion by merging is a very simple algorithm, but it has a serious drawback when it is applied many times in a row. The algorithm is asymmetric and favors one of the subtrees over the other. Repeated deletions will make the tree progressively unbalanced, thereby degrading the performance of the other dictionary operations. For example, if we alternately delete the successor and the predecessor of the root node, the tree will grow steadily deeper, even though its size remains small. Figure 6.33 illustrates this process.

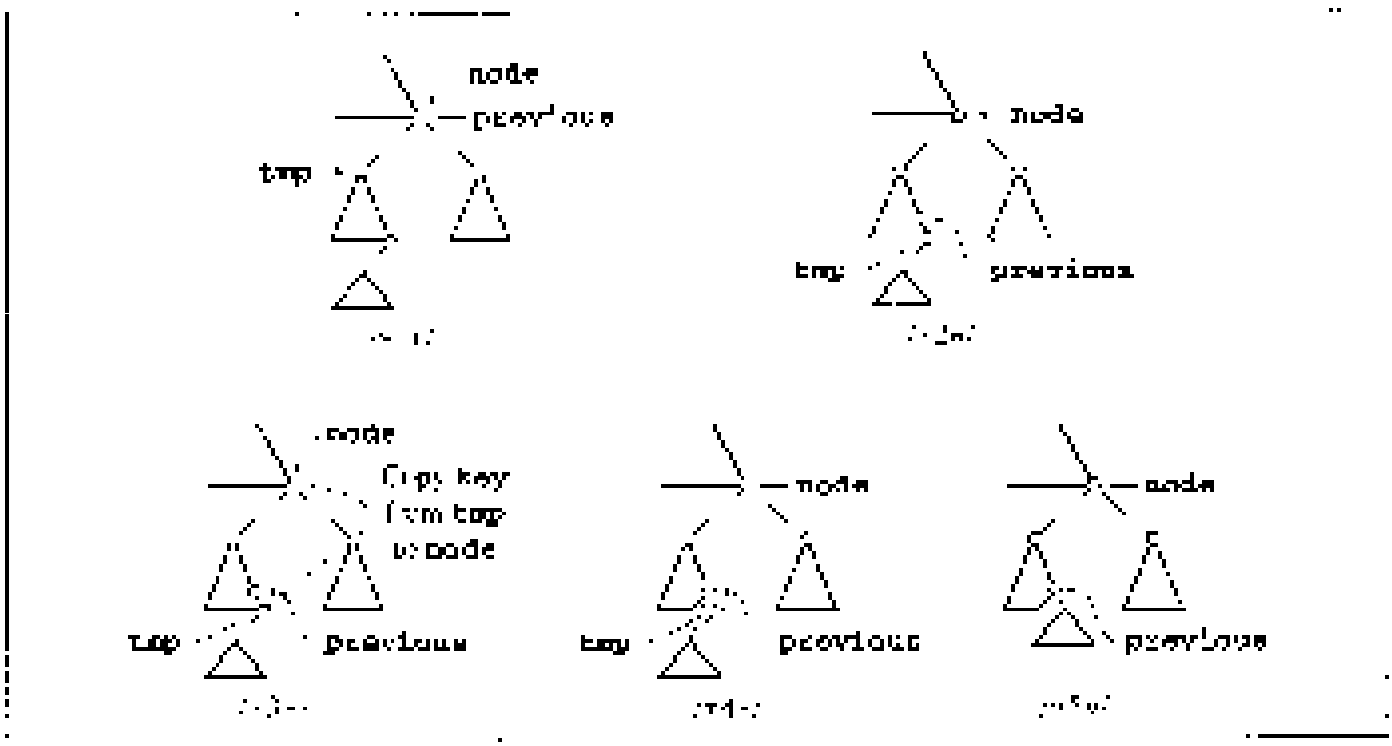
Avoiding unnecessary increases in the height of a tree is the motivation behind the deletion algorithm described by Hibbard [Hib62]. Hibbard deletion is symmetric with respect to the left and right subtrees. To delete a node k, we first locate k in the tree. If k is a leaf node, it can be deleted immediately without affecting the basic structure of the tree. If k is not a leaf node, then it must be replaced by another node in the tree. If the height of the left subtree is greater than the height of the right subtree, then k can be replaced by the largest node in the left subtree; otherwise, k can be replaced by the smallest node in the right subtree. These nodes are guaranteed to have at most one child, so they can be deleted recursively using one of the two previous cases. Figure 6.34 illustrates this algorithm.

Hibbard deletion is not perfectly balanced, but it has been analyzed both mathematically and empirically, and it appears to perform reasonably well. Eventually, however, the tree will grow too deep, and it will be necessary to rebalance it. Rebalancing algorithms are discussed in Section 6.4.

Figure 6.32 Implementing deletion by copying.

```
template<class T>
void BinaryTree<T>::deleteByCopying(BinaryTreeNode<T>* node) {
    BinaryTreeNode<T> *prev, *tmp = node;
    if (node->right == 0)                    // node has no right child;
        node = node->left;
    else if (node->left == 0)                // node has no left child;
        node = node->right;
    else {
        tmp = node->left;                    // node has both children;
        prev = node;                         // 1.
        while (tmp->right != 0) {            // 2.
            prev = tmp;
            tmp = tmp->right;
        }
        node->key = tmp->key;                // 3.
        if (prev == node)
            prev->left = tmp->left;          // 4.
        else prev->right = tmp->left;        // 5.
    }
    delete tmp;
}
```

Figure 6.33 Deleting by copying



Only for random nodes, and when symmetric deletions are used, the expected IPL becomes $O(n \log n)$. Theoretical results obtained by J. Culberson confirm these conclusions. According to Culberson, insertions and asymmetric deletions give $O(n)$, for the expected IPL and $O(\log n)$ for the average search time (average path length), whereas symmetric deletions lead to $O(\log n)$ for the average search time, and to better, $O(\log n)$ for the average IPL.

These results may be of minor importance for practical applications. Experiments show that for a 200-node binary tree, only after 1,000 sufficient insertions and symmetric deletions does the IPL become worse than in a randomly generated tree.

Theoretical results are underlying normally because of the extraordinary complexity of the problem. Anne Jonassen and Donald Knuth analyzed the performance of random insertions and deletions for a tree of only three nodes, which required using Bessel functions and bivariate integral equations, and the analysis ended as a challenge "the more difficult of all exact analyses of algorithms that have been carried out to date." Therefore, the elimination of experimental results is surprising.

6.7 BALANCING A TREE

At the beginning of this chapter, we argued in favor of trees. They are well suited to represent the binary choices of a certain decision, and the search process is much faster than in linked lists. The second argument, however, does not always hold. It all depends on what the tree looks like. Figure 6.24 shows three binary search trees. All of them store the same data, but obviously, the tree in Figure 6.24a is the best and Figure 6.24c is the worst. In the worst case, three tests are needed in the former and seven in the latter to locate an object. The problem with the trees in Figures 6.24b and 6.24c is that they are somewhat unsymmetrical, or lopsided; that is, objects in the tree are not distributed evenly to the extent that the trees in Figure 6.24b–c actually consist of two linked lists, although formally it is still a tree. Such a structure does not make sense in balanced trees.

A binary tree is height-balanced or simply balanced if the difference in height of both subtrees of any node in the tree is either zero or one, for example, for node 6 in Figure 6.24b. The difference between the two subtrees of node 6 being one is acceptable. For node 4 of the same tree, the difference is two, which means that the tree is unbalanced. For the same node 6 in Figure 6.24c, the difference is three, which is impossible, namely five. Also, a node is considered perfectly balanced if it is balanced and if both subtrees are of the same level.

Figure 6.24 shows how many nodes can be stored in binary trees of different heights. Since each node can have two children, the number of nodes on one level is double the number of nodes on the previous level (except, of course, the root). For example, if 10,000 items are to be stored in a perfectly balanced tree, then at most 14 nodes have to

FIGURE 6.34 Different binary search trees with the same information.

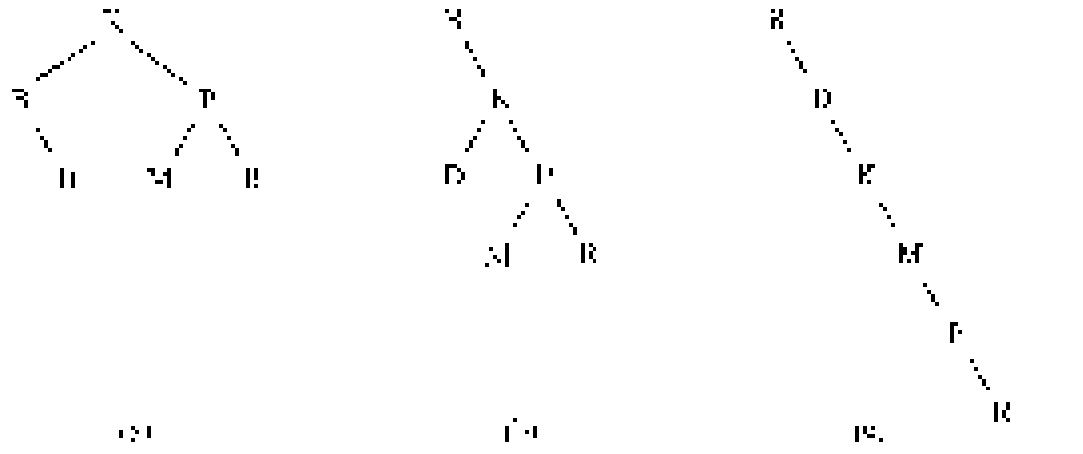
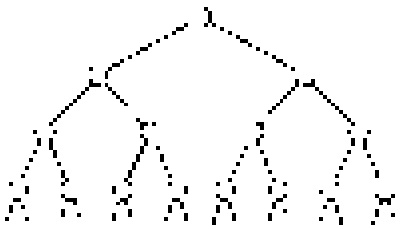


FIGURE 6.35 Maximum number of nodes in binary trees of different heights.

Height	Nodes at one level	Nodes at all levels
	$2^{1} =$	$= 2^{} - 1$
	$2^{} =$	$= 2^{} - 1$
	$2^{} = 4$	$= 2^{} - 1$
	$2^{} = 8$	$= 2^{} - 1$
	$2^{} =$	$= 2^{} - 1$
	$2^{} =$	$= 2^{} - 1$
	$2^{} =$	$= 2^{} - 1$

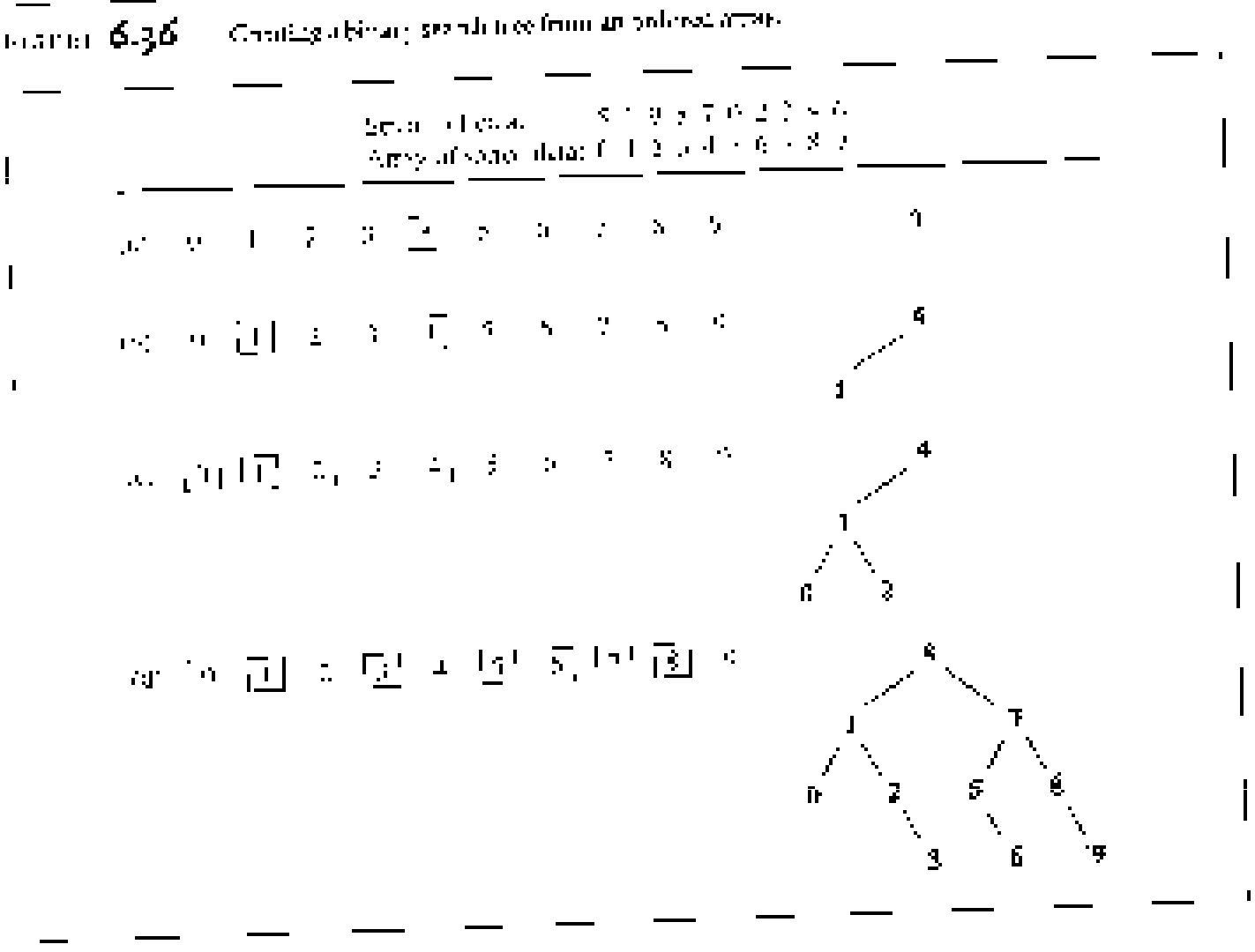


search field to locate a particular element. This is a substantial difference compared to the HEIGHT nodes needed in a linked list in the worst case. Therefore, it is worth the effort to build a balanced tree structure by an existing tree so that it is balanced.

There are a number of techniques to provide balanced binary trees. Some of them consist of algorithms for restructuring the tree when elements are inserted and deleted so

unbalanced trees. Some of them consist of reordering the data elements as they are inserted into the tree, or reordering the data after the tree is built, to make the tree more balanced. This section presents a simple algorithm for building a balanced tree.

The binary tree in Figure 5.14 is the result of a particular ordering of data values. That is, the data values could be arranged in a sorted list, such as 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,



6.7.1 The DSW Algorithm

The algorithm discussed in the previous section was somewhat inefficient in that it required a scratch array which needed to be traversed both when the tree was first created and when it was being rebalanced. In order to avoid storing or traversing a large array, Colin Day developed a method for balancing a tree in place. Although it still requires a stack, the stack is relatively small. Therefore, the algorithm is very efficient for balancing small trees. There are no recursive procedures, however, that require local variables, so the algorithm can be implemented in a relatively straightforward manner. The only algorithm that was developed by Colin Day and not previously published is the one presented by Quentin F. Stout and Bette L. Warren.

The rebalancing takes place on the way back up the tree. The algorithm is the same for both left and right subtrees. There are two types of rotations, left and right, which are symmetrical to one another. The right rotation of the node about itself is performed according to the following algorithm:

```
DSWRotateRight(node, stack)
    if node is not the root of the tree
        pop the parent of node off the stack
        make the left son of node the parent's right son
        push the parent of node onto the stack
    make the left son of node the new root
```

Figure 6.37 Right rotation of child ch about parent par.

The steps involved in the two rotational operations are shown in Figure 6.37. The first statement in the case of the rotate-in-a-right-direction, the grandparent node of child ch becomes the child of ch, whose height may or may not change. However, this exchange of children does not affect the principal property of the tree, namely, that it is a search tree. The first and the second steps of rotateRight() are needed to set suitable links in the data items, to preserve the search tree.

Basically, the DSW algorithm reconfigures an arbitrary binary search tree into a linked list that we called a backbone or vine. Then, this elongated tree is reconstructed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

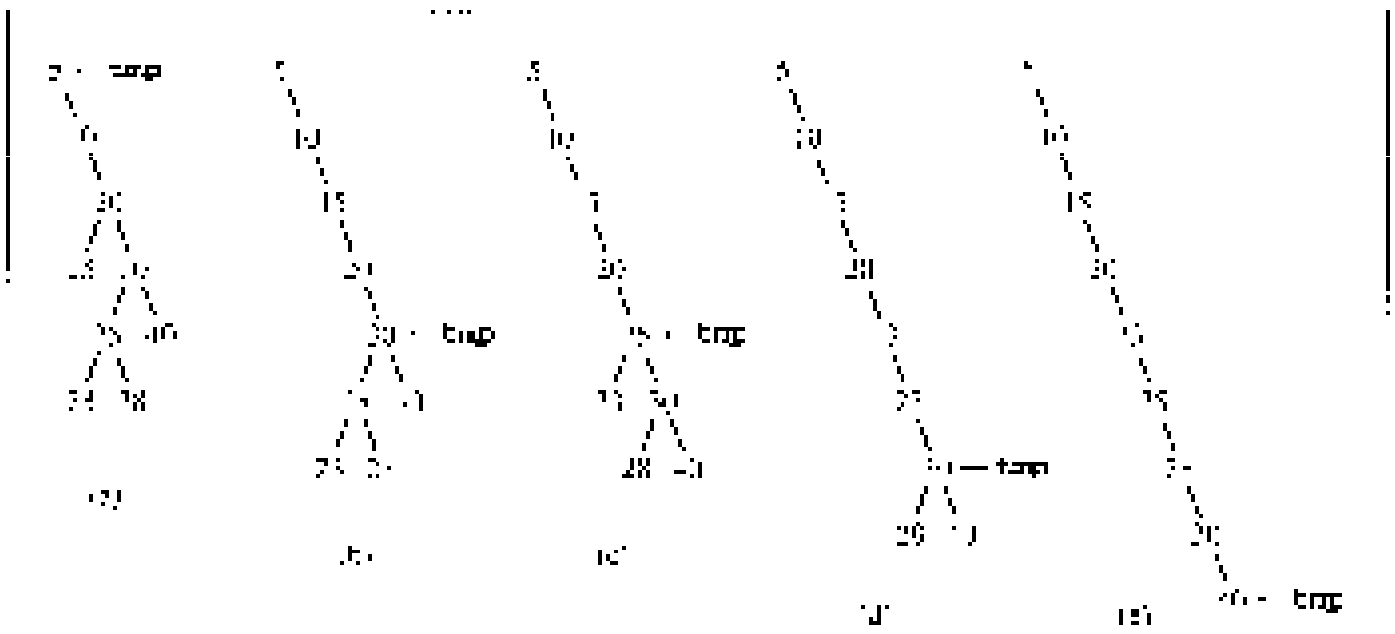
The initial backbone is created using the following routine:

```
createBackbone(theRoot, n);
  tmp = root;
  while (tmp != 0)
    if tmp has a left child
        rotate the child about tmp;  // hence the left child
                                     // becomes parent of tmp;
        set tmp to the child which just became parent
    else set tmp to its right child;
```

This algorithm is illustrated in Figure 6.38. Note that a rotation requires knowledge about the parent of tmp, so another pointer has to be maintained when implementing the algorithm.

In the best case, when the tree is already a backbone, the while loop is executed n times and no rotation is performed. In the worst case, when the root does not have a left child, the while loop executes $2n - 2$ iterations with $n - 1$ rotations performed, where n is the number of nodes in the tree. That is, the run time of the first phase is $O(n)$. In this case, for each node except the one with the smallest value, the left child of tmp is rotated about tmp. After all rotations are finished, root points to the root, and after n iterations, it descends down the backbone to become null.

FIGURE 6.38 Translating a binary search tree into a backbone.



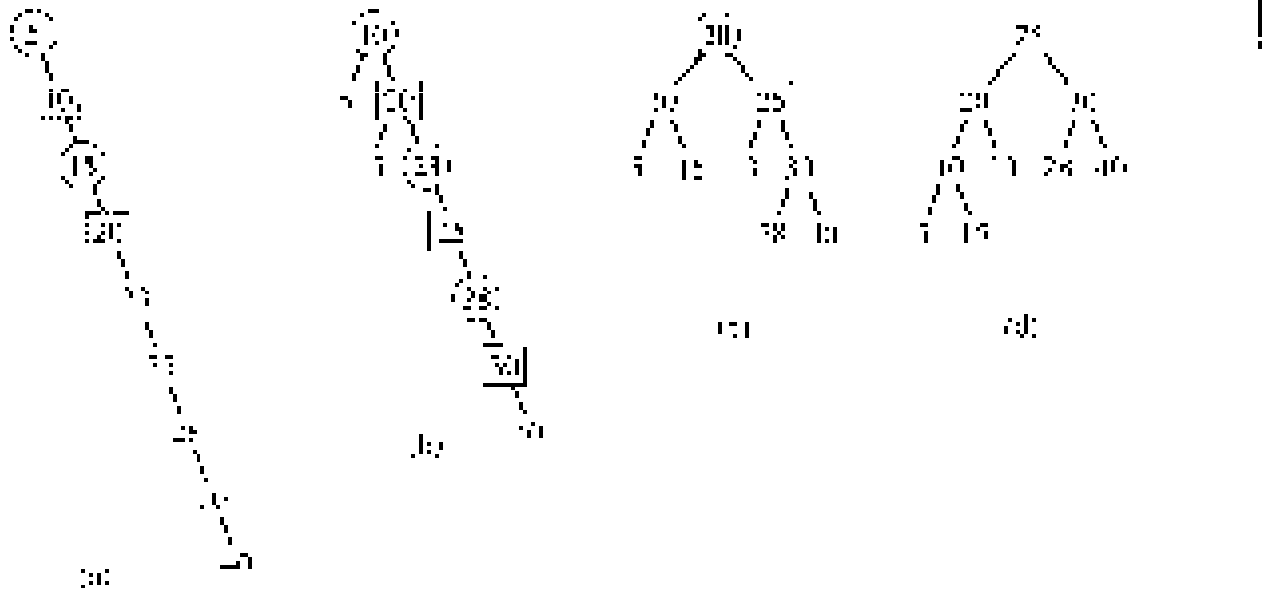
In the second pass, the backbone is traversed and rotations, but this time the tree is perfectly balanced by always moving only one node at each level. In each pass down the backbone, every second node down from a certain point is rotated about its parent. The first pass is used to count how many nodes there are in the tree, and then a number $2^{\lfloor \log(n+1) \rfloor} - 1$ of nodes is the largest complete binary tree where n is the total number of nodes. That is, the uppermost nodes are treated separately.

```
createPerfectTree(n)
  m = 2^⌊log(n+1)⌋ - 1;
  make n - m rotations starting from the top of the backbone;
  while (m > 1)
    m = m/2;
    make m rotations starting from the top of the backbone;
```

Figure 6.38 contains an example. The backbone in Figure 6.38a has nine nodes and is preprocessed by one pass outside the loop to be transformed into the backbone shown in Figure 6.38b. Now, two passes are taken. In each backbone, the nodes to be promoted by one level by left rotations are shown as squares; their parents, about which they are rotated, are circles.

To summarize the complexity of the tree building phase, observe that the number of iterations performed by the `while` loop equals

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



The number of rotations can also be expressed by the formula

The number of rotations can also be expressed by the formula

$$n - \lfloor \lg(n+1) \rfloor$$

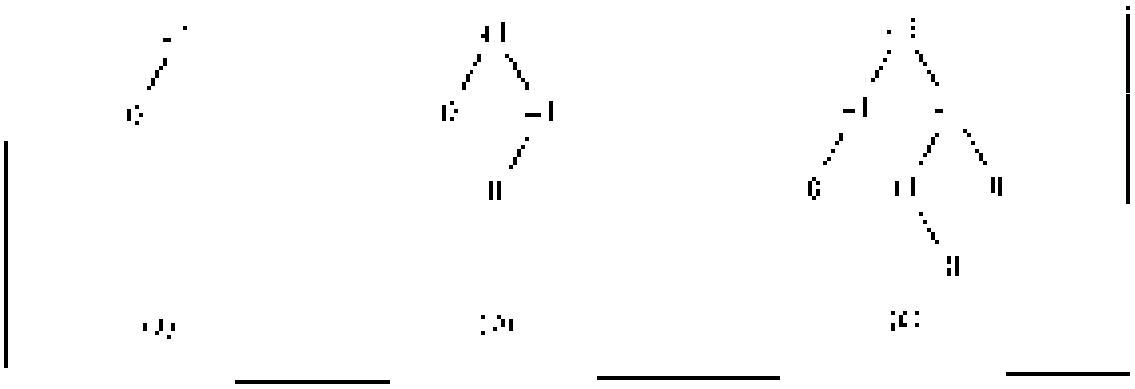
That is the number of rotations is $O(n)$. Because creating a backbone also requires at most $O(n)$ rotations, the overall algorithm to create a tree with the DSW algorithm requires $O(n)$ time in terms of rotations because it grows linearly with n, and the quantity is small and fixed amount of space.

6.7.2 AVL Trees

The previous two sections discussed algorithms which balanced binary trees globally, such that every node could have its value changed or be moved by modifying data from nodes or by reassigning pointers to point to different nodes. Such a complete rebalancing can be performed efficiently if only a perfectly balanced tree is required after an element is inserted into a sorted list or tree. One classical method has been proposed by Adel'son-Vel'skii and Landis, which we call AVL trees. One additional field has been added to the nodes of the tree, which is modified by AVL logic.

An AVL tree is a binary search tree in which the height of the left and right subtrees of every node differ by at most one. For example, all the trees in Figure 6.40 are AVL trees. Note that in a complete binary tree, which is the difference between the heights of the left and right subtrees is 0 or 1.

Figure 6.40 Examples of AVL trees.



the height of the right subtree must be both of the left subtree. Because AVL trees, all balance factors should be 1, 0, or −1. Due to this, the definition of the AVL tree is the same as the definition of the balanced tree. However, the concept of the AVL tree always implicitly includes the techniques for balancing the tree. Moreover, unlike the red-black trees previously discussed, the technique for balancing AVL trees does not guarantee that the resulting tree is perfectly balanced.

The definition of the AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions.[1] As shown by Adel'son-Vel'skii and Landis, this formula leads to the following theorem: the height of an AVL tree devoted to n internal nodes is

$$\log(n+1) \le h < 1.4404 \log(n+2) - 0.328$$

Therefore, it is bounded by $O(\log n)$; the worst-case search requires $O(\log n)$ comparisons. For a perfectly balanced binary tree of the same height, $h = \lfloor \log n \rfloor + 1$. Therefore, the search time in the worst case in an AVL tree is 44% more than in the best case tree. Empirical studies indicate that the average number of searches is much closer to the best case than to the worst and is equal to $\log n + 0.25$ for large n (Knuth 1998). Therefore, AVL trees are definitely worth studying.

If the balance factor of a node in an AVL tree becomes less than −1 or greater than 1, the tree has to be rebalanced. An AVL tree can become out of balance in four situations, but only two of them need to be analyzed; the remaining two are symmetrical. The first case describes that of inserting a node in the right subtree of the right child, as illustrated in Figure 6.41. The heights of the possible subtrees are indicated within the subtrees. In the AVL tree in Figure 6.41a, a node is inserted somewhere in

[1] Numbers generated by the recurrence formula are called Fibonacci numbers.

Figure 6.41 Balancing a tree after inserting a node in the right subtree (case 3).

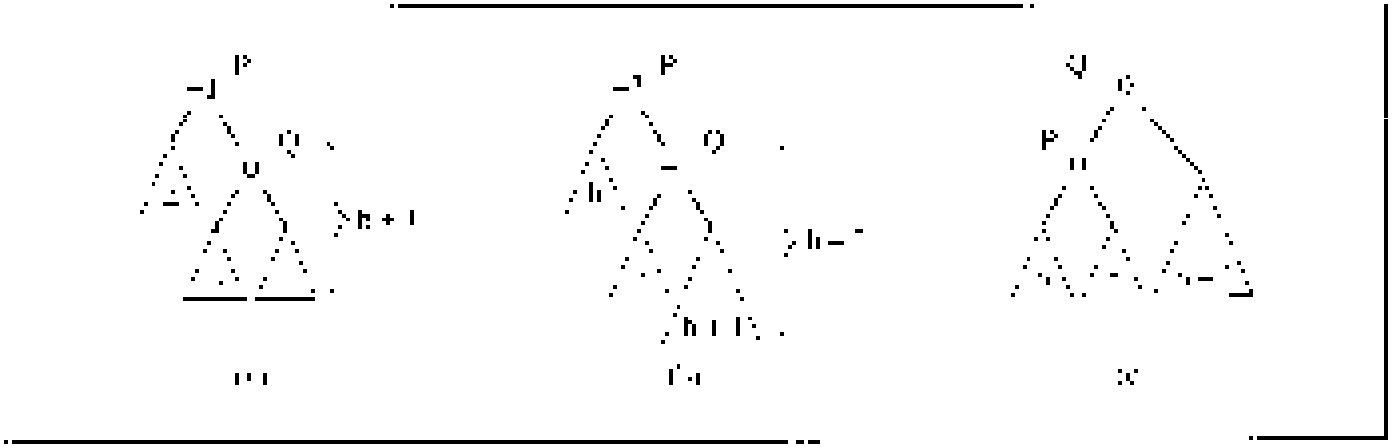
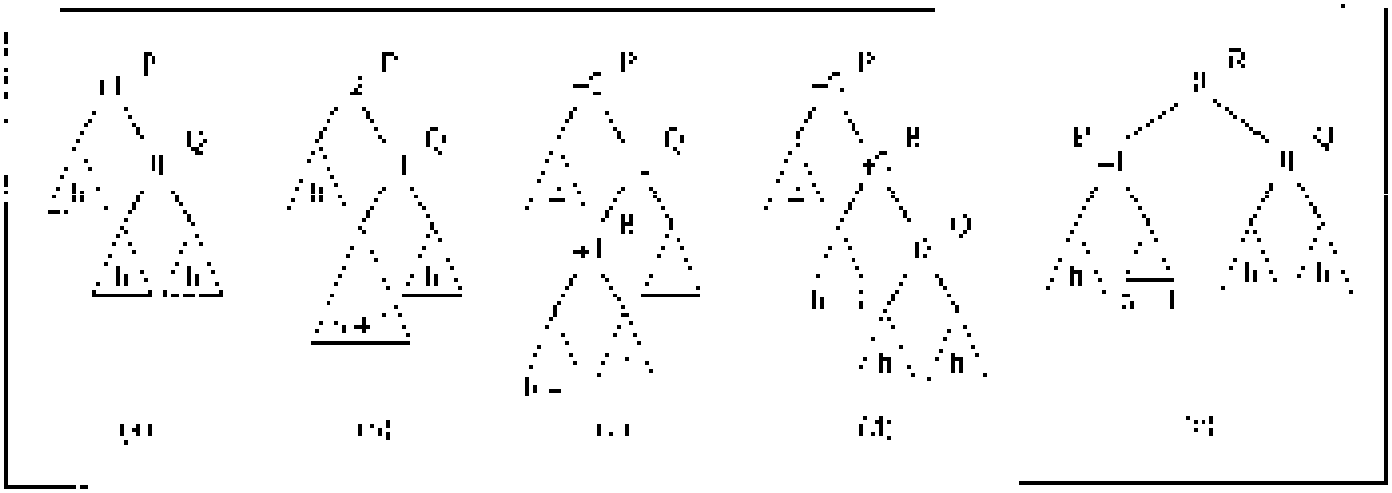


Figure 6.42 Rotating a tree after inserting a node in the left subtree (case 3).

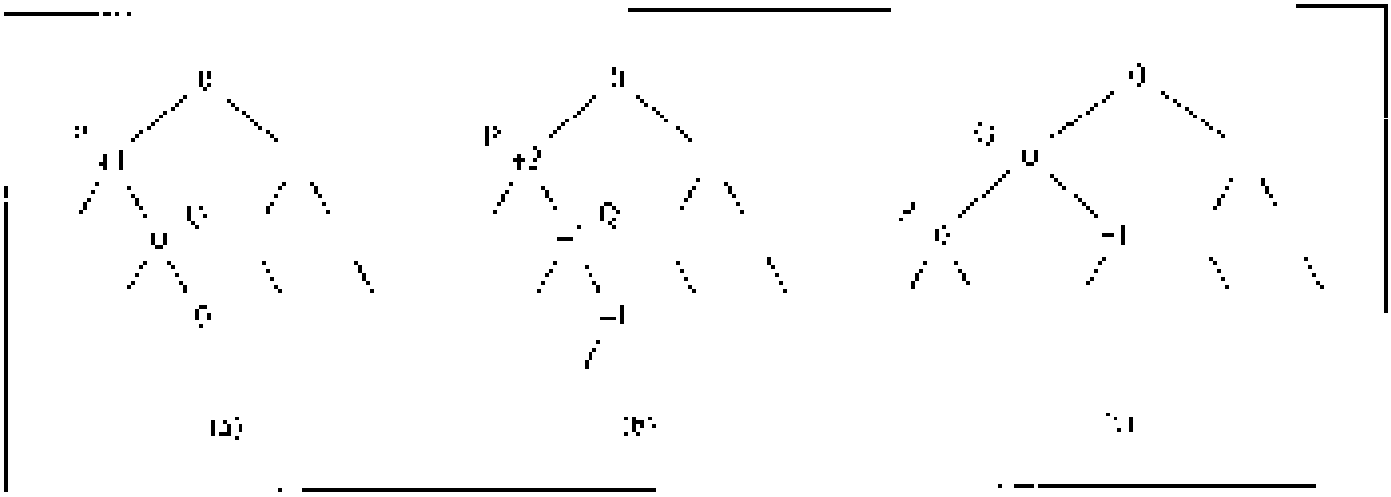


the right subtree of Q (Figure 6.41b), which disturbs the balance of the tree. To balance the tree, we can easily rectify by rotating node Q about its parent P (Figure 6.41c) so that the balance factors of both P and Q become zero, which is restored to the overall tree.

The second case, the result of inserting a node in the left subtree of the tree, is shown in Figure 6.42a. Inserting a node in the tree in Figure 6.42a, the result is shown in Figure 6.42b, and is transformed in Figure 6.42c. To bring the tree back to its original height, double rotation is performed. The balance of the tree P is restored by rotating P about node Q (Figure 6.42d) and then by rotating R about node P (Figure 6.42e).

In these two cases, the tree P is considered a stand-alone tree. However, P can be part of a larger AVL tree; it can be a child of some of the nodes in the tree. In such a case, the rotation in the tree and the balance of P is disturbed and then restored. Consequently

FIGURE 6.43 An example of inserting a new node (8) into an AVL tree (a), which requires one rotation (c) to restore the height balance.



work can be done to the predecessor(s) of P. In this example, no other nodes are affected. Note that the heights of the trees in Figures 6.43a and 6.43b are the same as the heights of the trees before insertion (Figures 6.41a and 6.42a), and are equal to $h = 2$. Thus we see that the balance factor of the parent of the new node (Q in Figure 6.43a, P in Figure 6.43b) remains the same as it was before the insertion, and no changes need to be made in the path from P and beyond to restore the balance of the entire AVL tree. The process of finding a node P for which the balance factor becomes unacceptable after a node has been inserted into the tree.

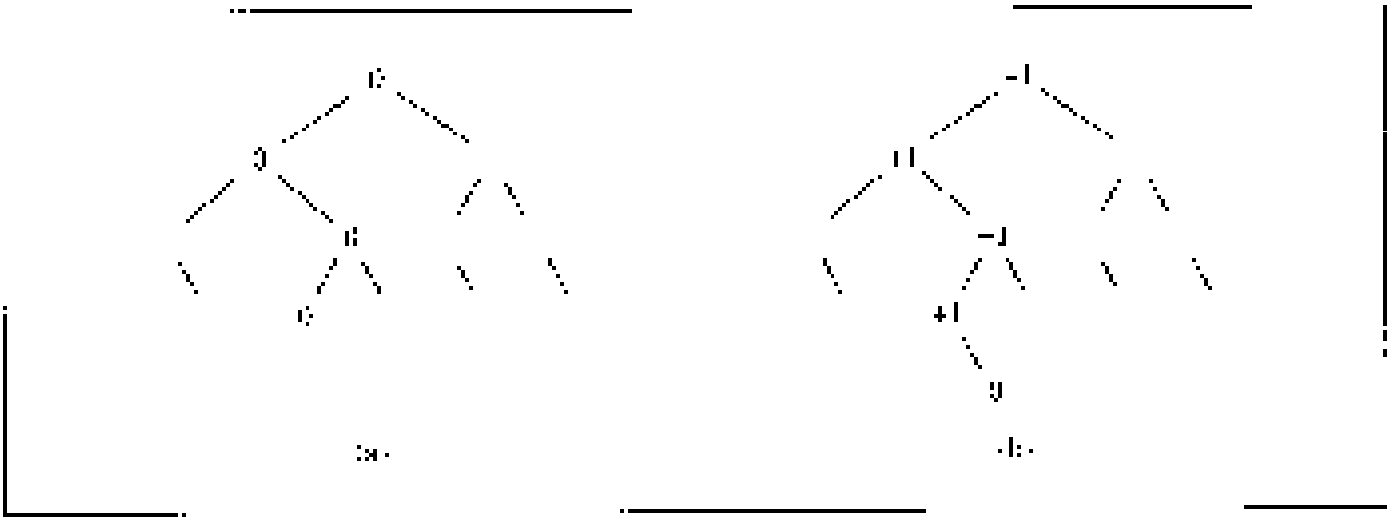
This node can be detected by moving up toward the root of the tree from the position in which the new node has been inserted and updating the balance factors of the nodes encountered. Then, if a node with a ± 1 balance factor is encountered, the balance factor may be changed to ± 2, and the first node whose balance factor is changed in this way becomes the root of a subtree or which the balance has to be restored. Note that the balance factors do not have to be updated once this node P (or the root) is reached.

In Figure 6.43a, a path is marked with one balance factor equal to +1. Insertion of a new node on the end of this path results in an unbalanced tree (Figure 6.43b), and the balance is restored by a single rotation (Figure 6.43c).

However, if the balance factors on the path from the newly inserted node to the root of the tree are all zero, then they have to be updated, but no rotations are needed for any of the encountered nodes. In Figure 6.41a, the AVL tree has a path of all zero balance factors. After a node will be appended to the end of this path (Figure 6.44a), no changes are made in the tree except for updating the balance factors of all nodes along this path.

Deletion may be more time-consuming than insertion. First we apply the lazy deletion technique to delete a node. This technique allows us to postpone the problem of deleting a node with two descendants to deleting a node with at most one descendant.

FIGURE 6.44 Unique AVL trees that store 3 nodes and have height 2, requiring no height adjustments.



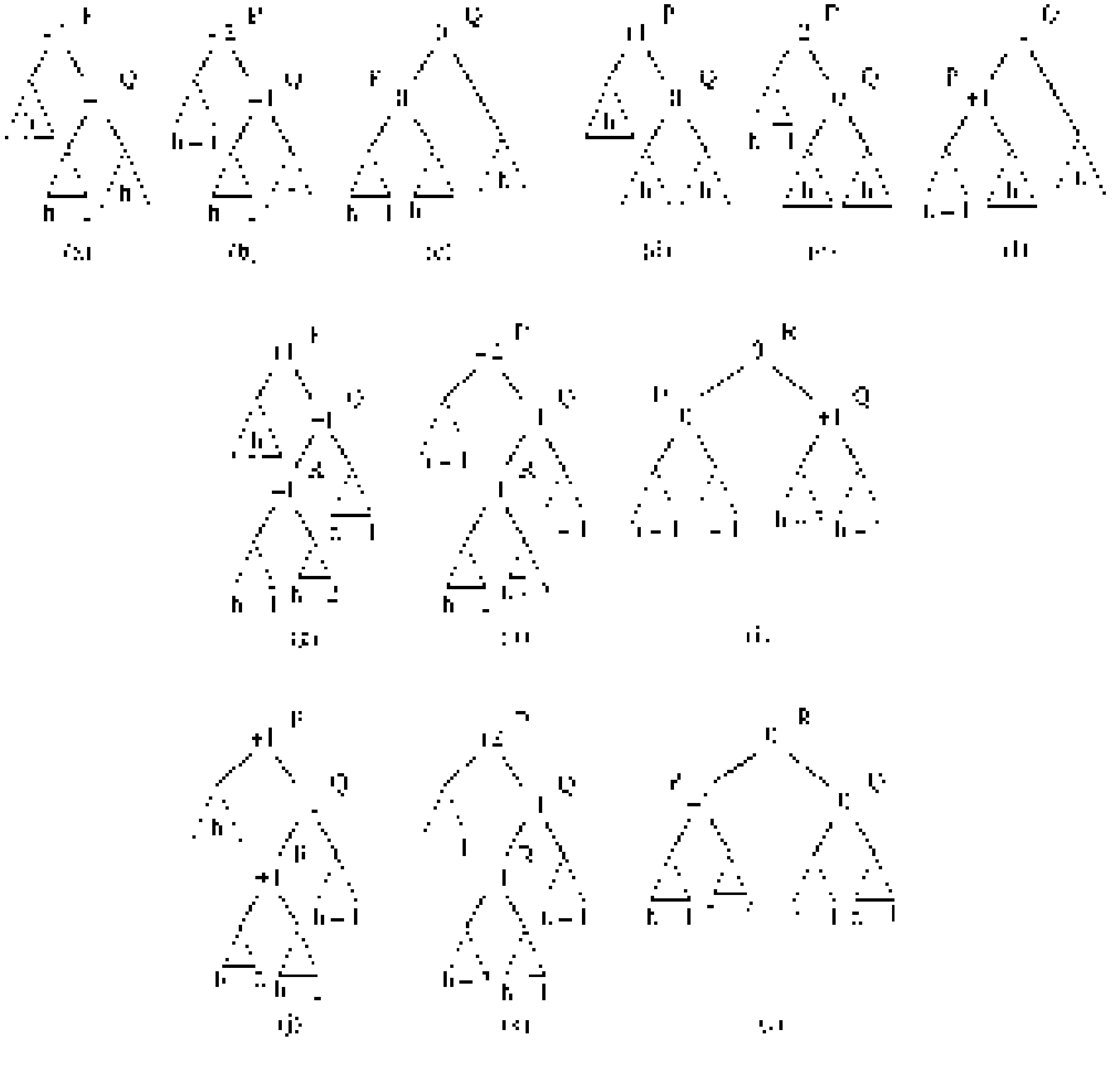
After a node has been deleted from an AVL tree, the balance factors must be updated from the parent of the deleted node up to the root. For each node on this path whose balance factor becomes ±2, a single or double rotation must be performed to restore the balance of the tree. Importantly, the rebalancing does not stop after the first node *P* is found for which the balance factor would become ±2, as is the case with insertion. This also means that deletion leads to at most $O(\lg n)$ rotations, since in the worst case, every node on the path from the deleted node to the root may require rebalancing.

Deletion of a node does not have to necessitate rebalancing solution because it may change the balance factor of its parent (by changing one of its subtrees to height 0), but it may also worsen the balance factor for the grandparent (by changing it from ±1 to ±2). We thus need only three cases that require immediate solution. There are four such cases (plus their symmetric cases). In each of these cases, we assume that the left child of node *P* is deleted.

In the first case, the tree in Figure 6.45a becomes unbalanced (node *P* in Figure 6.45b) when the node in subtree *Q* is deleted. The tree is rebalanced by rotating *Q* about *P* (Figure 6.45b). In the second case, *P* has a balance factor of 0 (Figure 6.46a), so its height does not change when its left child is deleted. Thus, no ancestor of *P* is affected by the deletion, and the process can stop after the rotation is performed (Figure 6.46b). In the third case, the left subtree of *Q* has a balance factor equal to 0 (Figure 6.47a). After deleting a node in the left subtree of *P* (Figure 6.47b), the tree is rebalanced by the same rotation as in the first case (Figure 6.47c). In this way, cases one and three can be processed together in an implementation (Exercise 6.44). Note that the height of the subtree after rotation decreases, so the ancestors of *P* may require rebalancing. Finally, we have two other cases, which are more complex. In case four, the left subtree of *Q* has a balance factor equal to +1 (Figure 6.48a). The rebalancing for this case first rotates node *R* of Figure 6.48a about node *Q* and then rotates *R* about *P* (Figures 6.48b–c). In case five, the balance factor of *Q* is equal to −1 (Figure 6.49a), so the rebalancing for this case requires two rotations, the same as for case four (Figures 6.49b–c). Cases four and five can be processed together in a program (Exercise 6.45).

The previous rebalancing rules for insertions and deletions require at most $1.44 \lg n$ (Figure 6.2) searches. Also, insertion requires one single or one double rotation, and

FIGURE 6.45 Rebalancing an AVL tree after deleting a node.



deletion can require $1.44 \log n - 2$ rotations in the worst case. But as we noted, the average case requires only about one rotation for every 5 deletions. The number of rotations in the case of deletion is not large. To be sure, deletion in the average case may be quite a bit faster than insertion. Experiments also indicate that deletion in AVL trees requires about as much time on the average as insertion. On the other hand, only 53% of the trees are balanced (Karlton et al. 1976). Therefore, it is more time-consuming to delete a node from a tree frequently than to insert a node, and it usually requires more than the theoretical average of 0.5 rotations to keep AVL trees.

AVL trees can be extended by allowing the difference in height to be 2 or 3. These trees have been shown experimentally to require fewer rotations than AVL trees, but the average search time increases by about 10 percent because the trees are not as balanced. Similarly, trees can be extended to allow differences in height of 3 or more. In each case, the amount of restructuring decreases, but the average search time increases. The optimum seems to be the AVL tree.

$$h_{\max} = \begin{cases} \text{[illegible]} & \text{if } \Delta = 2 \\ \text{[illegible]} & \text{if } \Delta = 3 \end{cases}$$

An experiment indicates that the average number of visited nodes increases by one half in comparison to pure AVL trees ($\Delta = 1$), but the amount of restructuring can be decreased by a factor of 10.

6.8 Self-Adjusting Trees

The main concern in balancing trees is to keep them from becoming lopsided and, ideally, to allow access to all elements at about the same level. Therefore, if a newly arriving element endangers the tree balance, the problem is immediately rectified by restructuring the tree locally (the AVL method), or by reconstructing the tree (the DSW method). However, we may question whether or not such a restructuring is always necessary. Binary search trees are used to insert, remove, and delete elements quickly, and the speed of performing these operations is the issue, not the shape of the tree. Performance can be improved by balancing the tree, but this is not the only method that can be used.

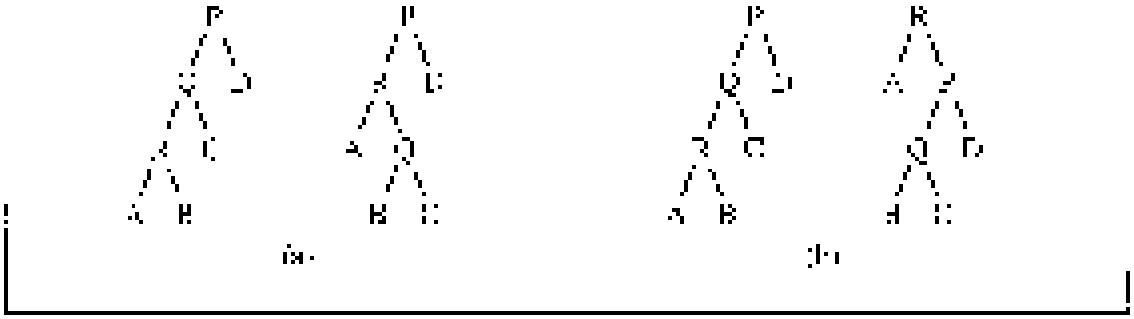
A totally opposite approach begins with the observation that not all elements are used with the same frequency. For example, if an element on the sixth level of the tree is used only extremely rarely, then the execution of the entire program is not greatly influenced by accessing this element. However, if this same element is constantly being accessed, then it makes a big difference whether it is on the tenth level or close to the root. Therefore, the strategy in self-adjusting trees is to restructure trees only by moving up the tree those elements that are used more often, creating a kind of "priority tree." The frequency of accessing nodes can be determined in a variety of ways. Each node can have a counter field which records the number of times the element has been used for any operation. Then the tree can be scanned to move the most frequently accessed elements toward the root. In a less sophisticated approach, it is assumed that a relatively new node is accessed has a good chance of being accessed again soon. Therefore, it is moved to the root. No restructuring is performed for new elements. This assumption may lead to promoting elements which are occasionally accessed, but the overall tendency is to move frequently accessed elements with a high frequency of access and for the most part these elements will populate the first few levels of the tree.

6.8.1 Self-Restructuring Trees

A strategy proposed by Irving Allen and Ian Munro and by James Bitner consists of two possibilities:

1. Single rotation: Rotate a child about its parent if an element in a child is accessed unless it is the root (Figure 6.4a).
2. Moving to the root: Repeat the child-parent rotations until the element being accessed is at the root (Figure 6.4b).

FIGURE 6.46 Restructuring a tree by using a single rotation or double rotation when accessing node A.



Using the single rotation strategy, frequently accessed elements are eventually moved up close to the root so that later accesses are faster than previous ones. In the move to the root strategy it is assumed that the element being accessed has a high probability to be accessed again, so it penetrates right away up to the root. Even if it is not used in the next access, the element remains close to the root. These strategies, however, do not work very well in unfavorable situations, when the binary tree is elongated as in Figure 6.47. In this case, the shape of the tree improves slowly. Nevertheless, it has been determined that the cost of moving a node to the root converges to the cost of accessing the node in optimal trees times 2 ln 2; that is, it converges to $2 \ln 2 \lg n$. The result holds for any probability distribution, that is, independently of the probability that a particular request is issued. However, the average search time when all requests are equally likely is for the single rotation technique equal to $\frac{n+1}{2}$.

6.8.2 Splaying

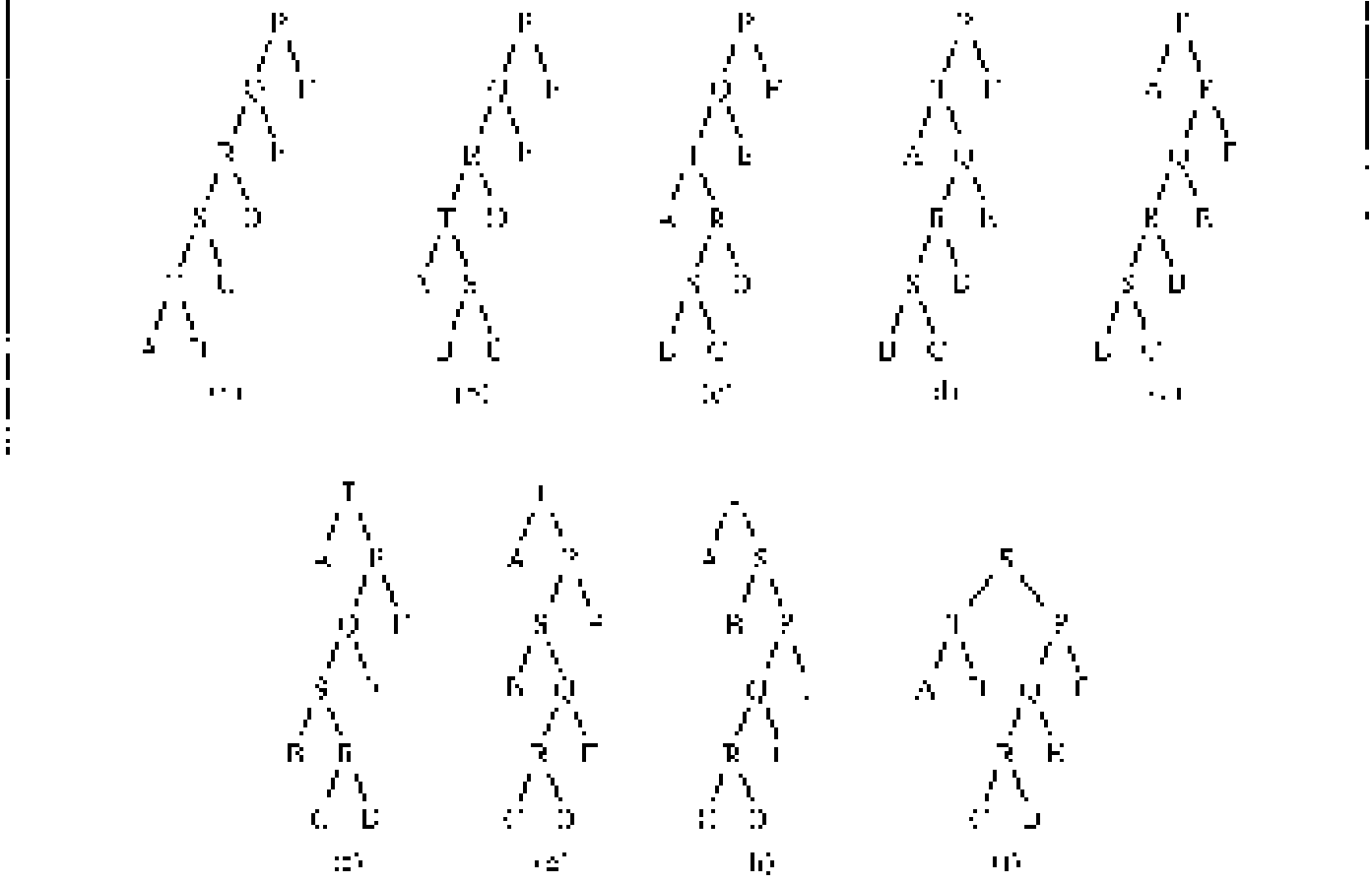
A modification of the move to the root strategy is called splaying, which applies single rotations in pairs in an order depending on the links between the child, parent, and grandparent (Sleator and Tarjan, 1985). First, three cases are distinguished depending on the relationship between a node A being accessed and its parent Q and grandparent P (if any) nodes.

Case 1: Node A's parent is the root.

Case 2: Homogeneous configuration: Node A is the left child of its parent Q, and Q is the left child of its parent P; or A and Q are both right children.

Case 3: Heterogeneous configuration: Node A is the right child of its parent Q, and Q is the left child of its parent P; or A is the left child of Q, and Q is the right child of P.

FIGURE 6.47 (a)–(f) Moving element T to the root and (g)–(j) moving element A to the root.



The configuration in which node T is processed is shown at the top of the next page as follows:

```
splaying(x,root)
    while (x != root)
        if (x's parent is the root)
            perform a single rotation, and update root (see Figure 6.45)
        else if (x is in the same configuration with its grandparent)
            perform a double rotation (zig-zig or zag-zag)
            rotate(x's parent); rotate(x);
        else if (x is in the "opposite" configuration with its grandparent)
            /* x will be in the grandparent */
            perform a double rotation (zig-zag or zag-zig)
            rotate(x); rotate(x);
```

Let T_{root} be the tree with root root as pictured in Figure 6.49, where the two subtrees of root are unspecified. Let T_x be the subtree rooted at node x. The shape of T_{root} is unchanged in the splay operation. The splay operation on node x is accomplished by a sequence of rotations as pictured in Figure 6.50.

FIGURE 6.48 Examples of splaying.

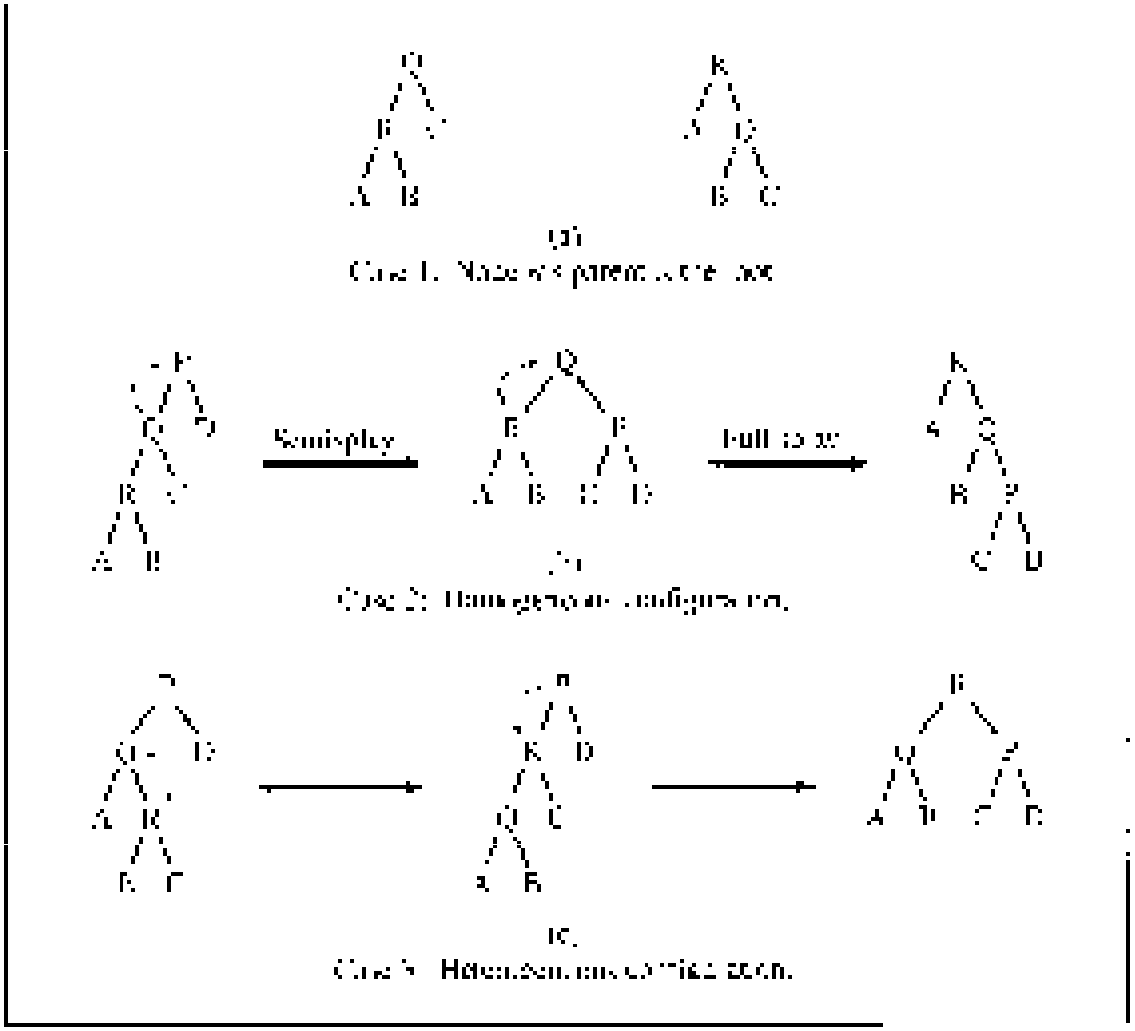
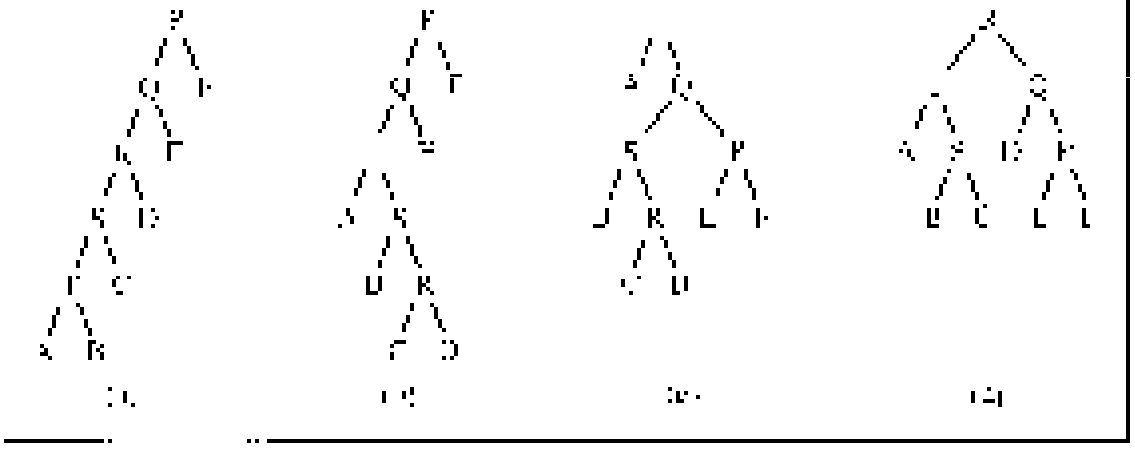


FIGURE 6.49 Restructuring a tree with splaying after accessing J and then K.



Although splaying is a comb nation of two rotations except when near the root, these rotations are not treated in the bottom-up fashion of a self-adjusting tree. For the homogeneous case (left and right rotations, for the parent and the grandparent of the node being accessed are carried out, and only after the node and its ancestors are rotated. For the heter

Until parent(p) = root(p) = nil, p becomes a real parent and the only [illegible] are modified. Now because nil(p) = root(p),

$$\text{amCost}_{ij}(x) = 1 + \text{rank}_j(p) + \text{nil}_{ij}(p)$$

and because nil(parent(x)) = root(x),

$$\text{amCost}_{ij}(x) = \text{nil}_{ij}(x) + \text{rank}_j(x).$$

Case 2: Two rotations are performed during a homogeneous zig-zig. As before, number 1 represents the actual cost of one splaying step.

$$\text{amCost}_{ij}(x) = 1 + \text{parent}(x) - \text{rank}_i(x) + \text{rank}_j(\text{parent}(x)) - \text{rank}_i(\text{parent}(x)) + \text{rank}_j(\text{grandparent}(x)) - \text{rank}_i(\text{grandparent}(x))$$

Because rank(x) = rank_j(parent(x)),

$$\text{amCost}_{ij}(x) = 1 + \text{rank}_j(x) - \text{rank}_i(\text{parent}(x)) + \text{rank}_j(\text{parent}(x)) - \text{rank}_i(\text{grandparent}(x))$$

Because rank_j(parent(x)) < rank_j(parent(parent(x))),

$$\text{amCost}_{ij}(x) < 1 + \text{rank}_j(x) - \text{rank}_i(\text{parent}(x)) + \text{rank}_j(\text{parent}(\text{parent}(x))) - \text{rank}_i(\text{grandparent}(x))$$

and because rank_j(x) > rank_i(parent(x)) that is, rank_j(parent(x)) < rank_j(x),

$$\text{amCost}_{ij}(x) < 1 + \text{rank}_j(x) - \text{rank}_i(\text{parent}(x)) + \text{rank}_j(\text{parent}(\text{parent}(x))) - \text{rank}_i(\text{grandparent}(x))$$

To eliminate number 1, consider the inequality rank_j(x) < rank_j(parent(x)), that is, rank_j(parent(x)) − rank_j(x) > 1, and add it to the above:

$$\text{amCost}_{ij}(x) < \text{rank}_j(\text{parent}(x)) - \text{rank}_i(x) + \text{rank}_j(\text{parent}(\text{parent}(x))) - \text{rank}_i(\text{grandparent}(x))$$

$$\text{amCost}_{ij}(x) < \text{rank}_j(\text{grandparent}(x)) - \text{rank}_i(x) + \text{rank}_j(\text{parent}(x)) - \text{rank}_i(x)$$

and because rank(x) = rank_i(parent(x)),

$$\text{amCost}_{ij}(x) < \text{rank}_j(\text{grandparent}(x)) - \text{rank}_i(x) + \text{rank}_j(x) - \text{rank}_i(x)$$

Case 3: Two rotations are performed during a heterogeneous splay. The only difference is that a zig-zig multiplies the inequality that rank_j(grandparent(x)) < rank_j(x) and rank_j(parent(x)) < rank_j(x) instead of rank_j(parent(x)) < rank_j(x), which ends the same as Case 2.

The total amortized cost of access(x) equals the sum of amortized costs of all the splaying steps performed during this access. If the number of steps equals s, then we get one plus the last step plus s − 2 zig-zig or zig-zag steps. Hence we get

$$\text{amCost}(x) = \sum_{i=1}^{s} \text{amCost}_i(x) = \sum_{i=1}^{s} \text{amCost}_{ij}(x) = \text{nil}_{ij}(x)$$

$$< \sum_{i=1}^{s-2} \text{rank}_j(x) - \text{rank}_i(x) + \text{rank}_j(x) - \text{rank}_i(x) + 1$$

Let us rank(x) = rank_1(x).

$$\text{amortized cost} = \sum_{i=1}^{n} \text{actual cost}(i) = \text{actual cost}(1) + \text{actual cost}(2) + \cdots + \text{actual cost}(n)$$

$$= \text{actual cost}(1) + \text{actual cost}(2) + \cdots + \text{actual cost}(n) = \text{actual cost}(1) + 1 + \cdots + 1 = O(\lg n)$$

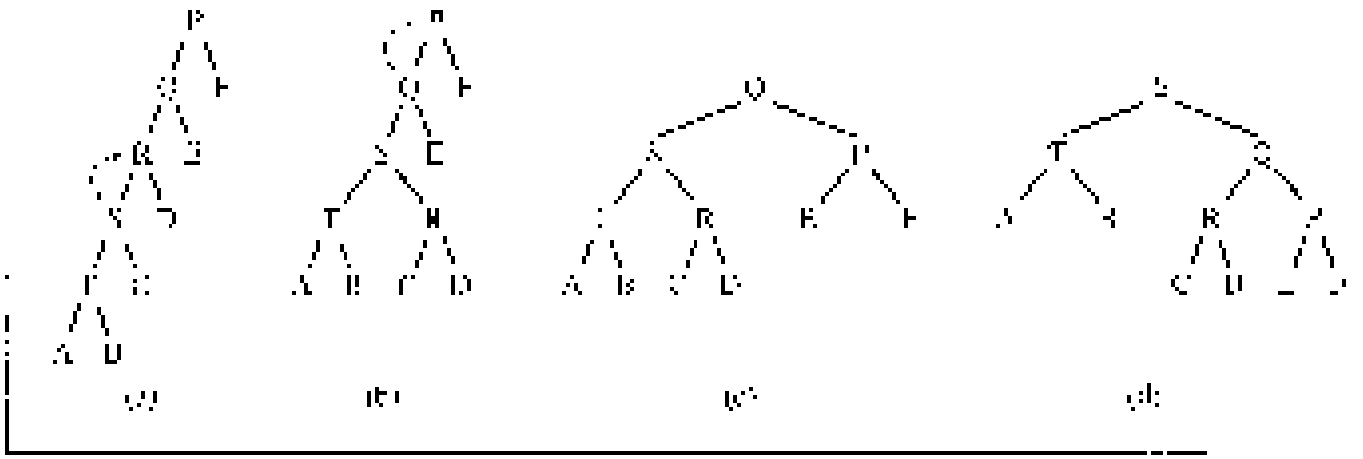
This indicates that the amortized cost of an access to a node in a tree that is restructured with the splaying technique equals $O(\lg n)$, which is the same as the worst case in balanced trees. However, to make the comparisons more adequate we should compare a sequence of n accesses to nodes rather than one access because with the amortized cost one isolated access can still be on the order of $O(n)$. The discussion above applies splaying is thus comparable to that of a balanced tree for a sequence of accesses and equals $O(n \lg n)$.

Splaying is a strategy focusing more on the elements rather than the shape of the tree. It may perform well in situations in which some elements are used much more frequently than others. If elements near the root are accessed with about the same frequency as elements on the lowest levels, then splaying may not be the best choice. In this case, a strategy that increases balancing the tree rather than frequency is better. A modification of the splaying method is a more viable option.

Semisplaying is a modification that requires only one rotation for a bottom-to-top path and no rotations if starting with the parent of the accessed node. It is illustrated in Figure 6.49b after k is accessed in parent Q of accessed node P and splaying continues with Q. As with k, knowledge of k's exact position is not performed, as would be the case for splaying.

Figure 6.50 illustrates the advantages of semisplaying. The elongated tree from Figure 6.48a becomes more balanced with semisplaying after accessing T (Figure 6.50a) and after the accesses again in the order in Figure 6.50d has nearly the same number of levels as the tree in Figure 6.48a. It may have one more level. For T was a subtree higher than any of its brothers (A, B, C, or D). For implementation of this technique, see the exercises at the end of the chapter.

Figure 6.50 (a) Accessing T and restructuring the tree with semisplaying; (b) accessing T again.



6.9 Heaps

A partial ordering can be represented by a binary tree if it satisfies two properties:

1. The value of each node is no less than the values stored in each of its descendants.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

Together, these two properties define what we call a *heap*. The first property is usually described as "greater," but we use the definition here to facilitate our later discussion. The root of a max-heap contains the largest element, and the root of each subtree contains the largest element in that subtree. Because the second property is such a severe restriction on the shape of a tree, the first property is also very restrictive: Only a few of the elements can be compared directly to the root element. Only a few more can be compared to the elements one level down, and so on. The second condition limits the number of nodes in the tree to a power of 2.

The trees in Figure 6.51(a) and (b) are both heaps. The tree in Figure 6.51(c) violates the first property, and the tree in Figure 6.51(d) violates the second.

Interestingly, heaps can be implemented as arrays. For example, the array containing [19, 6, 10, 3, 2, 4, 1] represents the heap tree in Figure 6.52. The elements are placed in sequential locations in the array as they appear in the tree, level by level, from left to right. The second property ensures that the array is packed, with no gaps. Now, a heap can't just be any array because arbitrary values will not satisfy the ordering

Because [illegible] heap [illegible] can be stored [illegible] $\frac{1}{2}$

Figure 6.51 Examples of (a) heaps and (b–d) nonheaps

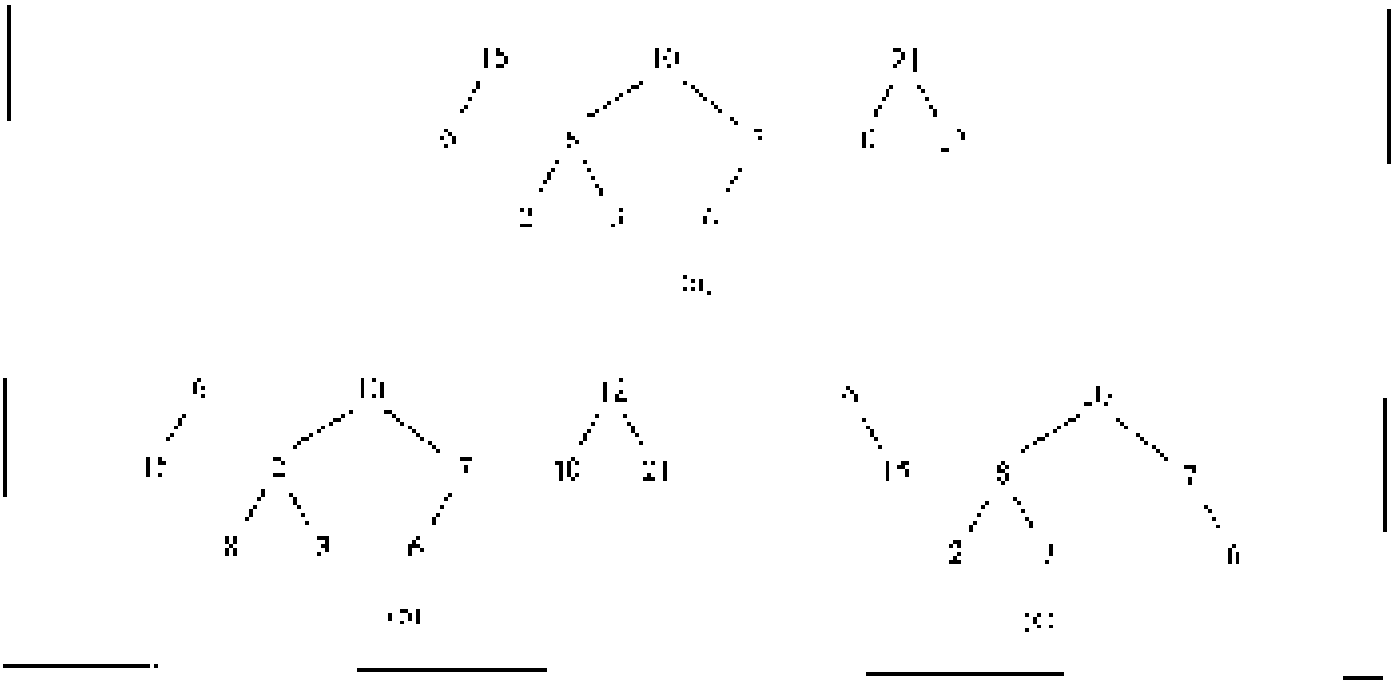


FIGURE 6.52 The array [24 65 13 15 68 32] seen as a tree.

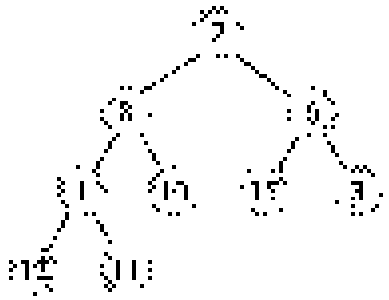
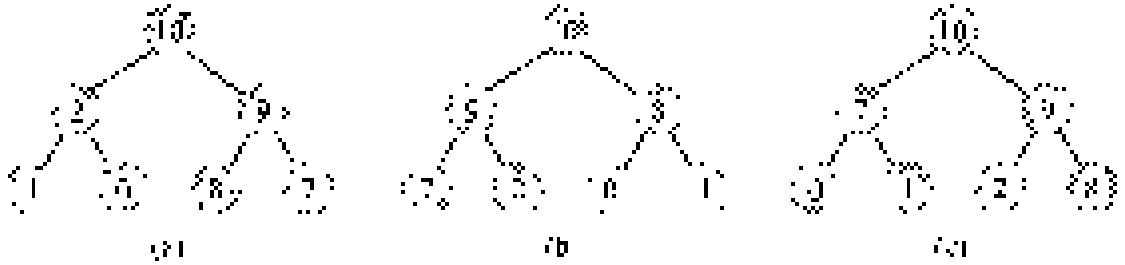


FIGURE 6.53 Different heaps constructed with the same elements



(a) (b) (c)

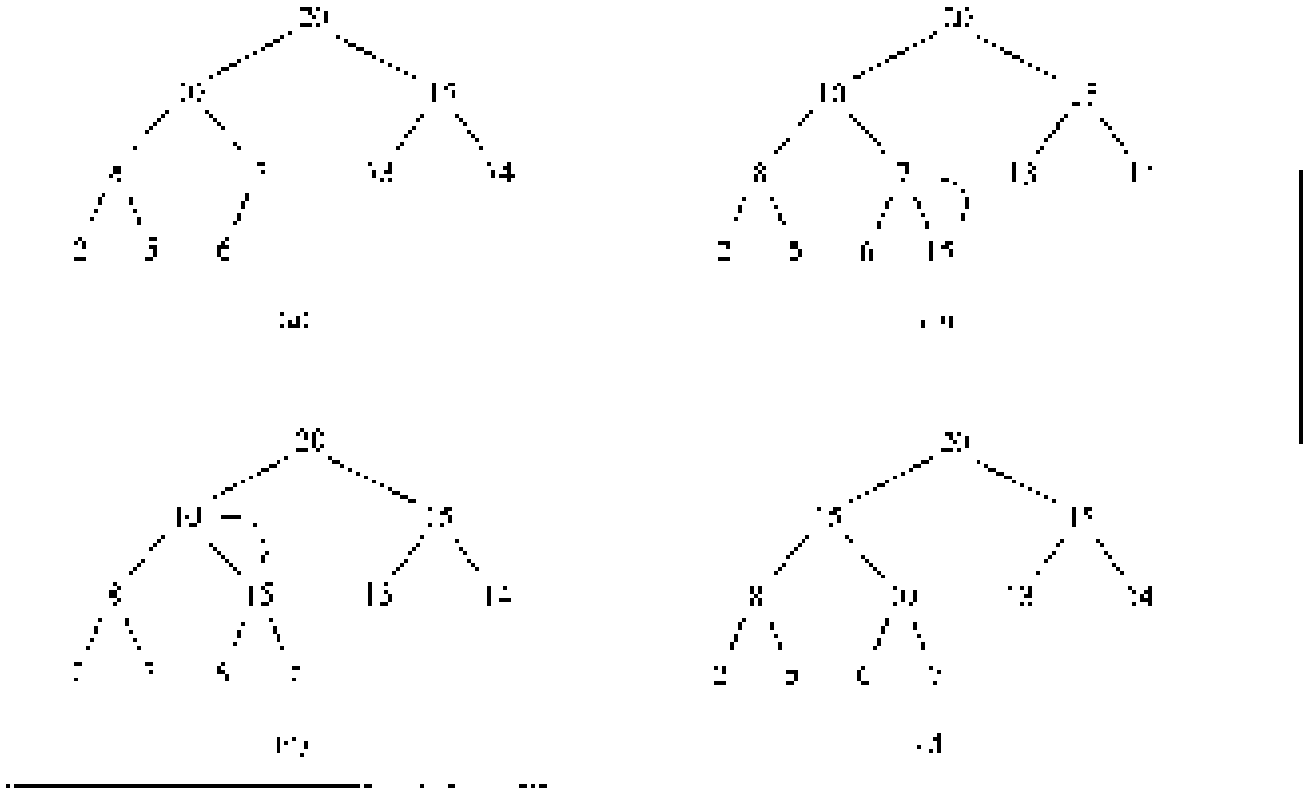
$$\text{swap}[i] = \text{heap}[n - i + 1], \text{ for } 0 \le i \le \frac{n-1}{2}$$

For instance, for the heap pictured in Figure 6.52, we know only that each element in the left subtree of the root is less than or equal to the root, but we don't know whether all the elements in the left subtree are less than or equal to the elements in the right subtree. Figure 6.53 shows three different heaps containing the same elements. Of the three trees in Figure 6.53, only the first one (the leftmost) is a binary search tree. The order of the elements in a binary tree is determined by the order in which they are inserted and deleted. In a heap, however, the shape of the tree is constrained by the number of elements in the heap, and the value of the elements is constrained by the values of their children. There is no relationship between the values of siblings or cousins in a heap.

6.9.1 Heaps as Priority Queues

A heap is an excellent way to implement a priority queue. We used linked lists to implement priority queues in Section 4.7, but we found that the complexity was expressed in terms of $O(n)$. Performance of heaps is usually $O(\log n)$. The heap is always a binary tree that is perfectly balanced, so the worst case is $\log n$ levels deep. This choice is very promising. Therefore, heaps can be used to implement priority

Figure 6.54 Enqueuing an element to a heap.



queues. To this end, however, two procedures have been implemented to enqueue and dequeue elements in a priority queue.

To enqueue an element, the element is added at the end of the heap in the first free leaf. Restoring the heap property in the case of enqueuing is achieved by moving the new leaf toward the root.

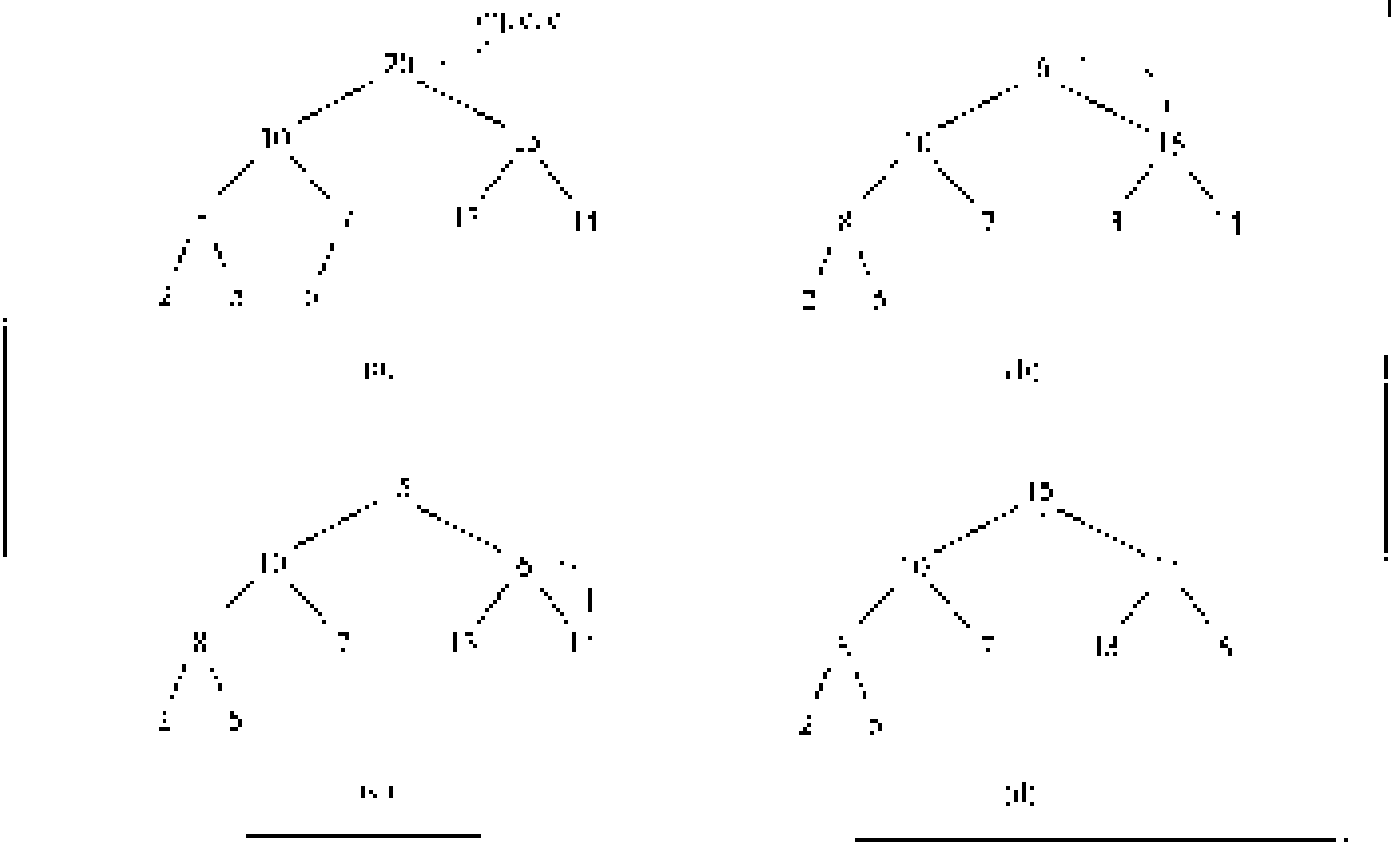
The algorithm for enqueuing is as follows:

```
heapEnqueue(...);
   put it at the end of heap;
   while it is not the root and it is smaller than its parent
      swap it with its parent;
```

For example, the number 15 is added to the heap in Figure 6.54a as the new leaf (Figure 6.54b), which destroys the heap property of the tree. To restore the property, 15 has to be moved up the tree until either it ends up as the root or finds a parent that is not less than it. In this example, the latter case occurs, and 15 must be moved only twice without reaching the root.

Dequeuing an element from the heap consists of removing the element from the heap (since by the heap property it is the element with the greatest priority), then the last leaf is put in its place, and the heap property is restored by moving the element just placed by moving it down the tree toward the root.

FIGURE 6.55 Percolating an element through a heap.



The algorithm for deleteMin is as follows.

```
boolean deleteMin( )
   remove the element in the root;
   put the element from the last leaf in the root;
   remove the last leaf;
   // Now the subtrees of the root are heaps;
   percolate down;
   while percolate is not finished and it has a left child
      swap with the larger child;
```

For example, 31 is dequeued from the heap in Figure 6.55a and 26 is put into its place (Figure 6.55b). To restore the heap property, 26 is swapped first with its larger child, number 44 (Figure 6.55c), and then again with the larger child, 31 (Figure 6.55d).

The last three lines of the deleteMin algorithm can be treated as a separate routine that restores the heap property only if it has been violated by the root of the tree. In this case, the element is moved down the tree until it finds a proper position. This algorithm is left to the reader. The deleteMin routine is presented in the prototypical implementation in Figure 6.56.

Program 6.56 Complete bottom-up algorithm for constructing a heap (recursive)

```
template<class T>
void makeheap(T data[], int first, int last) {
    int largest = 2*first + 1;
    while (largest <= last) {
        if (largest < last && // first has two children and
            data[largest] < data[largest+1]) // 2*first+1 was the smaller
          largest++;                         // of the two children
        if (data[first] < data[largest]) {   // if necessary,
          swap(data[first], data[largest]); // swap child and parent,
          first = largest;                  // and move down
          largest = 2*first+1;
        }
        else largest = last+1;  // or exit the loop: the heap property
    }                           // can't be violated by data[first]!
}
```

6.9.2 Organizing Arrays as Heaps

Heaps can be implemented as trees, but in fact we usually store a heap in an array. Recall from Section 6.3.2 that we can convert an array into a heap by organizing the data in the array so that the resulting complete tree satisfies the heap property. There are several ways to do this, but in light of the preceding section, the simplest way is to start with an empty heap and sequentially include the elements into a growing heap. This is a top-down method and it was proposed by John Williams; it creates the heap by enqueuing new elements in the heap.

Figure 6.57 contains a heap as an example of the top-down method. First, the number 3 is enqueued in the initially empty heap (6.57a). Next, 8 is enqueued by putting it at the end of the current heap (6.57b) and then swapping with its parent (6.57c). Enqueuing the third and fourth elements, 6 (6.57d) and 10 (6.57e), causes similar two steps. Enqueuing the fifth element, 10, amounts to putting it at the end of the heap (6.57f), then swapping it with its parent, 8 (6.57g), and then with its new parent, 6 (6.57h) so that eventually 10 percolates up to the root of the heap. All the preceding steps can be traced in Figure 6.57.

To check the complexity of the algorithm, observe that in the worst case, when a newly added element has to be moved up to the root of the heap, $\lg k$ exchanges are made in a heap of k nodes. Therefore, if n elements are enqueued, last in the worst case

$$\sum_{k=1}^{n} \lg k = \lg 1 + \cdots + \lg n = \lg(1 \cdot 2 \cdot \cdots \cdot n) = \lg(n!) = O(n \lg n)$$

exchanges are made during execution of the algorithm and the same number of comparisons. For the last equality, $\lg(n!) = \Theta(n \lg n)$, see Section A.2 in Appendix A. It turns out, however, that we can do better than that.

In another algorithm developed by Robert Floyd, a heap is built bottom up. In this approach, small heaps are formed and repetitively merged into larger heaps in the following way:

```
FloydHeapConstruction(data[])
    for i ← index of the last nonleaf, i ≥ 0, i ← i − 1
        ensure the heap property for the tree rooted at data[i] by calling
        trickleDown(data, i, n − 1)
```

Figure 6.8 contains an example of transforming the array data[0..5] = [2, 9, 7, 1, 6, 8] into a heap.

We start from the last nonleaf node, which is data[n/2 − 1], assuming the array size. If data[n/2 − 1] is less than one of its children, it is swapped with the larger child. In the resulting tree (Figure 6.8a), this is the case for data[3] = 1 and data[1] = 9. After exchanging the elements, a new tree is created, shown in Figure 6.8b. Next the element data[n/2 − 2] = data[2] = 7 is considered. Because it is smaller than its child data[5] = 8, it is swapped with that child and the tree is transformed as in Figure 6.8c. Now data[n/2 − 3] = data[1] = 9 is considered. Since it is smaller than one of its children, which is data[3] = 1, an interchange occurs, leading to the tree in Figure 6.8d. But now it can be viewed as the order established in the subtree whose root was 1 (Figure 6.8c). That subtree is somewhat disturbed since 9 is smaller than its new child 1. This simply means that it does not suffice to compare a node's value with its children's, but a similar comparison needs to be done with grandchildren's, great grandchildren's, and so on, until the node finds its proper position. Taking this into consideration, two more swaps are made, after which the tree in Figure 6.8e is created. Only now is the root data[0] = 2 compared with its children, as in Figure 6.8f.

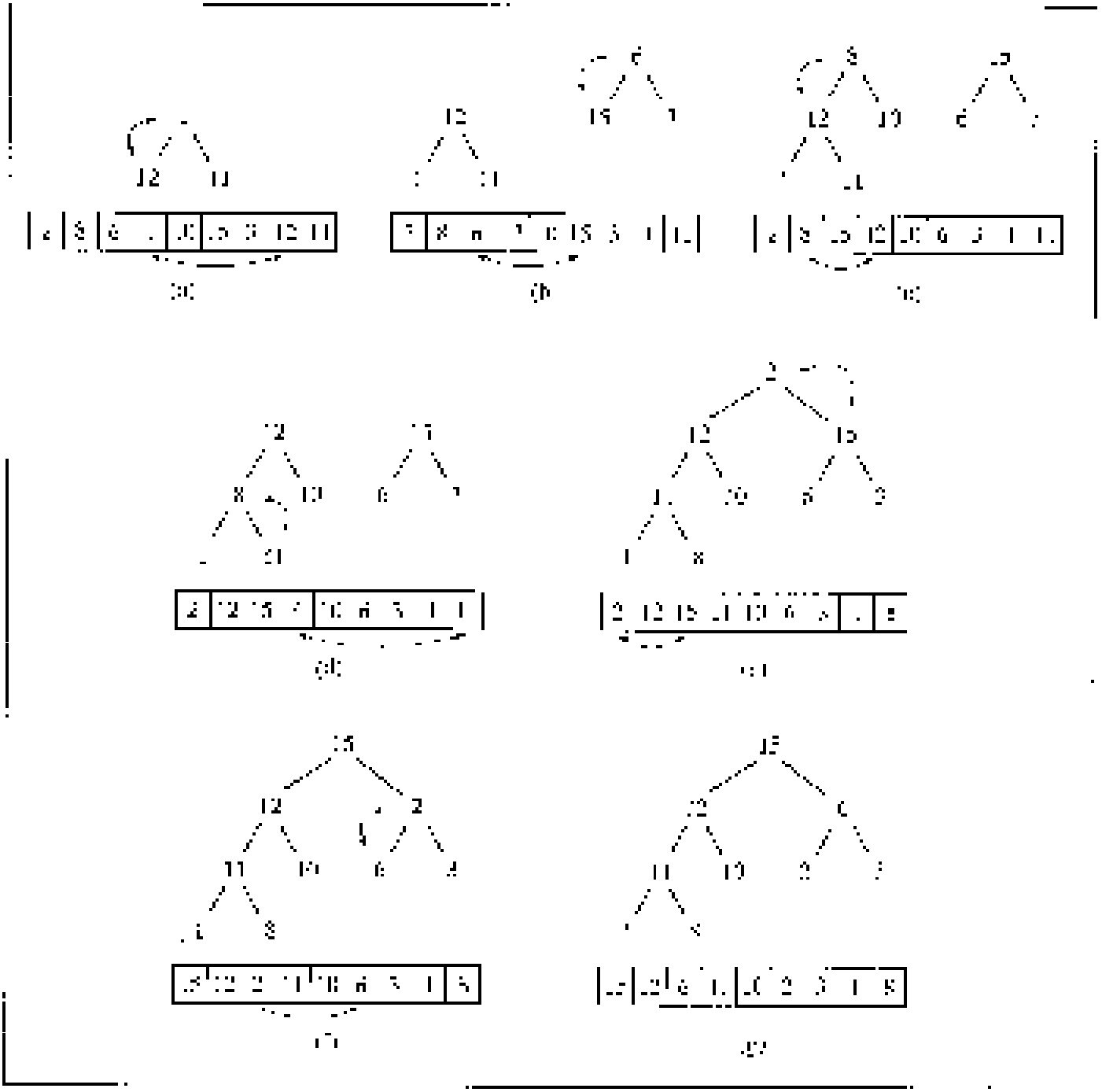
When a new element is analyzed, both its subtrees are already heaps, as is the case with the root 2 and both its subtrees with roots 9 and 7 (Figure 6.8f). Nevertheless, it may violate the heap property deeper down. This can be corrected by calling the procedure trickleDown, which repeatedly exchanges a given element with the larger of its children until the element is not smaller than either of its children or it becomes a leaf. For example, for the tree in Figure 6.8f, trickleDown(data, 0, 5) will make exactly two swaps, moving 2 down to the leaf level, as in Figure 6.8g. It is easy to see that the number of key comparisons made by the algorithm in the worst case is

$$C_{worst}(n) = 2(n - \log_2(n+1)) \in \Theta(n).$$

We assume here that $n = 2^k - 1$ for some k. To create the heap, we perform $\frac{n}{2}$ trickleDown operations, one for each parental node. In the worst case, however, $\frac{n}{2}$ moves down from the root to the leaf level, a total of $\frac{n}{2}\log_2 n$ moves. Of these, all but $\log_2(n+1) - 1$ nodes, the root and one node on each level of the tree, can be moved back down to reach the level of the leaves. Thus, aside from those $\log_2(n+1) - 1$ moves and a constant number of moves for the root, the number of key moves is proportional to $\frac{n}{2}\log_2 n$. Since each of the moves in the worst case requires two key comparisons, $C_{worst}(n) \in \Theta(n \log n)$. It can be proved that the number of key comparisons in the worst case is actually

$$C_{worst}(n) = 2n - 2\log_2(n+1) \in \Theta(n).$$

FIGURE 6.58 Transforming the array 2 8 6 1 10 15 3 12 11 into a heap with a bottom-up method.



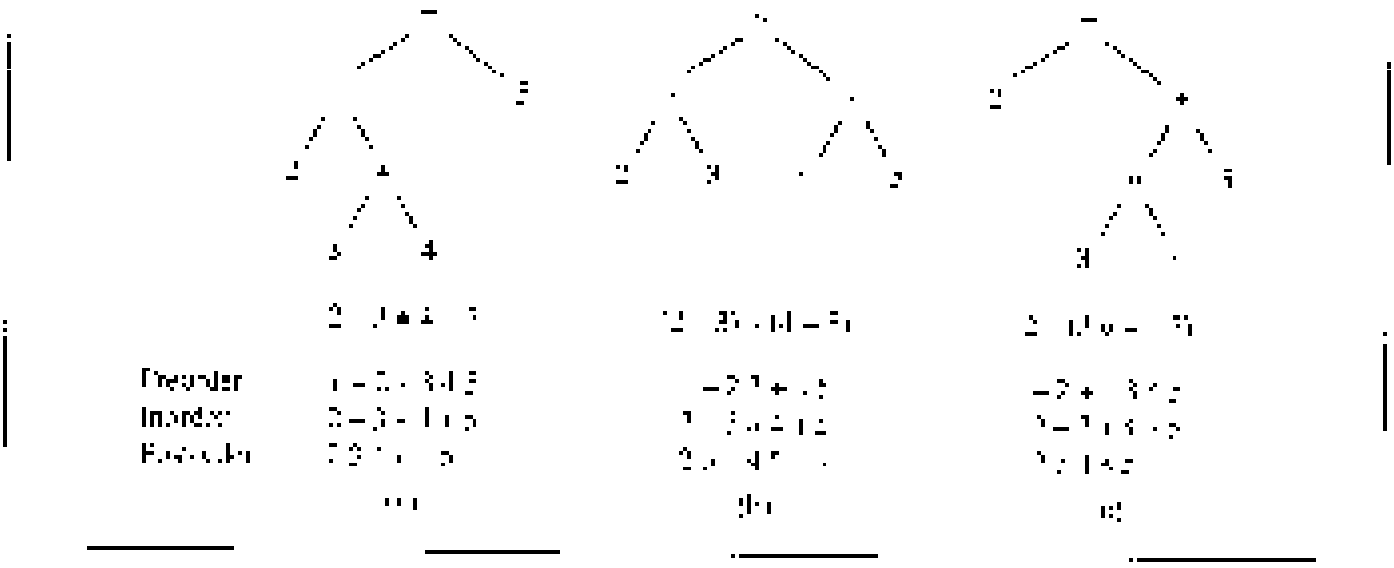
down the tree to find a correct spot for the lowest value. There is only one such. This can require up to $\lfloor \log n \rfloor$ moves. The total number of movements can be given by this sum

$$\sum_{i=1}^{\lfloor \log n \rfloor} i \cdot (\lfloor n/2^i \rfloor - \lfloor n/2^{i+1} \rfloor) \leq n \sum_{i=1}^{\lfloor \log n \rfloor} \frac{i}{2^i}$$

which is $O(n)$ since the series $\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges to 2 and $\sum_{i=1}^{k} \frac{i}{2^i}$ converges to 2 for any k. For an array which is already a near-heap, like a sorted array, the complexity is still the more bounded

[illegible] [illegible] [illegible] [

FIGURE 6.59 Examples of three expression trees and results of their traversals.



to be broken down manipulatively into separate operands and put into their proper order. That is where Polish notation is useful. It allows us to create an expression tree with no imposed rules for the execution of operations. For example, the first tree—$-+*2\ 3\ 4\ 5$—which is the same as $2 * 3 + 4 - 5$, is the one associated with the tree in Figure 6.59a. The second and third expressions correspond to the trees in Figures 6.59b and c, respectively. Note that in both Figures 6.59a and 6.59c we have to first multiply 2 by 3 to obtain 6. But 12 is subtracted from 5, according to the tree in Figure 6.59b, and added to 5, according to Figure 6.59c. There is no ambiguity involved in this tree representation. The final result can be computed only if intermediate results are calculated first.

Notice also that no parentheses are required to unambiguously express any arithmetic expression if the expression tree is linearized (that is, if the tree is transformed into an expression string) by one of these traversal methods. The three traversal methods differ in their conventions for nodes and operands in the traversals. Using these traversals, the outputs are generated as shown in Figures 6.59. Unambiguous node traversal in all three cases results in the same outputs, which are the original expressions that created all the trees. What is important here is that these traversals are valuable for providing unambiguous output. But the other four traversals are. They are different for different trees and are therefore useful for the purpose of creating unambiguous expressions and trees.

Because of the importance of these different representations, special terminology is used. Preorder traversal generates prefix notation, inorder traversal generates infix notation, and postorder traversal generates postfix notation. Note that we are not limited to these notations. In infix notation, an operation is surrounded by its two operands. In prefix notation, the operator precedes the operands and in postfix notation, the operator follows the two operands. Some hand-held calculators operate on expressions in postfix

notation. Also, some programming languages are using Polish notation. For example, Forth uses postfix notation, LISP and, to a large degree, LOGO use prefix notation.

6.10.1 Operations on Expression Trees

Binary trees can be created in two different ways: top-down or bottom-up. In the implementation of recursion, the first approach was used. This section applies the second approach by creating expression trees bottom-up while scanning infix expressions from left to right.

The most important part of this reconstruction process is retaining the same precedence of operations as in the expression being scanned, as exemplified in Figure 6.25. If parentheses are not allowed, the task is simple, as parentheses allow for many levels of nesting. Therefore, an algorithm should be powerful enough to process any number of nesting levels in an expression. A natural approach is a recursive implementation. Motivated by the recursive descent interpreter discussed in Chapter 5's case study and other examples, we define an expression tree constructor.

As Figure 6.25 indicates, a node contains either an operator or an operand. The latter is usually a single identifier or a number. To simplify our task, all of them can be represented as strings. Hence, a node is an object of the class defined as

```
class TreeNode {
public:
    TreeNode(char *s = 0, TreeNode *l = 0, TreeNode *r = 0)
        key = new char[strlen(s)+1];
        strcpy(key, s);
        left = l; right = r;
    }
    ~TreeNode();
private:
    char *key;
    TreeNode *left, *right;
};
```

Expressions that are converted to trees via the stack are infix expressions in the reverse Polish notation. Therefore, the same recursive algorithms can be used. Using those algorithms, a class TreeNode can be created in which the methods for processing infix and prefix expressions follow the algorithms for processing postfix expressions. For example, the function for processing infix expressions has the following form (assuming a binary operator):

```
TreeNode *
Expr::getInfix()
{
    if (token == NUMBER || token == IDENT) {
        TreeNode *p = new TreeNode(tokenStr);
        getToken();
        return p;
    }
    else if (token == '(') {
        getToken();
        TreeNode *p = expr();
        if (token == ')') {
            getToken();
            return p;
        }
        else error();
```

```
term();
    ExprTreeNode *p , *p2;
    p  = factor();
    while (token == '*' or  / )
        oper = token;
        p2 = factor();
        p  = new ExprTreeNode(oper,p,p2);
    return p;
```

Tree structures in expressions is very suitable for generating assembly code or intermediate code in compilers, as shown in this pseudocode of a function from ExprTree class:

```
void generateCode() {
     generateCode(root);
}
generateCode(ExprTreeNode *p) {
    if (p->key is a number) {
         return p->key;
    else if (p->key is an addition operator)
         result = nextTemporaryVar();
         output << "addl" << generateCode(p->left) << ","
                << generateCode(p->right) << ","
                <<result<<endl;
         return result;
    . . . . . .
}
```

With these member functions an expression

```
(x + 1) * (x/2 - y) + 5
```

intermediate code sequence can be shown as follows (produced by functions generateCode() printed out the following intermediate code):

```
addl    var_x       1           _tmp_3
addl    var_x       var_1       _tmp_4
mul     _tmp_3      _tmp_4      _tmp_5
div     _tmp_x      2           _tmp_
```

Expression trees are also very convenient for performing other symbolic operations, such as differentiation. Rules for differentiation given in the programming assignment in Chapter 5 are shown in the form of tree transformation in Figure 6.6, and in the following pseudocode.

```
differentiate(x) {
    if (p == 2)
        return 0;
    if (p->key is x)
        return new ExprTreeNode(1);
```

Figure 6.60 An expression tree.

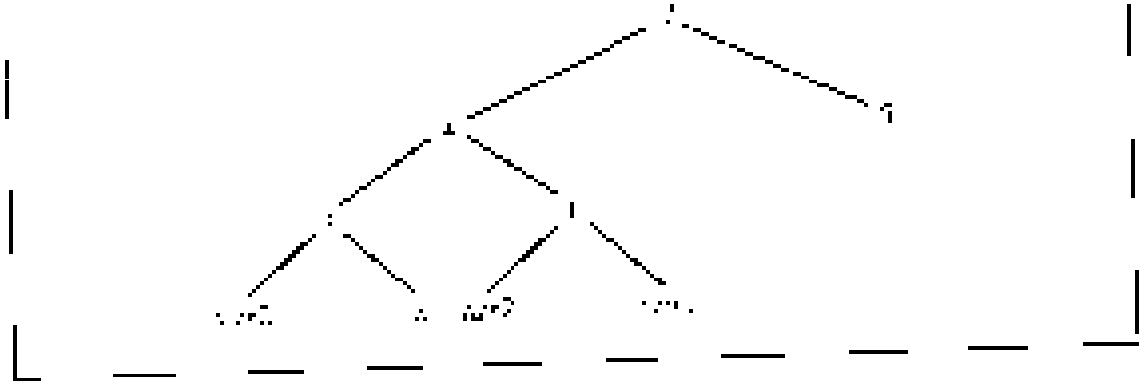
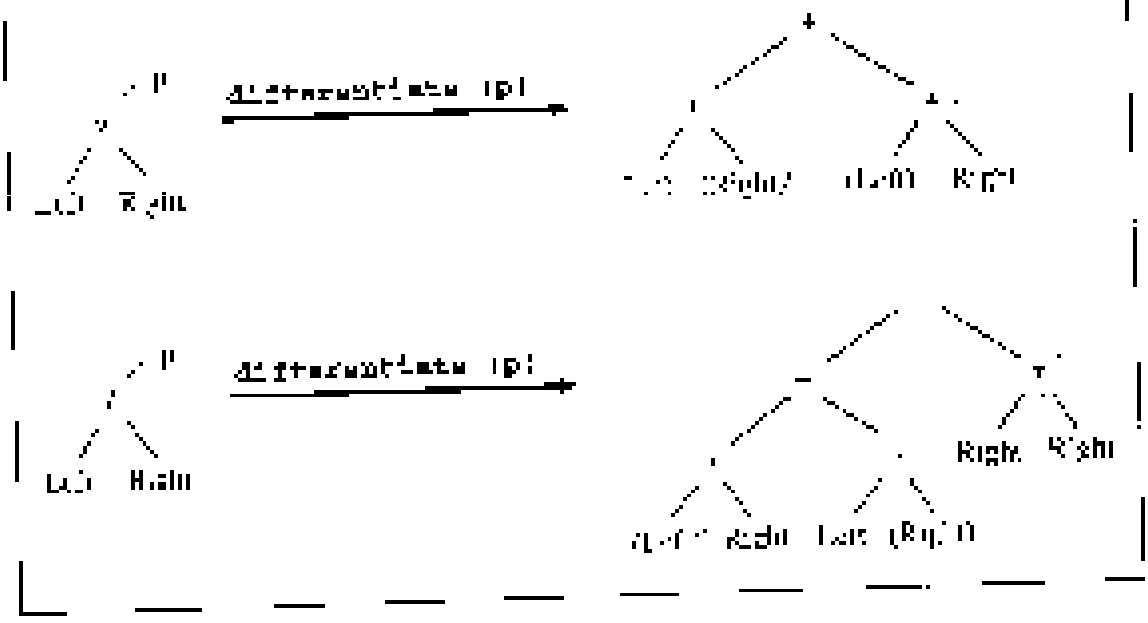


Figure 6.61 Tree transformations for differentiation of multiplication and division.



```
if (p->key
    return new ExpTreeNode('0');
if (p->key
    return new ExpTreeNode(p->key,Differentiate(p->left,x),
                                  Differentiate(p->right,x));

if (p->key
    ExpTreeNode *q = new ExpTreeNode('*');
    q->left = new ExpTreeNode('*',p->left,new ExpTreeNode(*p->right));
    q->right = Differentiate(q->left->right,x);
    q->right = new ExpTreeNode('*',new ExpTreeNode(*p->left),q->right);
```

```
    q.right->left = difference(p->right, q->left);
    return q;
  }
  ...
}
```

Here is a program that expresses both of these rules. Enjoy!

```
Enter a postfix expression (q to quit):
23 4 + 5 *
Enter a postfix expression (q to quit):
q
```

6.11 Case Study: Computing Word Frequencies

Consider the following problem: given a piece of text, such as a novel or a collection of newspaper articles, count the number of times each word occurs in the text. For example, if the text is "to be or not to be", then the word "to" occurs twice, and all other words occur once. Let's assume for the moment that the text is stored in a file, and that we want to write a program that reads the file and prints a table of words and their frequencies.

Regardless of how we decide to implement this program, we will have to make a decision about what we mean by a word. Should "Program" and "program" be considered the same word? Should "program." (with a period) be considered the same word as "program"? Should "data's" be considered the same word as "data"? Should "data's" be considered two words, "data" and "s", although it is not a word? Should "Data's" be considered the same word as "data's"? Should "data" be considered the same word as "data"? Should "data" be considered the same word as "data"? Should "data" be considered the same word as "data

FIGURE 6.62 Hierarchy used for computing word frequencies.

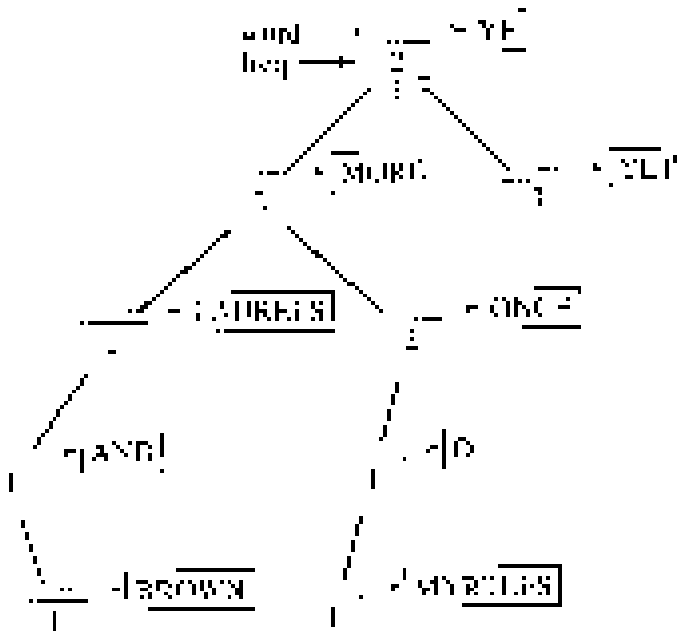


FIGURE 6.63 Implementation of word frequency computation.

```
//                      generic splaying tree class

#ifndef SPLAYTREE
#define SPLAYTREE

#include <assert.h>

template<class T> class SplayTree;

template<class T>
class SplayingNode {
public:
    SplayingNode() {
        left = right = parent = 0;
    }
    SplayingNode(const T& el, SplayingNode *l = 0, SplayingNode *r = 0,
```

Figure 6.63 (continued)

```
                SplayingNode *p = 0;
            node = x; left = 1; right = 2; parent = 3;
        }
        T data;
        SplayingNode *left, *right, *parent;
    };

    template<class T>
    class SplayTree {
    public:
        SplayTree() {
            root = 0;
        }
        void inorder() {
            inorder(root);
        }
        T* search(const T&);
        void insert(const T&);
    protected:
        SplayingNode<T> *root;
        void rotateR(SplayingNode<T>*);
        void rotateL(SplayingNode<T>*);
        void splay(SplayingNode<T>* gr, SplayingNode<T>* par,
                   SplayingNode<T>* p, SplayingNode<T>* node);
        void visit(SplayingNode<T>*);
        void inorder(SplayingNode<T>*);
        void virtual visit(SplayingNode<T>*) {
        }
    };

    template<class T>
    void SplayTree<T>::rotateR(SplayingNode<T>* gr,
    SplayingNode<T>* par, SplayingNode<T>* ch, SplayingNode<T>* node) {
        if (par != 0) { // ch's new parent:
            if (par->right == ch) par->right =
                gr->right;
            else par->left    ch;
        }
        else root    ch;
        if (node != 0)
```

Figure 6.63 (continued)

```
        Class->parent = par;
    par->rightChild = ch;
    ch->parent = par;
}

template<class T>
void SplayTree<T>::rotateRight(BinaryNode<T>* p) {
    p->rightChild->leftChild = p->right;
    p->right = p->parent;
    rotateRight(p->parent->parent,p->right,p->parent->parent);
}

template<class T>
void SplayTree<T>::rotateLeft(BinaryNode<T>* p) {
    p->parent->rightChild = p->left;
    p->left = p->parent;
    rotateRight(p->parent->parent,p->left,p,p->left->right);
}

template<class T>
void SplayTree<T>::splay(BinaryNode<T>* p) {
    while (p != root) {
        if (p->parent->parent == 0)     // p's parent is the root;
            if (p->parent->left == p)
                rotateR(p);
            else rotateL(p);
        else if (p->parent->left == p) // p and left child;
            if (p->parent->parent->left == p->parent) {
                rotateR(p->parent);
                p = p->parent;
            }
            else {
                rotateR(p); // rotate p and parent;
                rotateL(p); // rotate p and new parent;
            }
        else                            // if p is a right child;
            if (p->parent->parent->right == p->parent) {
                rotateL(p->parent);
                p = p->parent;
            }
```

Continues

FIGURE 6.63 (continued)

```
            else {
                rotateL(p); // rotate p and its parent
                rotateR(p); // rotate p and its new parent
            }
        if (root == 0)              // update the root:
            root = p;
    }
}

template<class T>
T* SplayTree<T>::search(const T& el) {
    SplayingNode<T> *p = root;
    while (p != 0)
        if (p->info == el) {        // if el is in the tree,
            semisplay(p);           // move it toward;
            return &p->info;
        }
        else if (el < p->info)
            p = p->left;
        else p = p->right;
    return 0;
}

template<class T>
void SplayTree<T>::insert(const T& el) {
    SplayingNode<T> *p = root, *prev = 0, *newNode;
    while (p != 0) {  // find a place for inserting a new node;
        prev = p;
        if (el < p->info)
            p = p->left;
        else p = p->right;
    }
    if ((newNode = new SplayingNode<T>(el,0,0,prev)) == 0) {
        cerr << "No room for new nodes\n";
        exit(1);
    }
    if (root == 0)    // the tree is empty;
        root = newNode;
    else if (el < prev->info)
        prev->left  = newNode;
```

FIGURE 6.63 (continued)

```
    else prev->right = newNode;
}

template<class T>
void BSplayTree<T>::InorderTraversal(BSplayingNode<T> *p) {
    if (p != 0) {
        InorderTraversal(p->left);
        visit(p);
        InorderTraversal(p->right);
    }
}

#endif

//************************  bsplay.cpp  ************************

#include <fstream.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h> // exit()
#include "bsplay.h"

class Word {
public:
    Word() {
        freq = 1;
    }
    int operator== (const Word& ir) const {
        return strcmp(word,ir.word) == 0;
    }
    int operator< (const Word& ir) const {
        return strcmp(word,ir.word) < 0;
    }
private:
    char *word;
    int freq;
    friend class WordSplay;
    friend ostream& operator<< (ostream&,const Word&);
};
```

Figure 6.63 (continued)

```
class wordDelay : public DelayTree<Word> {
public:
    WordDelay() {
        differentWords = wordCnt = 0;
    }
    void run(ifstream&,char*);
private:
    int differentWords, // counter of different words in a text file;
        wordCnt;        // counter of all words in the same file;
    void visit(DisplayingNode<Word>*);
};

void WordDelay::visit(DisplayingBinNode<Word> *p) {
    differentWords++;
    wordCnt += p->info.freq;
}

void WordDelay::run(ifstream &in, char *fileName) {
    char ch = ' ', i;
    char s[20];
    Word *w;
    while (!in.eof()) {
        while (!
            if (!in.eof() && !isalpha(ch)) // skip nonletters
                in.get(ch);
            else break;
        if (in.eof())       // appears at the end of in;
            break;
        for (i = 0; !in.eof() && isalpha(ch); i++) {
            s[i] = tolower(ch);
            in.get(ch);
        }
        s[i] = '\0';
        if (!(w = new Word(s,1))) {
            cerr << "No room for new words.\n";
            exit(1);
        }
        strcpy(w->word,s);
        Word *p = search(w);
        if (p == 0)
```

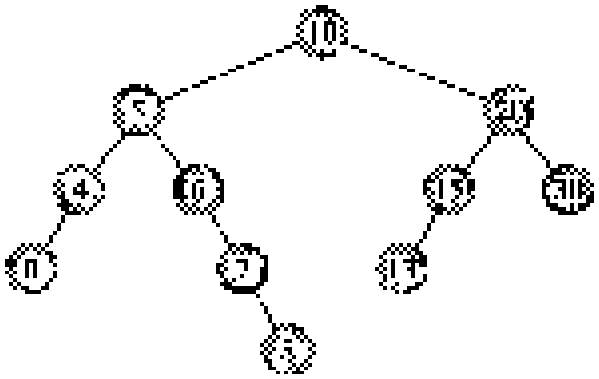
Figure 6.63 (continued)

```
            insert(ch);
        else p->right =
    }
    inorder();
    cout << "\nWords in file " << filename
         << " contain " << wordCount << " words among which "
         << diffCountWords << " are different";
}

void main(int argc, char* argv[]) {
    char fileName[80];
    WordDisplay wordDisplay;
    if (argc != 2) {
        cout << "Enter a file name: ";
        cin >> fileName;
    }
    else strcpy(fileName,argv[1]);
    ifstream fin(fileName);
    if (!fin) {
        cout << "Cannot open " << fileName << endl;
        return;
    }
    wordDisplay.run(fin,fileName);
    fin.close();
}
```

■ 6.12 EXERCISES

1. The function search() given in Section 6.3 is well suited for searching binary search trees. Try to adapt all four traversal algorithms so that they become search procedures for any binary tree.
2. Write functions to count the number of nodes in a binary tree, the number of leaves, the number of right children, and the height of the tree.
3. Write a function that checks whether or not a binary tree is perfectly balanced.

Figure 6.64 An example of a binary search tree.



4. Design an algorithm to test whether a binary tree is a binary search tree.

5. Write a function to delete all leaves from a binary tree.

6. Apply preorder(), inorder(), and postorder() to the tree in Figure 6.64. If visit(p) is defined as:

   a. `if (p->left != 0 && p->key < p->left->key) p->left->key = 2;`

   b. `if (p->left == 0) p->right = 0;`

   c. `if (p->left == 0) p->left = new treeNode(p->key - 2);`

   d. `{ tmp = p->right; p->right = p->left; p->left = tmp; }`

7. Show a tree for which the preorder and inorder traversals generate the same sequence.

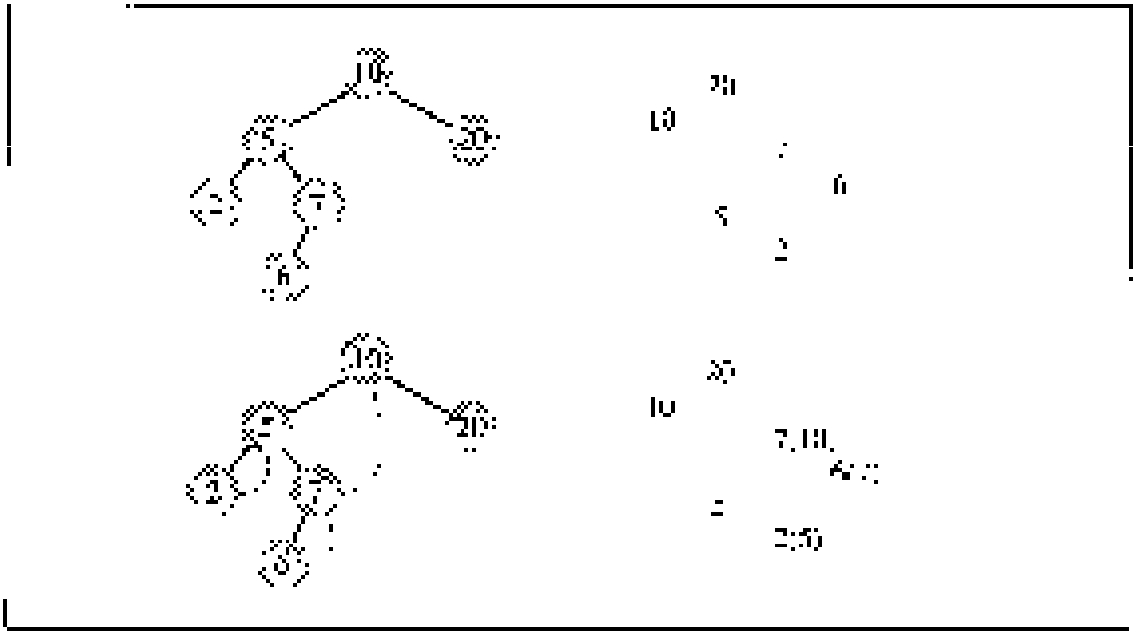
8. Figure 6.79 indicates that the inorder traversal for different trees can result in the same sequence. Is it possible for the preorder or postorder traversal? If it is, show a counterexample.

9. Draw all possible binary search trees for the three elements A, B, and C.

10. What are the minimum and maximum numbers of leaves in a balanced tree of height h?

11. Write a function to generate a mirror image of a binary tree.

FIGURE 6.65 Printing a binary search tree (a) and a threaded tree (b) sideways.



12. Consider an operation R that for a given traversal method processes nodes in the opposite order than I and an operation C that processes nodes of the mirror image of a given tree using traversal method I. For the tree traversal methods preorder, inorder, and postorder determine which of the following nine equalities are true:

$$\begin{aligned}
R(\text{preorder}) &= C(\text{preorder})\\
R(\text{preorder}) &= C(\text{inorder})\\
R(\text{preorder}) &= C(\text{postorder})\\
R(\text{inorder}) &= C(\text{preorder})\\
R(\text{inorder}) &= C(\text{inorder})\\
R(\text{inorder}) &= C(\text{postorder})\\
R(\text{postorder}) &= C(\text{preorder})\\
R(\text{postorder}) &= C(\text{inorder})\\
R(\text{postorder}) &= C(\text{postorder})
\end{aligned}$$

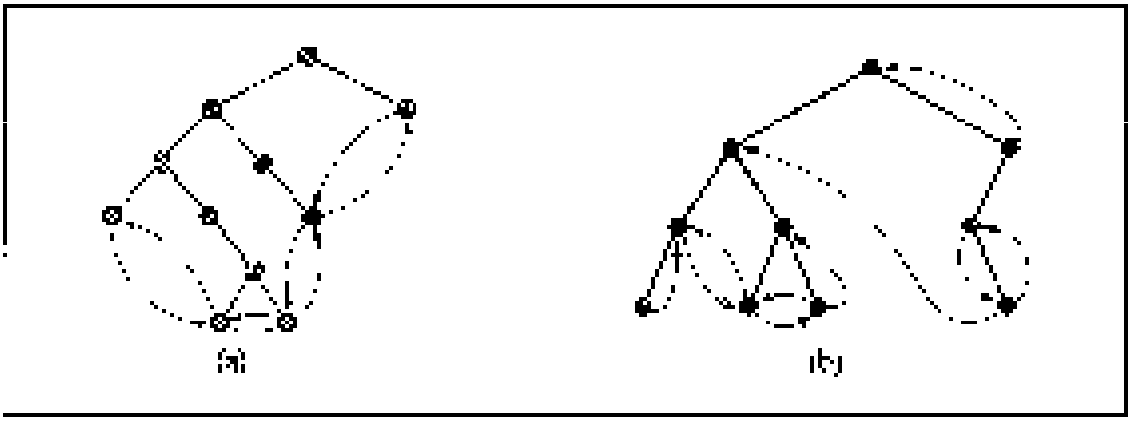
13. Using inorder, preorder, and postorder tree traversal, visit only leaves of a tree. What can you observe? How can you explain this phenomenon?

14. (a) Write a function which prints sideways such binary tree with process information, as in Figure 6.65(a). Accept this function to print a threaded tree sideways, if appropriate, print the key in the successor nodes, as in Figure 6.65(b).

15. Outline functions for inserting and deleting a node in a threaded tree in which threads are put only in the form as in the way illustrated by Figure 6.66.

16. The tree in Figure 6.66 includes threads linking predecessors and successors according to the postorder traversal. Are these threads adequate to perform threaded preorder, inorder, and postorder traversals?

FIGURE 6.66 Examples of threaded trees



17. Apply the function balance(1) to the English alphabet to create a balanced tree.

18. A sentence Dpq that uses a Sheffer's alternation is false only if both p and q are true. In 1923, Łukasiewicz simplified Nicod's axiom from which all theses of propositional logic can be derived. Transform the Nicod-Łukasiewicz axiom into an infix parenthesized sentence and build a binary tree for it. The axiom is DDpDqrDsDttDDsqDDtpDtp.

19. Write an algorithm for printing a parenthesized infix expression from an expression tree. Do not include redundant pairs of parentheses.

20. Tibbard's (1978) algorithm to delete a key from a binary search tree requires that if the node containing the key has a right child, then the key is replaced by the smallest key in the right subtree; otherwise, the node with the key is removed. For what reason is Knuth's algorithm (Algorithm 6.4) an improvement?

21. Which tree of four nodes is the worst case for processing a lookup in the DSW algorithm? Execute this procedure in this tree by hand.

22. Define a binary search tree in terms of the inorder traversal.

23. A Fibonacci tree can be considered the worst case AVL tree in that it has the smallest number of nodes among AVL trees of height h. Draw Fibonacci trees for h = 1, 2, 3, 4 and justify the name of the tree.

24. Class-ified height-balanced trees are AVL trees in which only two types of nodes are allowed—L and R or R and L (Zweben and McDonald 1978). What is the ratio for arranging this type of tree?

25. In lazy deletion, nodes to be deleted are retained in the tree but only marked as deleted. What are the advantages and disadvantages of this approach?

26. What is the number of comparisons and swaps in the best case for creating a heap using (a) Williams's method and (b) Floyd's method?

27. A crossover between Floyd's and Williams's methods for constructing a heap is a method in which an empty position created by the removal is moved down to the bottom of the tree and then the element is moved up, as in Williams's method, but not past the position that was just moved down. A pseudocode of this function is as follows:

```
i = (size-1)/2; // position of the last parent in the array of n elements

while (i >= 0)
    // Floyd's phase:
    top = array[i];
    consider element array[i] empty and move it down to the bottom
        swapping it every time with larger child;
    put top in the leaf at which this process ended;
    // Williams's phase:
    while top is not the root and it is larger than its parent
        swap top with its parent;
    i--; // go to the preceding parent
```

It has been shown that this algorithm requires $1.625n$ comparisons on the average case (McDiarmid and Reed 1989). Show changes in the array [2 8 6 1 10 15 3 12 ...] during execution of the algorithm. What is the worst case?

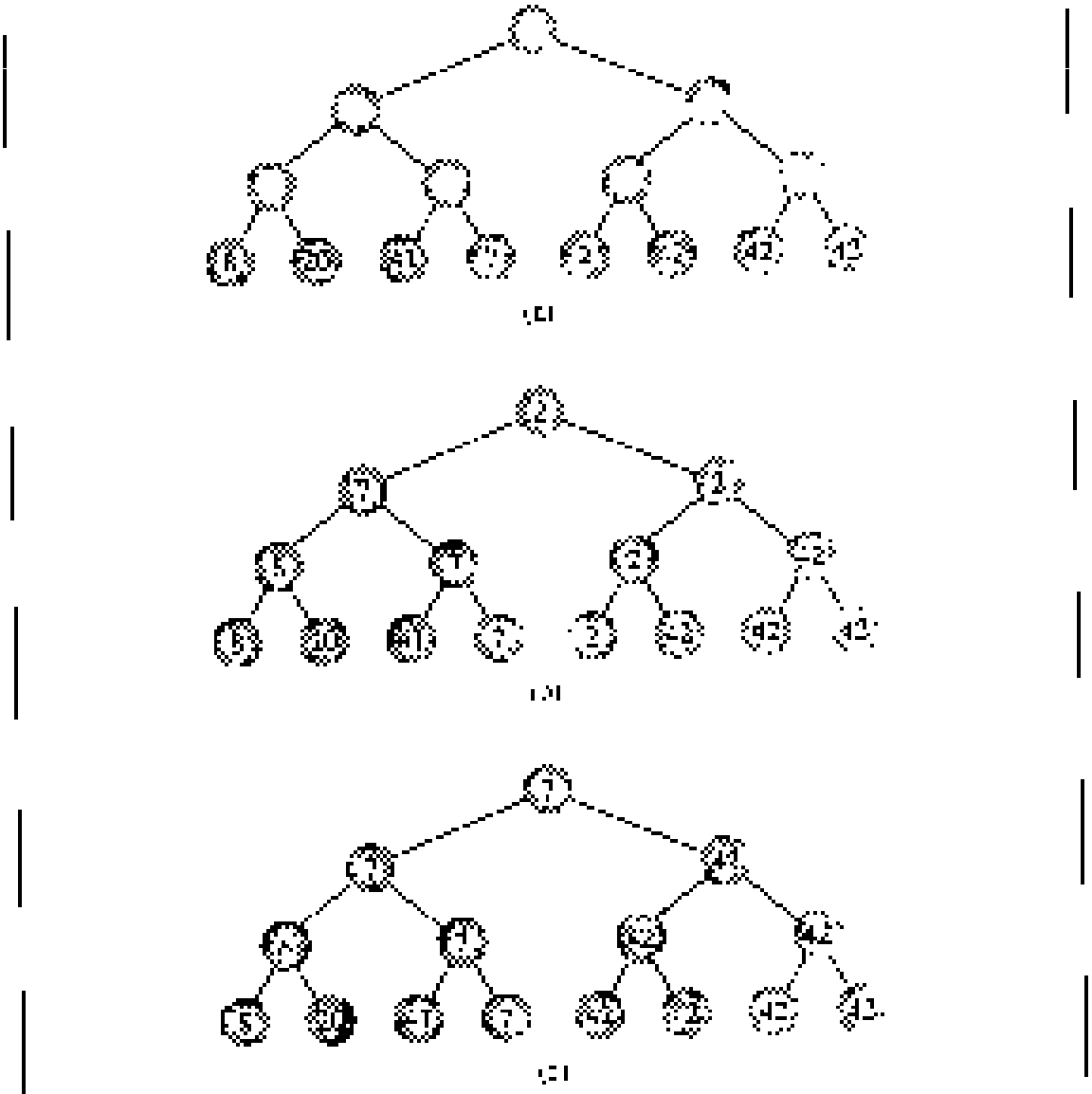
6.13 PROGRAMMING ASSIGNMENTS

1. Write a program that accepts an arithmetic expression written in prefix (Polish) notation, builds an expression tree, and then traverses the tree to evaluate the expression. The evaluation should start after a complete expression has been entered.

2. A binary tree can be used to sort n elements of an array data. First, create a complete binary tree, a tree with all leaves at one level, whose height is $h = \lfloor \lg n \rfloor + 1$, and store all elements of the array in the first n leaves. In each empty leaf, store an element δ greater than any element in the array. Figure 6.63*a* shows an example for $size = 9$, $n = \lfloor \lg 9 \rfloor + 1 = 4$, and $\delta = 42$. Then, starting from the bottom of the tree, assign to each node the minimum of its two children values. In Figure 6.63*b*, so that the smallest element a_{min} in the tree is assigned to the root. Next, until the element δ is assigned to the root, execute a loop that in each iteration stores δ in the leaf with the value of a_{min} and then, also starting from the bottom, assigns to each node the minimum of its two children. Figure 6.63*c* displays this tree after one iteration of the loop.

3. Implement a menu-driven program for managing a software store. All information about the available software is stored in a file on the disk. This information includes the name, version, quantity, and price of each package. When it is invoked, the program automatically creates a binary search tree with one node corresponding to one software package and includes as the key the name of the package and its version.

FIGURE 6.67 Binary tree used for searching



Another field in this node should indicate the position of the record in the file and leaves. The only access to the information stored in leaf leaves should be through this node.

The program should allow the binary tree to be updated as new software packages arrive at the store and when single packages are sold. The tree is updated in the following way. All packages are entry added in the file and leaves. If a new package arrives, then it is put at the end of the file. If the package already has an entry in the tree (and the file), then only the quantity field is updated. If a package is sold out, the corresponding node is deleted from the tree, and the quantity field in the file is changed to 0. For example, if the file has three entries:

Adobe Photoshop	4.0	21	250
Norton Utilities NU5	3.0	10	45
Norton Utilities DOS/WIN	8.0	6	40
Visual C++	1.0	19	40
Visual C++	1.1	27	40

Then after will upgrade copies of Norton Utilities DOS/WIN 90, the file is

Adobe Photoshop	4.0	21	250
Norton Utilities NU5	3.0	10	45
Norton Utilities DOS/WIN	9.0	9	40
Visual C++	1.0	19	40
Visual C++	1.1	27	40

If an exception is chosen from the menu, the program discards the file by removing entries from the end of the file to the position marked with the pointer. For example, the previous file becomes

Adobe Photoshop	4.0	21	250
Norton Utilities NU5	3.0	10	45
Visual C++	1.0	27	40
Visual C++	1.1	19	40

4. Implement alphabetic hashing for constructing expression trees and for differentiating the expressions they represent. Extend the program to simplify expression trees. For example, zero nodes can be eliminated from the subtrees representing $x + 0$, $0 + x$, or $x * 1$.

5. Write a cross-reference program which constructs a binary search tree with all words included from a text file and records the line numbers on which these words were used. These line numbers should be stored in linked lists associated with the nodes of the tree. After the input file has been processed, print in alphabetical order all words of the text file along with the corresponding lists of line numbers of the lines in which the words occur.

6. Perform an experiment to verify informally the insertion and deletion of random elements in a randomly created binary search tree. Apply asymmetric and symmetric deletions discussed in this chapter, for both these variants of the deletion algorithm. Always work with trees consisting of the same number of nodes. Start with an empty tree and perform a sequence of, say, 1000 insertions. Then perform a sequence of 1000 deletions, intermixed randomly with insertions. Use two different random number generators to create randomness. This leads to eight combinations. Execute all of these combinations for trees of the sizes 5000, 10000, 15000, and 20000. Plot the results and compare them with the expected IPL given in this chapter.

7. Each night a Latin scholar conjures up a new English vocabulary of words which have come into use in particular topic. Write a program that reports a word of each vocabulary stored in file and its usage, as English-Latin word translations. Make the following assumptions:

   a. Unit names are preceded by a percentage symbol.

   b. There is only one unit per line.

c. A Latin word is separated by a colon from its English equivalent(s). If there is more than one equivalent, they are separated by a comma.

To output English words in alphabetical order, create a binary search tree for each unit containing English words and linked lists of Latin equivalents. Make sure that there is only one node for each English word in the tree. For example, there is only one node for *and*, although *and* is used twice in unit 5 with words *et* and *atque*. After the task has been completed for a given unit (that is, the contents of the next unit have been stored in an output file), delete the tree along with all linked lists from computer memory before creating a tree for the next unit.

Here is an example of a file containing Latin-English vocabularies:

```
%Unit 5
ante : before, in front of, previously
antiquus : ancient
ardeo : burn, be on fire, desire
arma : arms, weapons
aurum : gold
aureus : golden, of gold

%Unit 6
animal : animal
Athenae : Athens
atque : and
ac : and
aurora : dawn

%Unit 7
amo : love
amor : love
annus : year
Asia : Asia
```

From these units, the program should generate the following output:

```
%Unit 5
ancient : antiquus
arms : arma
be on fire : ardeo
before : ante
burn : ardeo
desire : ardeo
gold : aurum
golden : aureus
in front of : ante
of gold : aureus
```

```
previously : since
weapons : arms

%Unit 6
Athens : Athenian
sad : sorry, tragic
animal : animals
dead : deceased

%Unit 7
Asia : Asia
love : amor, amo
year : annus
```

Bibliography

Insertions and Deletions

Culberson, Joseph. "The Effect of Updates in Binary Search Trees." *Proceedings of the 17th Annual Symposium on Theory of Computing* (1985): 205–213.

Eppinger, Jeffrey L. "An Empirical Study of Insertion and Deletion in Binary Search Trees." *Communications of the ACM* 26 (1983): 663–669.

Hibbard, Thomas N. "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting." *Journal of the ACM* 9 (1962): 13–28.

Jonassen, Arne Tarald, and Knuth, Donald E. "A Trivial Algorithm Whose Analysis Isn't." *Journal of Computer and System Sciences* 16 (1978): 301–322.

Knuth, Donald E. "Deletions That Preserve Randomness." *IEEE Transactions of Software Engineering* SE-3 (1977): 351–359.

Tree Traversals

Hardgrave, W. T. "Binary Trees with Recursive Balancing." *Proceedings of the 1980 National Computer Conference* (1980): 517–526.

Knuth, Donald E. *The Art of Computer Programming*. Vol. 1, *Fundamental Algorithms*. Reading, Mass.: Addison-Wesley, 1968.

Morris, Joseph M. "Traversing Binary Trees Simply and Cheaply." *Information Processing Letters* 9 (1979): 197–200.

Balancing Trees

Baer, J. L., and Schwab, B. "A Comparison of Tree-Balancing Algorithms." *Communications of the ACM* 20 (1977): 322–330.

Chung, Kai-min [et al.]. "A Sharp Lower Bound for Comparison-Based Exclusive Scan." Information Processing Letters, vol. 110, no. 24 (2010): 1087–1090.

Dey, Arindam, Sudebkumar Prasant Pal, and Sandeep Sen. "Faster Random Walks in Groups with Some Applications." Theoretical Computer Science, vol. 411, no. 10 (2010): 1229–1234.

Martin, W. A., and J. I. Munro. "Heaps, Double Heaps, and Double-Ended Priority Queues." Communications of the ACM, vol. 18, no. 8 (1975): 423–424.

Sack, Jörg-Rüdiger, and Thomas Strothotte. "A Characterization of Heaps and Its Applications." Information and Computation, vol. 86, no. 1 (1990): 69–86.

AVL Trees

Adelson-Velskii, G. M., and E. M. Landis. "An Algorithm for the Organization of Information." Soviet Mathematics Doklady, vol. 3 (1962): 1259–1263.

Brass, Peter. "Fast Set Operations Using AVL Trees." Computational Geometry, vol. 30, no. 1 (2005): 1–21.

Karlton, P. L., J. H. Fuller, R. E. Scroggs, and E. B. Kaehler. "Performance of Height-Balanced Trees." Communications of the ACM, vol. 19, no. 1 (1976): 23–28.

Knuth, Donald E. The Art of Computer Programming, Vol. 3: Sorting and Searching. Reading, MA: Addison-Wesley, 1998.

Zweben, Stuart H., and Monte A. McDonald. "An Optimal Method for Deletion in One-Sided Height-Balanced Trees." Communications of the ACM, vol. 21, no. 6 (1978): 441–445.

Self-Adjusting Trees

Allen, Brian, and Munro, Ian. "Self-Organizing Binary Search Trees." Journal of the ACM, vol. 25, no. 4 (1978): 526–535.

Bitner, James R. "Heuristics that Dynamically Alphabetize and Optimize Binary Tree Structures." Journal of the ACM, vol. 24, no. 1 (1977): 82–110.

Sleator, Daniel D., and Robert E. Tarjan. "Self-Adjusting Binary Search Trees." Journal of the ACM, vol. 32, no. 3 (1985): 652–686.

Treaps

Aragon, Cecilia R., and Raimund Seidel. "Randomized Search Trees." Algorithmica, vol. 16, no. 4–5 (1996): 464–497.

Blelloch, Guy E., and Margaret Reid-Miller. "Fast Set Operations Using Treaps." Theory of Computing Systems, vol. 36, no. 5 (2003): 521–568.

Seidel, Raimund, and Cecilia R. Aragon. "Randomized Search Trees." Algorithmica, vol. 16, no. 4–5 (1996): 464–497.

Vuillemin, J. "A Unifying Look at Data Structures." Communications of the ACM, vol. 23, no. 4 (1980): 229–239.

Wenger, Rephael. "Randomized Quickhull." Algorithmica, vol. 17, no. 3 (1997): 322–329.

Maynard-Reid, Pedro, and Dan Chiu. "Software Plagiarism Detection for Java." Technical Report, Computer Science Department, Stanford University, 1999.

Wise, Michael J. "YAP3: Improved Detection of Similarities in Computer Program and Other Texts." Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education, 1996.

Multiway Trees

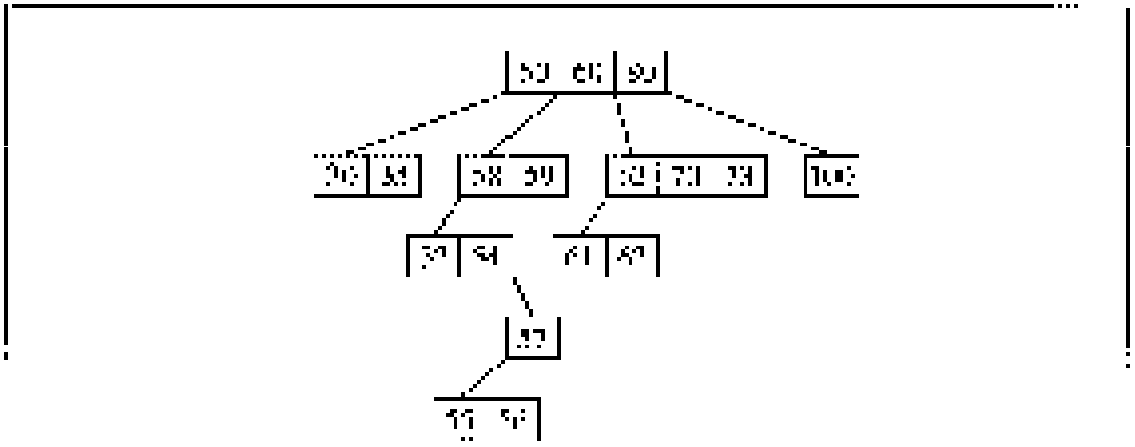
At the beginning of the preceding chapter a general definition of a tree was given, but the focus of that chapter was binary trees, in particular, binary search trees. A tree was defined as either an empty structure or a structure whose children are disjoint trees $t_0, \ldots, t_{n-1}$, according to the definition, each node of this kind of tree can have more than two children. Such a tree is called a multiway tree of order n, or an n-way tree.

In order to be useful as part of a multiway tree, a total ordering is imposed on the keys residing in each node. A multiway search tree of order n is an n-way search tree that is a multiway tree in which

1. Each node has n children and $n - 1$ keys.
2. The keys in each node are in ascending order.
3. The keys in the first i children are smaller than the ith key.
4. The keys in the last $n - i$ children are larger than the ith key.

The multiway search trees play the same role among multiway trees that binary search trees play among binary trees, and they are used for the same purpose: information retrieval and update. The problems they cause are similar. The tree in Figure 7.1 is a bad way to store information because accessing the keys can require a different number of tests for different keys. The number 30 can be found in the second node checked, and 55 can be found in the fifth node checked. The tree therefore varies from a known maximum; it is unbalanced. This problem is of practical importance if we want to use trees to process data on secondary storage, such as disks or tapes, where each access is costly. Constructing such trees requires a more complex approach.

FIGURE 7.1 A 4-way tree



7.1 The Family of B-Trees

The best search algorithms we've examined so far are all based on the assumption that the information we need to search is stored in memory. Such algorithms give up this performance when they are applied to data stored on disk. Every time a disk block is read, the information it contains is referred to as being *transferred* from the disk into memory. Because disk accesses are relatively slow, we want to be sure that the algorithms we use minimize the number of times we must read from or write to the disk. To be sure that a given algorithm is efficient, we must be able to count the number of times it accesses the disk, and then try to modify the algorithm so that it performs as few accesses as possible. Each time information is requested from or written to the disk, we must position the read/write heads over the part of the disk where the information is stored, and then wait for the disk to rotate so that the appropriate place passes underneath the head. Only then can the information be transferred to memory. The major cost factors are therefore the seek time and the latency:

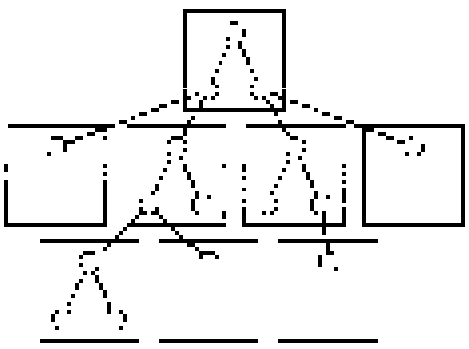
$$\text{access time} = \text{seek time} + \text{latency} + \text{transfer time}$$

The process is extremely slow compared to transferring information within memory. The first component, seek time, is particularly slow because it depends on the mechanical movement of the disk head. The head must be accurately positioned over the track containing the desired block, and on high-capacity disks this can require as much as a full revolution. For example, the time to transfer 8K bytes of data from a disk requiring 10 ms (milliseconds) to locate a track, rotating 3600 revolutions per minute, and with a data transfer rate of 1000K per second is

$$\text{access time} = 4 \text{ ms} + 10 \text{ ms} + 5 \text{ ms} = 19 \text{ ms}$$

This example indicates that transferring information to and from the disk is on the order of milliseconds. On the other hand, the CPU processes data at the rate of microseconds, 1000 times faster or more. On the order of nanoseconds, 1000 times faster yet. We can see that processing information on secondary storage can significantly decrease the speed of a program.

FIGURE 7.2 Nodes of a binary tree can be located in different blocks on a disk.



This approach is better by several orders of magnitude than accessing secondary storage for each node, but there is still room for improvement. Binary search trees are optimal trees with respect to the number of comparisons required to locate a key, but they are not optimal when it comes to minimizing the number of accesses to secondary storage. Because the time required to read information from secondary storage is significantly greater than the time required to process that information once it is in main memory, we sometimes trade additional processing for fewer accesses to secondary storage. In the case of binary search trees, we can improve the performance of disk-based algorithms by increasing the number of keys stored in each node, thereby increasing the fanout factor of the tree.

The other factor we must consider is the amount of data that can be transferred to memory from the disk during each access. In the case of binary trees, we transfer only a few hundred bytes with each read operation. The standard algorithm for reading a disk block is optimized for sequential processing, so reading larger blocks does not significantly increase the transfer time. Most of the time required to read a block is taken up by seek time and rotational latency. Seek time is the time required to position the read/write heads on the track. Rotational latency is the time required for the platter to rotate until the first byte of the block is under the read/write head. Once the head is properly positioned, data can be read very quickly. Depending on the disk device, seek time and rotational latency are roughly the same for each track. Let's assume that we can read 50 blocks per second from a typical disk. Then

$$\text{access time} = 40\text{ms} + 10\text{ms} + 10\text{ms} = 60\text{ms}$$

If we read 5 blocks on the same track, then

$$\text{access time} = 40\text{ms} + 10\text{ms} + 5 \times 10\text{ms} = 110\text{ms}$$

which is not quite double the time required to read a single block. The conclusion is that we can read several kilobytes of data with only a minor penalty over reading a single byte. Therefore, to minimize the number of accesses required to locate a piece of data, we should store as much information as possible in a single node. Instead of a binary tree, we could use a tree with hundreds of keys in each node and a fanout factor of hundreds. This approach greatly reduces the number of accesses required to find a key, because the tree is much shorter than a binary tree containing the same number of keys. The maximum number of accesses required to locate a key is equal to the height of the tree.

7.1.1 B-Trees

In database applications, we usually want to be able to update the information in the database. Therefore, we must consider secondary storage algorithms that not only provide efficient access but also allow for efficient insertion and deletion. Several algorithms have been developed for maintaining balanced trees with logarithmic insertion and deletion routines. The most popular of these algorithms is the B-tree. B-trees are multiway trees that automatically maintain their balance. They also maintain perfect balance: Every leaf node in a B-tree has the same depth. Refinements to the insertion and deletion routines guarantee that the tree will never become too sparse. Before we look at the insertion and deletion algorithms, we must first look at the structure of B-trees.

A B-tree of order m is defined to have the following shape properties:

- Every node has at most m children.
- Every node except the root and the leaves has at least $\lceil m/2 \rceil$ children. The root node has at least two children unless it is a leaf. All leaf nodes are at the same level.
- A nonleaf node with k children contains $k - 1$ keys.

Each node contains a list of keys and pointers in the following pattern:

$$P_0, K_1, P_1, K_2, P_2, \ldots, K_{n-1}, P_{n-1}, K_n, P_n$$

where each K_i is a key and each P_i is a pointer to another node in the tree or a null pointer. Within each node, $K_1 < K_2 < \cdots < K_{n-1} < K_n$. All keys in the subtree pointed to by P_0 are less than K_1. All keys in the subtree pointed to by P_i are greater than K_i but less than K_{i+1}. All keys in the subtree pointed to by P_n are greater than K_n. Figure 7.3 shows a B-tree of order 4.

one node can vary depending on the size of the keys, organization of the data (are only keys kept in the nodes or entire records), and of course, on the size of a block. Block size varies for each system; it can be 2 kbytes, 16 kbytes or more; block size is the size of each node of a B-tree. The amount of information stored in a node of the B-tree can be rather large.

A B-tree of order m is a multiway search tree with the following properties:

1. The root has at least two subtrees unless it is a leaf.
2. Each node, except the root and the leaves, has between $\lceil m/2 \rceil$ and m subtrees.
3. Each node holds one less keys than its number of subtrees.
4. All leaves are on the same level.[1]

A complete, or full, binary tree is a B-tree of order 2 and height h, all of whose leaves are on level h.

A node of a B-tree is usually implemented as a class containing an array to hold the keys and another array to hold pointers to the node's children. A template format for declaring the node type is given below. The node type has a constructor and a destructor. Here is

```
template <class T, int M>
class BTreeNode {
public:
    BTreeNode();
    BTreeNode(const T&);
private:
    bool leaf;
    int keyCount;
    T keys[M-1];
    BTreeNode *pointers[M];
    friend BTree<T,M>;
};
```

Figure 7.1 shows how a [illegible] that is formatted as shown in this paper. This kind of [illegible] is [illegible]. Figure 7.2 shows an example of a B-tree of order 5 which stores integer keys. In this B-tree, the keys appear in the [illegible] objects in the leaf nodes, [illegible] are [illegible] only in the [illegible] of the [illegible] nodes, possibly [illegible] to them. In these cases, the [illegible] is [illegible] objects such that [illegible] if a child [illegible] the data for [illegible] in Figure 7.2, the [illegible] of [illegible] could [illegible] in Figure 7.3. If the [illegible] of [illegible] nodes [illegible] by [illegible] keys and [illegible] would require two [illegible] accesses in the long run. This is [illegible] to [illegible] the [illegible] accesses in the [illegible], since [illegible] [illegible] [illegible]

FIGURE 7-3 One node of a B-tree of order 5 that will be used with an additional illustration.

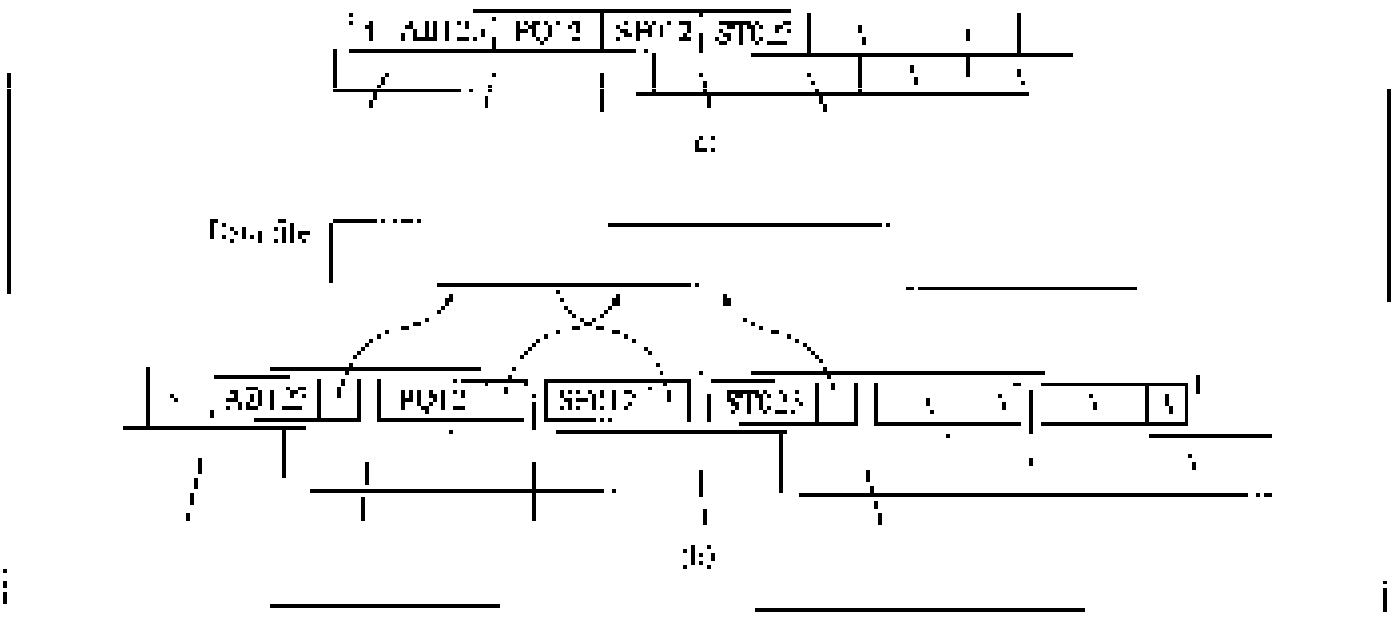
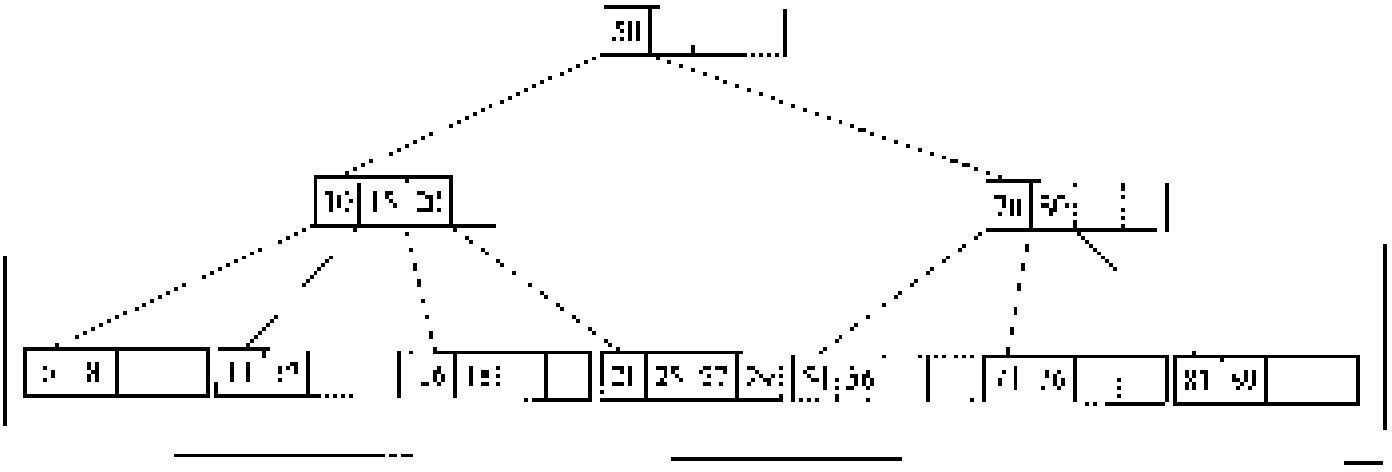


FIGURE 7-4 A B-tree of order 3 shown in its expanded form.



number of search records. The resulting B-tree is not deeper, and search paths through it are not longer than in a B-tree of the same size without duplicates.

From now on, B-trees will be shown in an abstract form without explicitly including the key values or the pointers. Like the one in Figure 7-5.

Searching in a B-Tree

The worst case of searching is when a B-tree has the smallest allowable number of pointers per nonroot node, $q = \lfloor p/2 \rfloor$, and the search has to reach a leaf (either for a successful or an unsuccessful search). In this case in a B-tree of height h there are

$$\begin{aligned}
&1 \text{ key in the root} \\
&2(q-1) \text{ keys in the second level} \\
&2q(q-1) \text{ keys in the third level} \\
&\vdots \\
&2q^{h-2}(q-1) \text{ keys in the lowest level (leaves)}
\end{aligned}$$

With n being the number of keys in the B-tree,

$$n = 1 + \sum_{i=1}^{h} 2q^{i-1}(q-1) \text{ keys in the B-tree}$$

With the formula for the sum of the first n elements of a geometric progression

$$\sum_{i=1}^{n} a_1 = \frac{a_1(r^n - 1)}{r - 1}$$

the number of keys in the worst case B-tree can be expressed as

$$n = 2(q-1)\sum_{i=1}^{h-1} q^{i-1} + 1 = 2(q-1)\left(\frac{q^{h-1} - 1}{q - 1}\right) + 1 = 2q^{h-1} - 1$$

The relation between the number of root keys in any B-tree and the height of the B-tree is then expressed as

$$n \geq 2q^{h-1} - 1$$

Solving this inequality for the height h results in

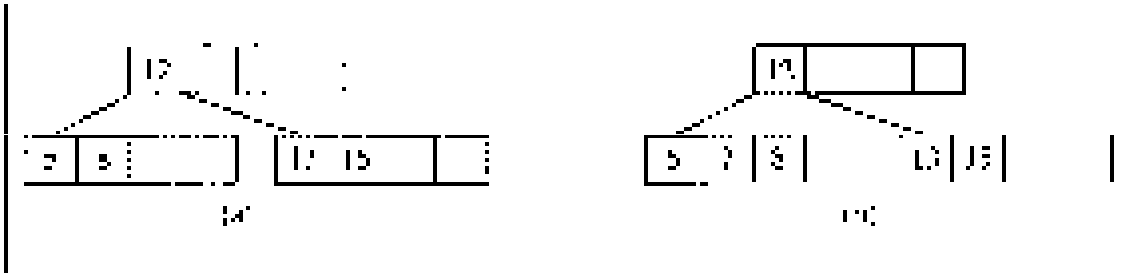
$$h \leq \log_q \frac{n+1}{2} + 1$$

This means that for a sufficiently large order m the height h and hence for a large number of keys n stored in the B-tree. For example, if $n = 200{,}000{,}000$ and $m = 200$, then $h \leq 4$ in the worst case, finding a key in this B-tree requires four seeks. If the root can be kept in memory at all times, this number can be reduced to only three seeks into secondary storage.

An algorithm for finding a key in a B-tree is simple and proceeds as follows:

```
BTreeNode *BTree::Search(KeyType K, BTreeNode *nodeptr)
{
    if (nodeptr != 0)
    {
        for (i = 0; i < nodeptr->KeyCount; i++)
            if (K <= nodeptr->Key[i]) break;
        if (K == nodeptr->Key[i])
            return nodeptr;
        else return Search(K, nodeptr->Ptr[i]);
    }
    else return 0;
}
```

FIGURE 7.5 A B-tree of order 5 after insertion of the number 2 in the tree in Figure 7.4(c).



Inserting a Key in a B-Tree

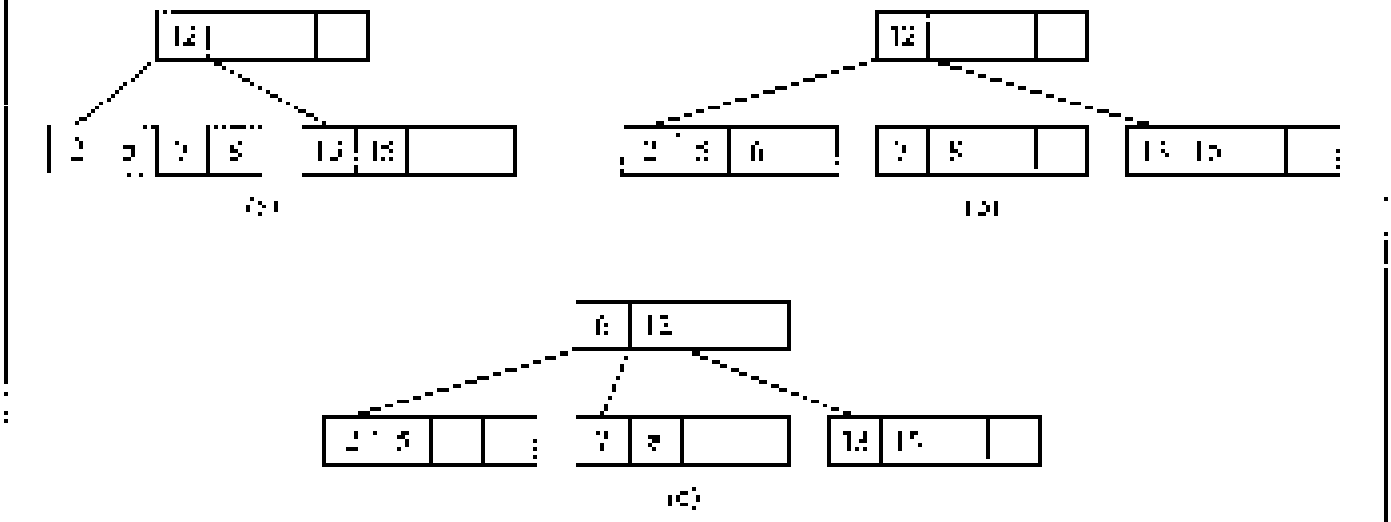
Both the insertion and deletion operations appear at first somewhat discouraging if we remember that all leaves have to be at the same level. Not even balanced binary trees require that. Implementing insertion becomes easier when the strategy of building a tree is changed. When inserting a node in a binary search tree, the tree is always built from the bottom, resulting in unbalanced trees. If the first incoming key is the smallest, then that key is put in the root, and the root does not have a left descendant. Later special provisions are made to balance the tree.

But a tree can be built from the bottom up so that the root is certain always to be there, and only at the end of all insertions can we know for sure the contents of the root. This strategy is applied to inserting keys into B-trees. In this process, given an incoming key, we go directly to a leaf and place it there, if there is room. When the leaf is full, another leaf is created, the keys are divided between these leaves, and one key is promoted to the parent. If the parent is full, the process is repeated until the root is reached and a new root created.

To approach the problem more systematically, there are three common situations encountered when inserting a key in a B-tree.

1. A key is placed in a leaf that still has some room, as in Figure 7.6. In a B-tree of order 5, a new key is placed in a leaf, preserving the order of the keys in the leaf so that keys must be shifted to the right to make room.
2. The leaf in which a key should be placed is full, as in Figure 7.6. In this case, to the key causing overflow and half of the remaining keys from the full leaf are moved to a new leaf. But the new leaf has to be incorporated into the B-tree. The last key of the full leaf is moved to the parent, and a pointer to the new leaf is placed just to the right of this key. This case is called a split. Note that a split creates an additional node in the B-tree. Moreover, such a split guarantees that each leaf node has at least $\lceil m/2 \rceil - 1$ keys.
3. A special case arises if the root of the tree is full. In this case, a new root and a new sibling of the root must be created. Two keys from the old root node must be moved to the new nodes, and one key must be promoted to the new root. In Figure 7.7(a), the root is full. In Figure 7.7(b), a new root is created, and the key C is about to be moved to the parent, but the root has no room. In Figure 7.7(c), the root is split, and one

FIGURE 7.6 Inserting the number 9 into a full leaf



node can't be split because it has only one key. This is followed by inserting the new record and moving the last key from the old node to the new one. Figure 7.6c shows what this looks like for the B+-tree insertion algorithm.

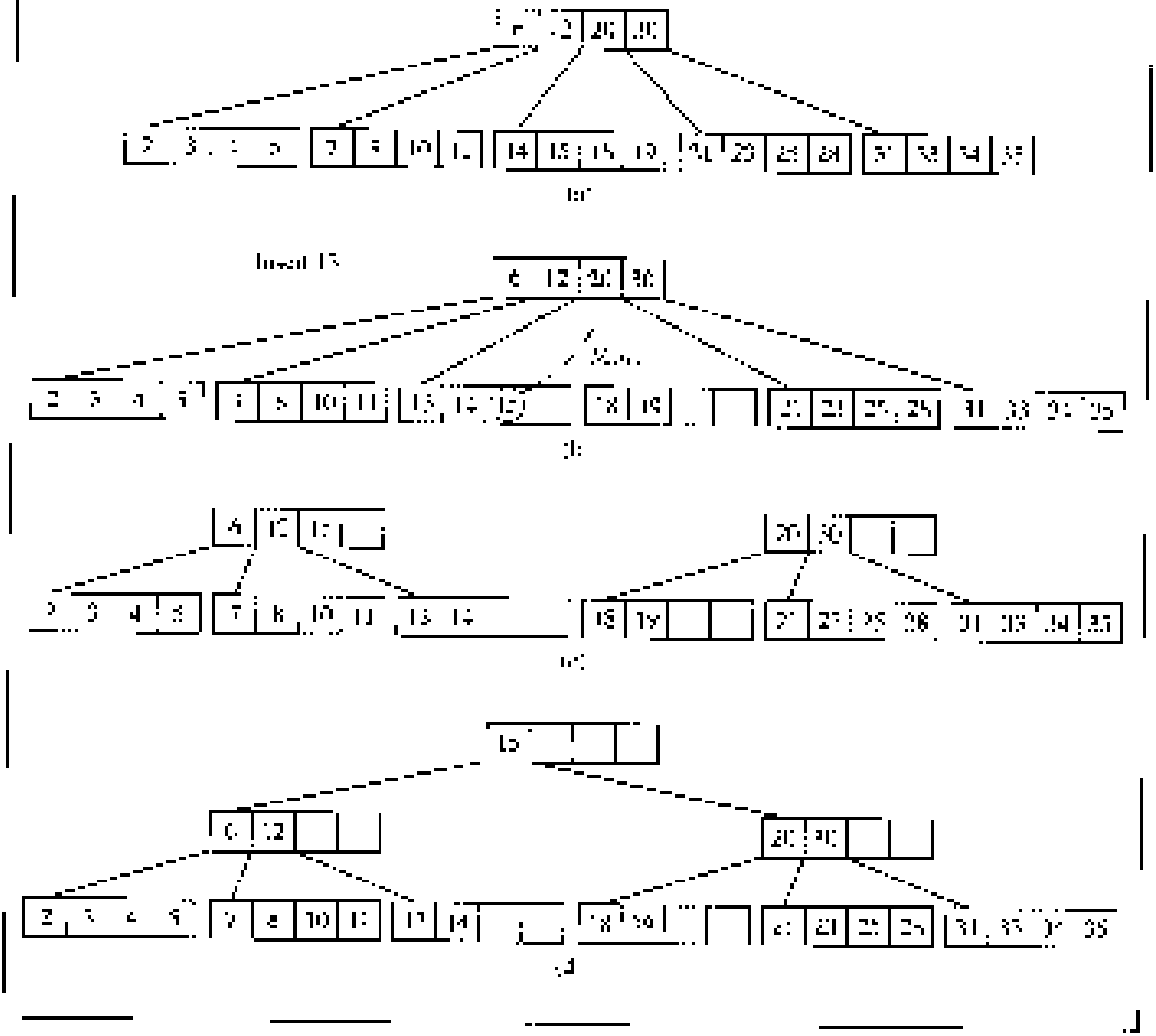
An algorithm for inserting a record into a B-tree follows:

```
BTREE insert(KEY k)
  find a leaf node to insert k;
  while (1) {
    find a proper position in node for key k;
    if node is not full {
      insert k into node in key order;
      return;
    }
    else split node into node and node2; // node = node, node2 is new
      distribute keys and pointers between node and node2;
      middle = largest key in node;
      p = pointer to node;
      if node was the root {
        create a new root pointing to node and node2;
        put k and pointers to node and node2 in the root in key order;
        return;
      }
      else node = its parent; // move up to process the parent of node
```

Figure 7.6 shows the growth of a B-tree of order 5 as the records 9, 5, 6, 12, 3, 2, 15, 18, 7, 8, 10, 4, 11, 13, 14, 16, 17, 1 are inserted in that order. Note that all nodes (except the root) are at least half full.

A search proceeds by performing a sequential search within a node until a key greater than or equal to the target key is found. If the key is equal, then the search is successful. Otherwise, the child pointer to the left of the key is followed to a new node, and the process is repeated.

FIGURE 7.7 Inserting the number 13 into a full root.



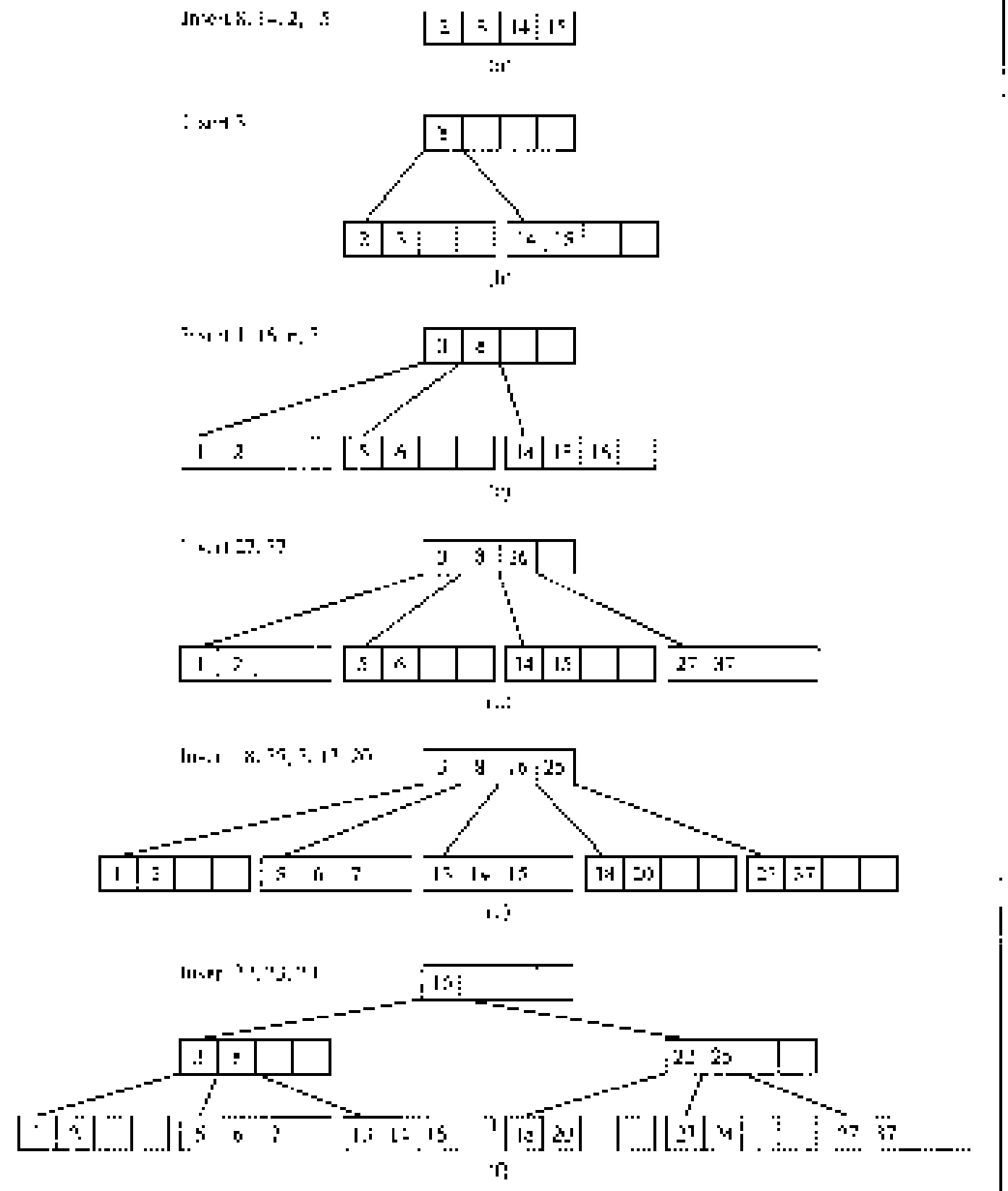
How often are node splits expected to occur? A split of the root node of a B+-tree creates two new nodes. All other splits add one new node to the B+-tree. During the construction of a B+-tree of p nodes, $p - 1$ additional nodes, to be partitioned, will be created. The height of the B+-tree also increases by 1. There are at least

$$1 + \lceil (m/2) - 1 \rceil (p - 1)$$

leaves. The exact split is with respect to the number of keys in the B+-tree and is equivalent to

$$\frac{p - 1}{1 + \lceil (m/2) - 1 \rceil (p - 1)}$$

FIGURE 7.8 Building a B-tree of order 5 with the BTreeInsert() algorithm.



After dividing the interval and denoting the left-hand side by x, we observe that $x \cdot p^{t-1}$ and $(1-p)^{t-1}$ are the increase and decrease, respectively, in the average probability of a split at

$$\frac{1}{\ln\frac{p}{1-p}}$$

For example, for $p = 0.1$ this probability is equal to .25; for $p = 0.01$ it is .05; and for $p = 0.001$, it is .008, and so on. The larger the separation between p and $1-p$, the more frequently splits occur.

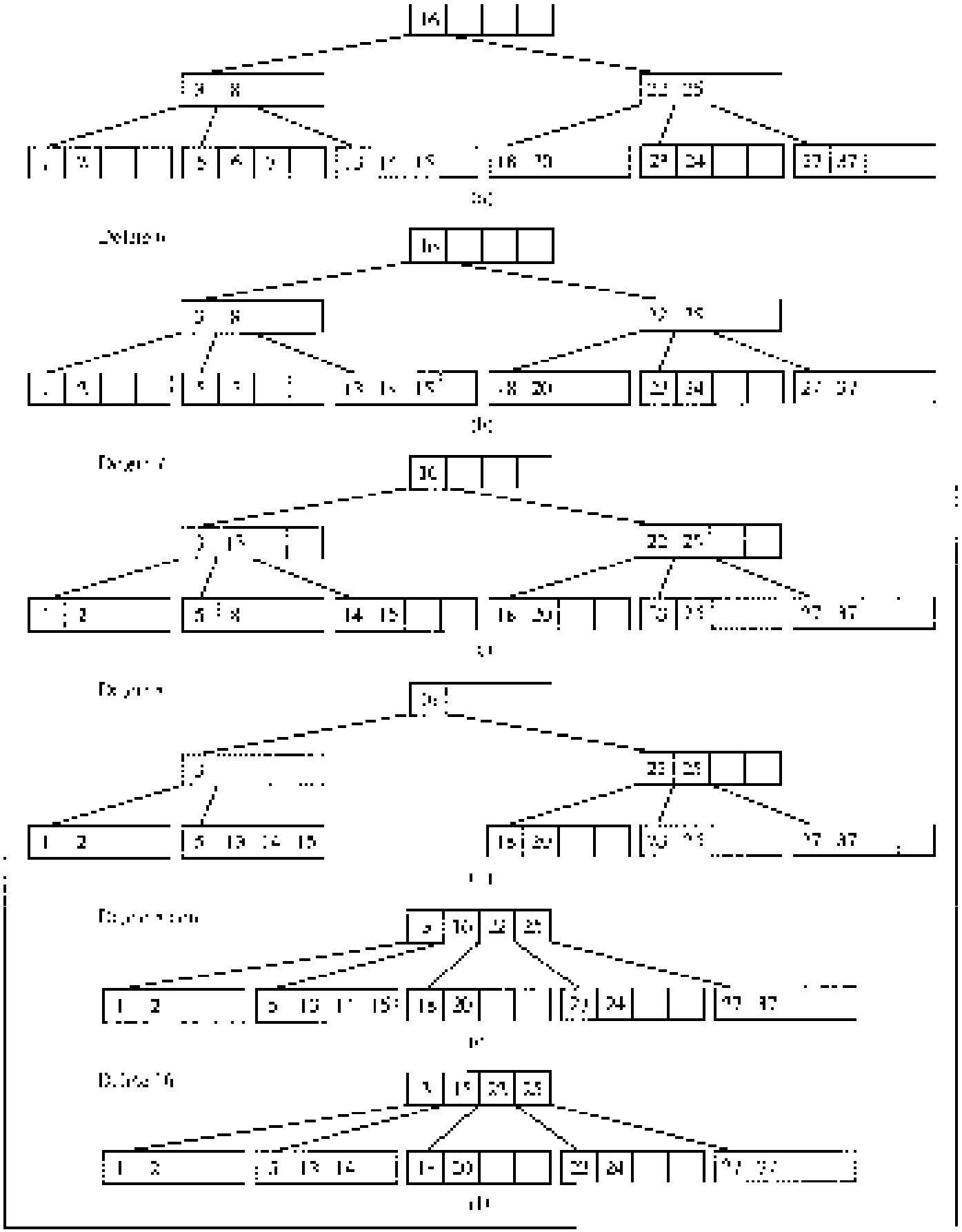
Deleting a Key from a B-Tree

Deletion is the opposite of insertion, although it is a little more complex. Care has to be taken to avoid allowing any node to be less than half full (except the root) after a deletion. This is because nodes sometimes have to be merged.

In deletion there are two main cases: deleting a key from a leaf, and deleting a key from a nonleaf node. In the latter case, we will use a procedure similar to one we met when deleting from a binary search tree (Section 5.3.1).

1. Deleting a key from a leaf.
   1.1 Search for the key to be deleted. If it is not found, then the deletion is impossible. Otherwise, delete the key from the leaf. (This is the interesting case, as it may violate the B-tree properties.)
   1.2 If after deleting K, the number of keys in the leaf is less than $\lceil m/2 \rceil - 1$, then the leaf underflows.
      1.2.1 If one of the left or right sibling contains more than the minimum number of keys, then all keys from the leaf and both siblings are redistributed between them so that they contain the same number of keys, to within one. This can be done by concatenating the keys from the two siblings and the key from the parent that separates them (in the correct order) and then putting the first $\lceil (2m-1)/2 \rceil$ keys back in the left sibling, the next key in the parent, and the remaining keys in the right sibling. (See Figures 7.9–11.)
      1.2.2 If the leaf underflows and the number of keys in its siblings is $\lceil m/2 \rceil - 1$, then the leaf and a sibling are merged: the keys from the leaf, from the sibling, and the separating key from the parent are all put in the leaf, and the sibling node is discarded. The keys in the parent are moved left to fill the space (see Figures 7.9–12). This can initiate a chain of operations if the parent underflows. The parent is now treated as though it were a leaf, and either step 1.2.1 is repeated or, if step 1.2.1 cannot be executed, or the root of the tree has been reached, this is the inverse of insertion's case 2.
         1.2.2.1 If the root is the case reached, it is merged with its sibling, which is guaranteed to be the root with only one key. The key from the root and its sibling, along with the pointers to the root's children, are put in the node, which becomes the new root. Thus, the only case when keys disappear at the root is when the number of the tree is decreased (see Figure 7.9–13). This is the inverse of insert's case 3.

Figure 7.9 Deleting keys from B trees.



2. Replace key by its inorder successor. This can be done by replacing key with the smallest key in the subtree rooted at node's rightmost child. Then delete the inorder successor from the leaf node in which it was found. (See Figure 7.9.)

Algorithm for deletion operation:

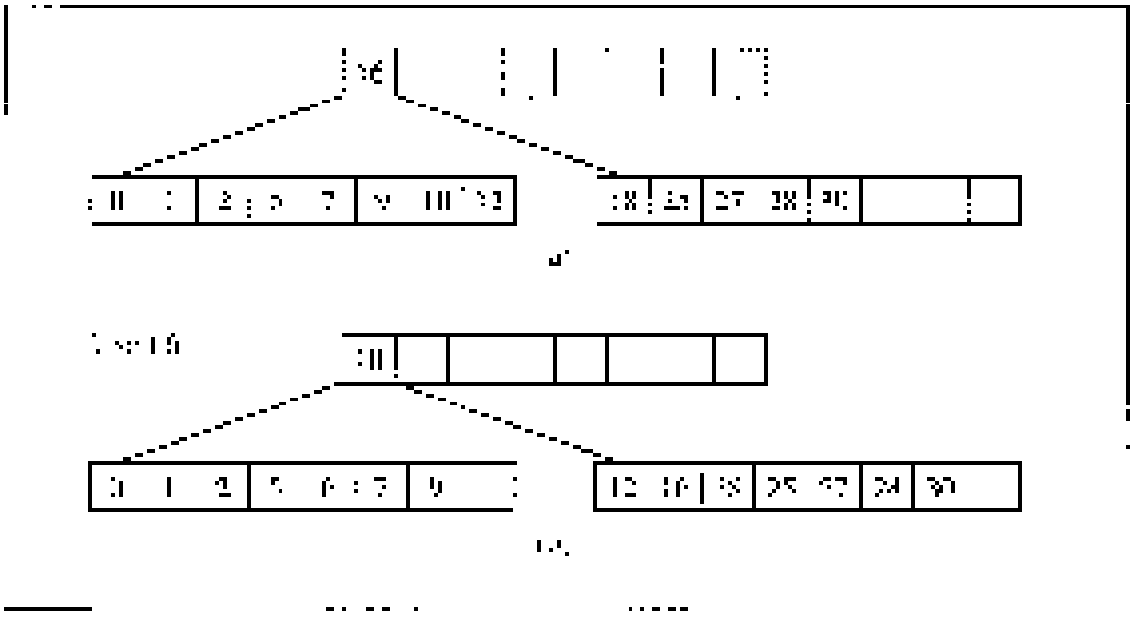
```
BTreeDelete (K)
    node = BTreeSearch(T, root)
    if (node != null)
        if node is not a leaf
            Replace K with its inorder successor S of K
            Copy S over K in node
            node = the leaf containing S
            Delete S from node
        else delete K from node
        while (1)
            if node is the root and empty
                return;
            else if node is adequate and complete
                Update branching between node and its siblings
                return;
            else if node's parent is the root
                if the parent has only one key
                    Merge node, its sibling, and the parent to form new root
                else Merge node and its sibling
                return;
            else Merge node and its sibling
                node = its parent;
```

Performance according to this definition are guaranteed to be at least 50% full, and may be as bad as 50% if space is basically wasted. How often does this occur? It depends how often the deletion algorithm must reconsider one of these conditions. Empirical studies of B-tree algorithms, however, indicate that a series of pseudo-random insertions and deletions of B-trees approximately 50% full (Yao 1978), after which the changes in the percentage of occupied cells are very small. But it is very unlikely that any B-tree will ever be utilized to its fullest because some additional storage is always needed.

7.1.2 B*-Trees

Since each node of a B-tree represents a block of secondary memory, we usually use the maximum size of secondary memory which is expected to be available for storing keys in the root resulting in primary memory. Therefore, the fewer nodes there are created, the better.

FIGURE 7.10 Overflow in a B+ tree is handled by redistributing keys between a neighboring node and its sibling.



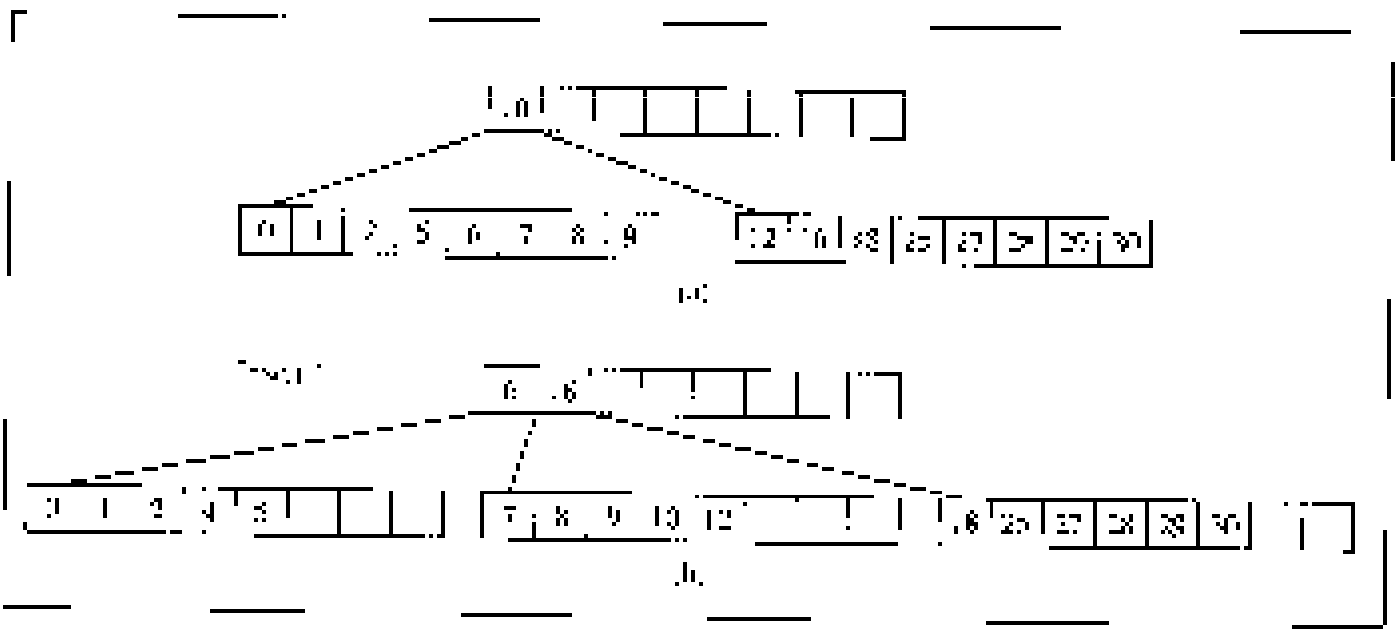
A B+ tree is a variation of the B tree originally defined by Donald Knuth and named by Douglas Comer. In the B+ tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes. Also, all leaves are linked in sequential order. This last feature of B+ trees is important because it allows fast sequential processing of the entire file. The higher-level nodes of the B+ tree can be thought of as an *index* to the records. The algorithms for searching, inserting, and deleting records from a B+ tree are extensions of the corresponding algorithms for B trees. For a good description of B+ trees, see Comer (1979).

A variation of the B+ tree, known as the *B* tree*, is defined by specifying that each node must be at least two-thirds full (instead of only one-half full). Instead of splitting a node when it overflows, keys are redistributed among neighboring nodes. When two nodes are full and a record must be inserted into one of them, the two nodes are divided into three nodes. These nodes will be at least two-thirds full because the two nodes contained enough keys for three nodes that were only one-half full.

The algorithms for searching, inserting, and deleting keys from B* trees are left as exercises. Knuth (1973b) suggests that the B* tree usually yields the best performance if the order of the tree is chosen to be $2k + 1$, where k is the number of keys that will fit in a disk page. Thus, a B* tree of order 101 would have pages that could hold 100 keys and 102 pointers. Such a tree would be 67 percent full on the average.

Note that as the order of the tree increases, the number of levels required to hold a given number of keys decreases. This means that the number of disk accesses required to find a key decreases. For example, a B tree of order 10 can hold 1,000,000 keys in seven levels. A B tree of order 100 can hold the same number of keys in four levels. Since each level of the tree corresponds to a disk access, the larger tree can access any key in at most four disk accesses. Of course, the larger tree requires more space for each node, and more time to search within each node. Nevertheless, the reduction in disk accesses usually makes the trade-off worthwhile.

Figure 7.11 The node and its 14 keys are split in a B+-tree, split causes a new node to be created and keys are distributed between these nodes.



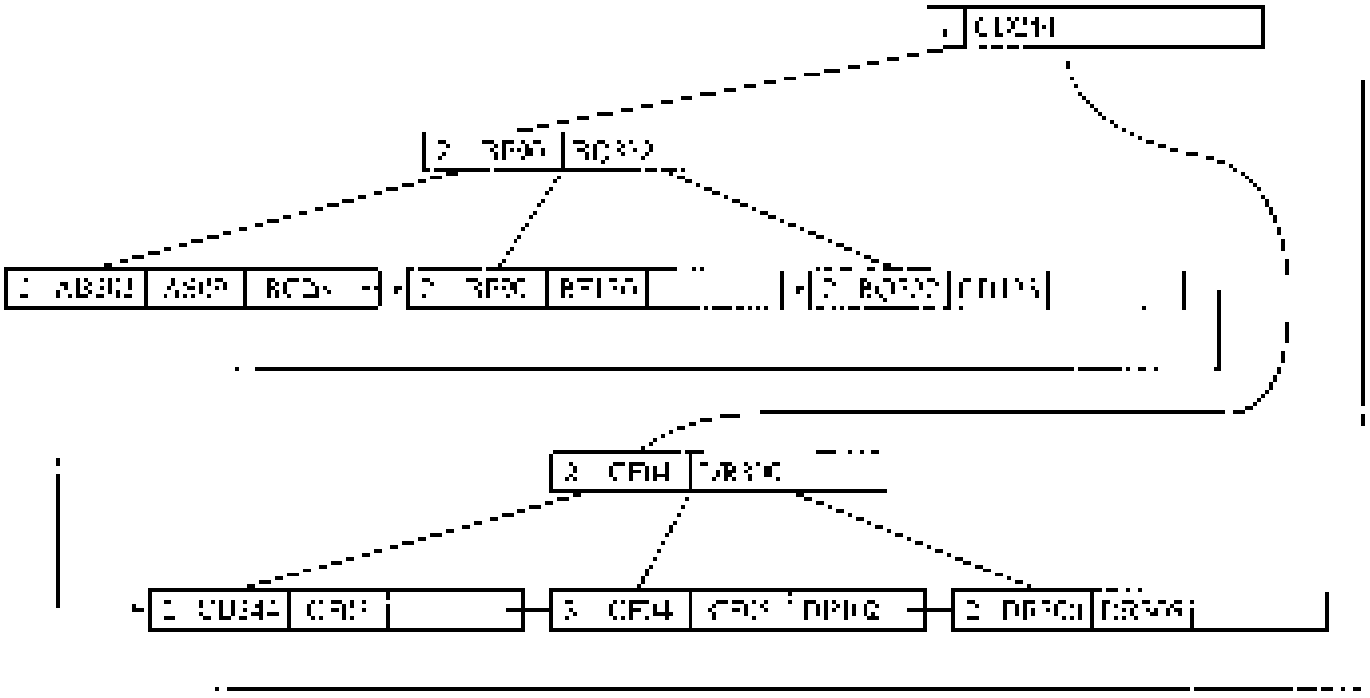
7.1.3 B+-Trees

Since the node of a B-tree commonly corresponds to a memory page or block, the swapping of a page from one node to another requires a time consuming page change. Therefore, we would like to reduce the number of accesses to pages. We can happen if we organize all the keys in the B-tree be printed in ascending order? An inorder traversal can be used which is easy to implement, but for the internal nodes, only one key is displayed on the terminal even though the page has to be accessed. Therefore, we would like an efficient B-tree to allow us to process data sequentially in a faster manner than using inorder traversal. A B+ tree is a solution presented in 1973.[1]

In a B-tree, references to data are made for every node of the tree, but in a B+ tree, B+ tree, all the data are made available from the leaves. The internal nodes of a B+-tree are indexes for fast access of data. The part of the tree is called an index set. The leaves are called the sequence set. The leaves are linked sequentially to form a sequence set that contains all the keys in the data given in ascending order. Hence a B+ tree is mainly used to provide fast random as well as sequential access to the data. Figure 7.12 contains an example of a B+ tree. Note that the internal nodes store keys, pointers, and a few of the leaves store keys, but these do not contain any data file associated with the keys. Pointers to the next leaf.

Other than the B+ trees are not very different from operations on B-trees. Insertion may cause overflow which will require some more operations putting the keys and pointers in

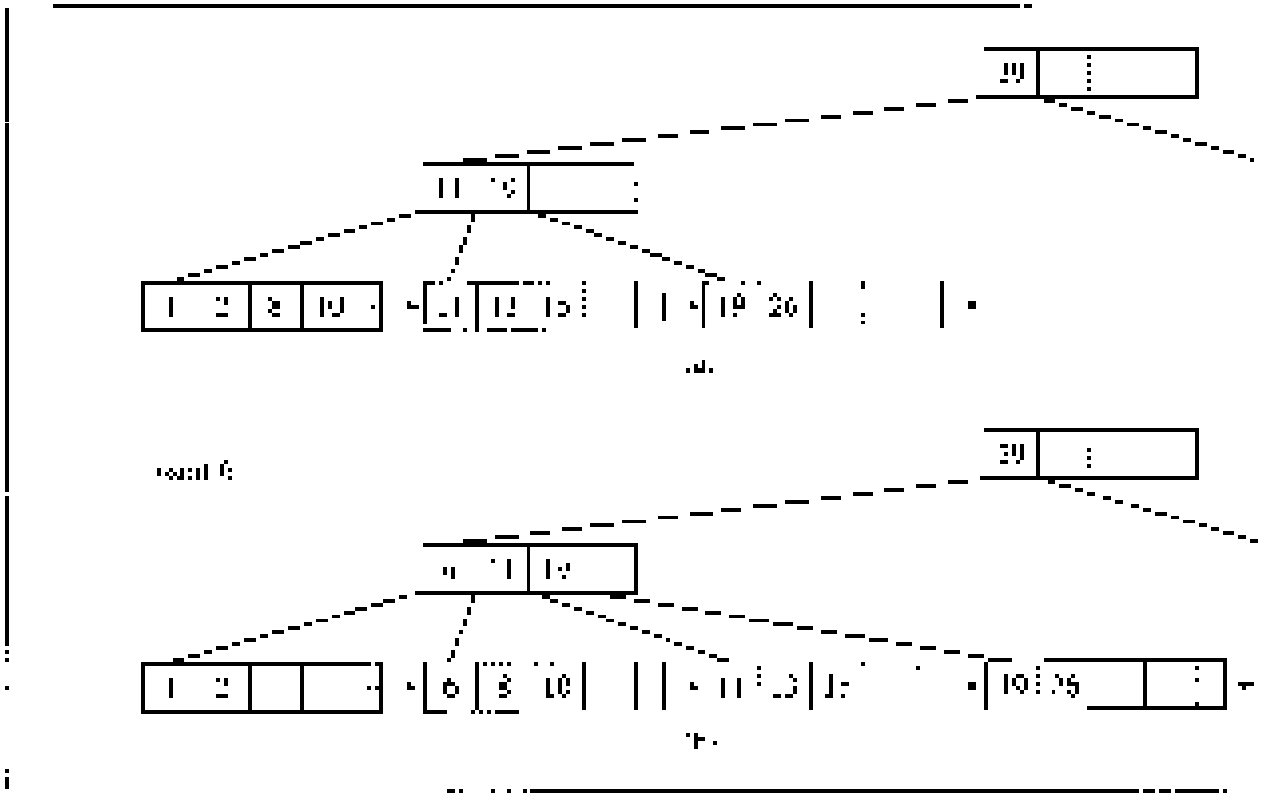
[1] Unfortunately, there are also these names to refer to the slightly variations of B-trees, and thus B+ trees.

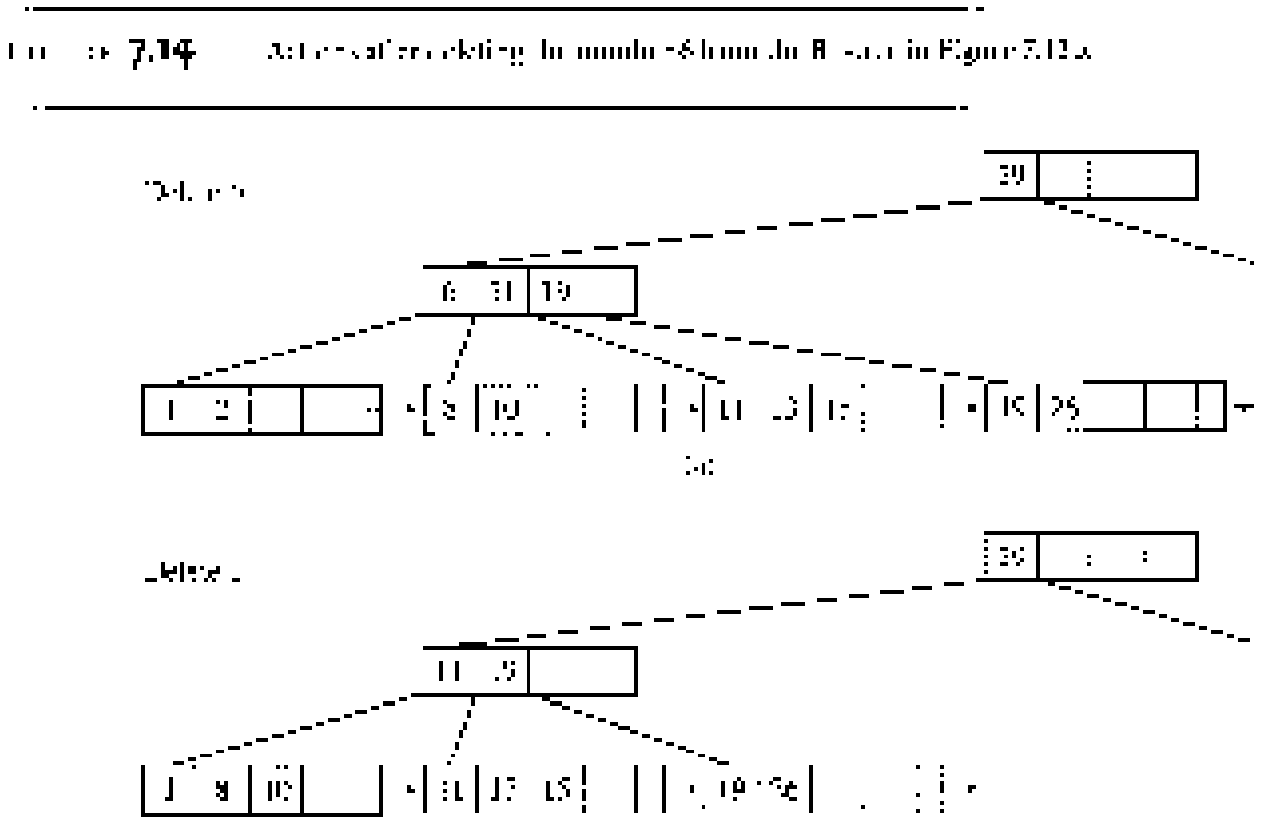


moved. No changes are made in the index set. If a key is inserted into a full leaf, the leaf is split, the new leaf node is included in the sequence set, keys are distributed evenly between the old and the new leaves, and the first key from the new node is copied (not moved) as in a B+ tree) to the parent. If the parent is not full, this may require reorganization of the keys of the parent (see Figure 7.10a). If the parent is full, the splitting process is performed the same way as in B+ trees. After a few insertions we have

Deleting a key from a leaf leading to an underflow requires pooling the remaining keys in order. No changes are made to the index set. In particular, if a key which occurs once in a leaf is deleted, then it is simply deleted from the leaf but can remain in the internal node. The reason is that it still serves as a proper guide when navigating down the B+ tree because it still properly separates keys between two adjacent children even if the separator itself does not occur in either of the children. The deletion of key 6 from the tree in Figure 7.10b results in the tree in Figure 7.11a. Note that the number 6 is not deleted from an internal node.

When the deletion of a key from a leaf causes an underflow, then either the keys from this leaf and the keys of a sibling are redistributed between the leaf and its sibling or the leaf is deleted and the remaining keys are included in its sibling. Figure 7.11b illustrates the latter case: after deleting the number 2 an underflow occurs and two leaves are combined to form one leaf. The first key from the right sibling of the node remaining after merging is copied to the parent node, and keys in the parent are pulled down. Both these operations require updating the separators in the parent. As, removing a leaf may trigger merges in the index set.





7.1.4 Prefix B+-Trees

The key occurrence in a leaf node or in an internal node of a B+-tree has been given to direct access only from the leaf because the key retained in the node is still a good predictor for subsequent searches. So it very rarely does not matter whether a key in an internal node is in any sense real. What counts is that it is an acceptable separator for keys in adjacent children; for example, for two keys K_1 and K_2, the separator must meet the condition $K_1 < \text{separator} \leq K_2$. This property of the separator keys is also obtained if we make keys in internal nodes as small as possible by removing all redundant contents from them and still have properly working B+-trees.

A simple prefix B+-tree stores keys and separators in such a way that the chosen separators are the shortest prefixes that allow us to distinguish two neighboring index keys. For example, in Figure 7.11, the left child of the root has two keys, 19830 and 30322. If a key is less than 30 we take the first leaf; it is chosen if it is less than 19831; the second leaf is the right guess. But observe that we also have the same result if instead of 19830 we just use 198 as the key and instead of 30322 we only use 30. Only if a key is equal to 198 or 30 do we have to look at the next character before choosing the shortest prefix of each key. If any key is less than 198, the search ends up in the first leaf, and if the key is less than 30 the second leaf is chosen. The result is the same as before. Reducing the size of the separators in the branch nodes does not change the result of the search, but only makes separators smaller. As a result more separators can be placed in the same node, thereby making such a node capable of more children. The entire B+-tree can have fewer levels, which reduces the branching factor and makes processing the tree faster.

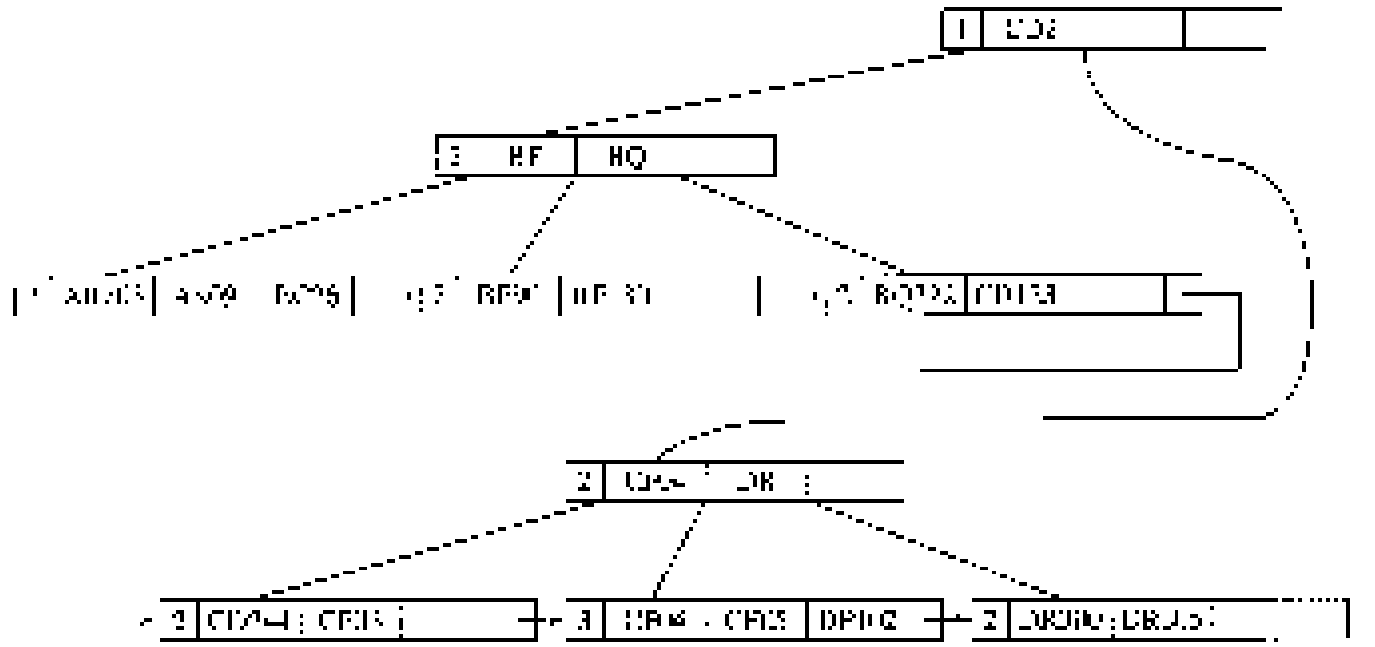
This reasoning does not stop at the level of parents of the leaves. It can extend over to any other level so that the entire interior of a B+-tree is filled with prefixes (see Figure 7.12).

The operations on a simple prefix B+-tree are much the same as for ordinary B+-trees with some modifications to account for prefixes and separators. In particular, after a split, the first key from the new node is not just moved or copied to the parent, but the shortest prefix is found which discriminates it from the node's left neighbor key in the old node, and then the shortest prefix is placed in the parent. Deletion, however, some separators stored in the index array may turn out to be no longer, but not more discriminating than they should be and immediately should not.

The idea of using prefixes as separators can be carried even further. Two observations that prefixes of prefixes can be omitted in lower levels of the tree, which is the idea behind a prefix B+-tree. This method works particularly well if prefixes are long and redundant. Figure 7.13 contains an example. Each key in the tree has a prefix "AB 2X", and this prefix appears in all internal nodes. This is redundancy. Figure 7.13 shows the same tree with "AB 2X" stripped from prefixes in children of the root. To restore the original prefix, the key from the parent node, except for its last characters, becomes the prefix of the key found in the current node. For example, the first cell of the child at the root in Figure 7.13 has the key "08". The last character of the key in the root is discarded and the obtained prefix "AB 2X" is apart from a "08". The new prefix, "AB 2X08" is used to determine the direction of the search.

The efficiency of prefix B+-trees in experimental runs indicates that there is almost no difference in the time needed to execute algorithms on B+-trees and simple prefix

FIGURE 7.15 A B+ tree from Figure 7.14 presented as a simple prefix B+ tree.



B+ trees, but prefix B+ trees need to be fetched more often in terms of absolute access time. The difference between these trees is the number of times the disk is accessed. For trees of four accesses or less, two trees of four accesses, each simple prefix B+ tree and prefix B+ trees require only two level accesses (Bayer and McCreight 1977). This is because a simple prefix B+ tree stores a variable-length portion of the prefix B+ tree's key, which is usually much shorter than the original key.

7.1.5 Bit-Trees

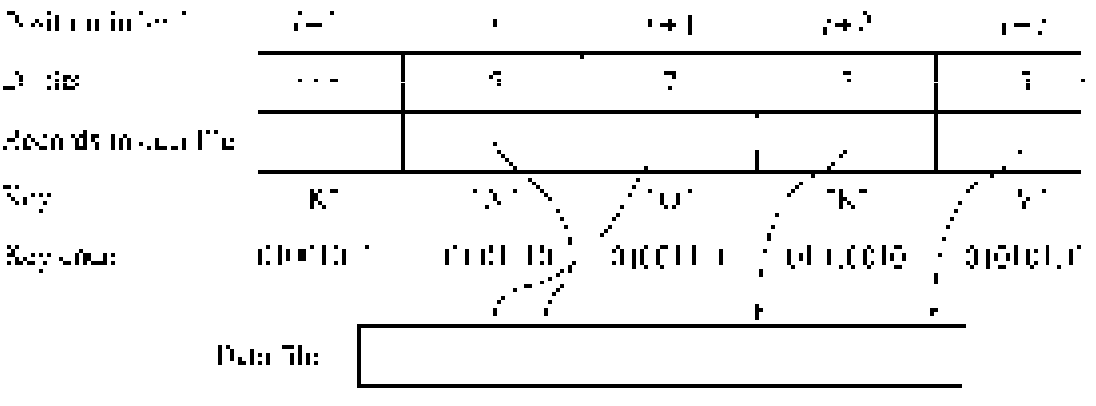
A very interesting approach is, in a sense, taking us back to the examination of the prefix B+ tree method. In this method, bytes are used to specify separators in B+ trees. The bit-level is reached (Bergman 1992).

The bit tree is based on the concept of discrimination bits. A discrimination bit, usually, is the number of the most significant bit at which either of two keys, K and L, and K is the key length in bits. For example, the D bits for the strings "Bob" and "Bobby", whose ASCII codes are 01000010 and 01000110, is 5, the position at which the first difference between the two codes has been detected. D("Bob", "Bobby") = 5.

A bit tree uses the discrimination bits to separate keys in the leaves of the tree. The bit tree is a B+ tree. This means that it contains keys and entire records from which these keys are extracted in a data file. But the leaves contain not just

FIGURE 7.16 (a) A simple prefix B+ tree and (b) its abbreviated version presented as a prefix B+ tree.

FIGURE 7.17 A k-d-tree and its data file.



...

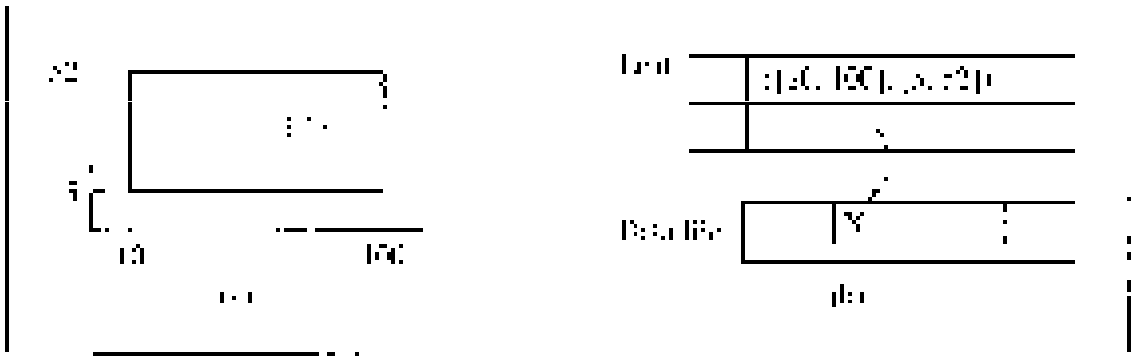
...

...

```
...
    B = ...
    ...
        ...
            B = ...
        ...
    ...
    ...
    ...
    ...
```

...

Figure 7.18 An object X in the Cartesian plane enclosed tightly by the rectangle $[a, b][c, d][e, f]$. The rectangle parameters which uniquely identify X are stored in a leaf of an R-tree.



1. In the first iteration of the for loop for $D_1 = 3$ in key "3" = 3 01 11 01 is checked and because $3 \le 5$, it is assigned R_1.
2. In the second iteration for $D_2 = 7$ is tested. It is 0, but nothing is skipped, as required by the else statement, and right away a 0 bit is found which is smaller than 7.
3. The third pass finds $D_3 = 5$ and $5 \le 5$, so R_3 is set.
4. In the fourth iteration for $D_4 = 5$ is checked again and because it is 1, R_4 is assigned R_2. This is the last entry in the leaf, the algorithm is terminated and R_2 is properly returned.

What happens if the desired key is not in the data file? We can try to locate "8" (1000) using the same assumptions as 3 and 5. But $D_1 = 8$ is 0, so the pointer with 0 bit 7 is skipped, and since bit $D_2 = 3$ (or "8") is 1, the algorithm would return record R_2. To prevent this, a successful search should include the record's found key to verify that it is the desired key. If not, a negative number should be returned to indicate failure.

7.1.6 R-Trees

Spatial data is the kind of objects which are utilized frequently in many areas. Computer assisted design, geographical data, and VLSI design are examples of domains in which spatial data are created, searched, and altered. This type of data requires special structures to be processed efficiently. For example, we may request that all counties in an area specified by geographic coordinates be printed, or that all buildings within walking distance from city hall be identified. Many different data structures have been developed to accommodate this type of data. One example is an R-tree [Guttman 1984].

An R-tree of order m is a tree-like structure containing at least m entries in one node for some $m \ge 2$ (except the root, which may have fewer) and at most $2m$ entries per node (except the root). Hence each node is required to be at least half full.

A leaf in an R-tree contains entries of the form $(rect, id)$ where $rect = (x_1, y_1, \ldots, x_n, y_n)$ describes an n-dimensional rectangle and id points to the record that contains the smallest rectangle containing the object. For example, the entry in a leaf node containing the object X in Figure 7.18 is the pair $([a, b][c, d][e, f], id)$.

A nonleaf node in an R tree has the form (*n*, *ptr*), where *n* is the smallest rectangle that can accommodate all the rectangles found in child. The structure of an R tree is not identical to that of a B tree. The former can be viewed as a series of rectangles and pointers corresponding to these keys.

Inserting new rectangles in an R tree is made in B tree fashion with splits and recombinations. A crucial operation is finding a proper leaf in which to insert a rectangle *r*. When moving down the R tree, the subtree chosen in the current node is the one that corresponds to the rectangle requiring the least enlargement to include *r*. If a split occurs, two encompassing rectangles have to be created. The detailed algorithm is more involved since, among other things, it must decide which rectangles of a node being split the algorithm should get to use. Rectangles which end up as rectangles of the two resulting nodes and are minimum in size.

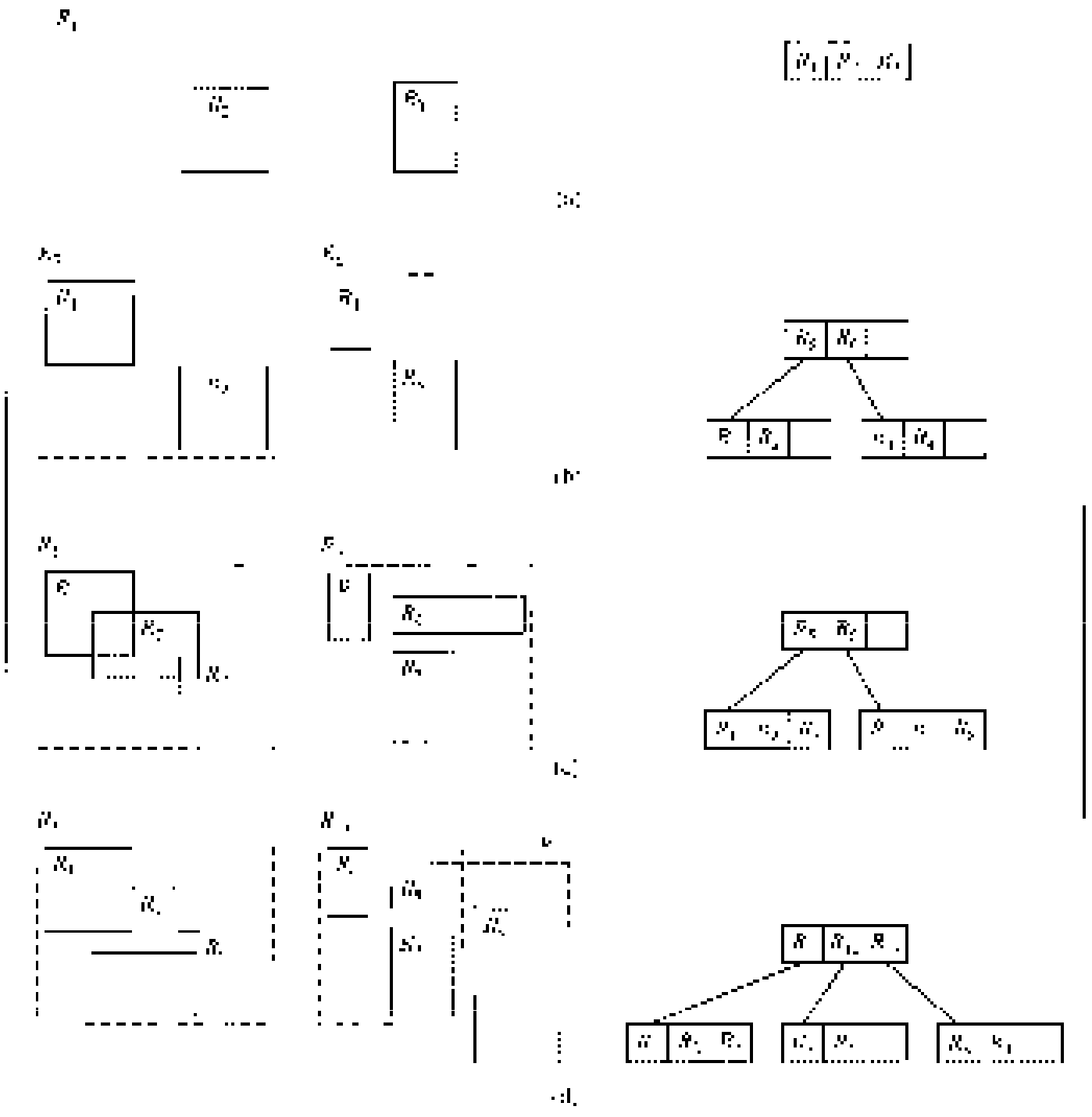
Figure 7.19 contains an example of inserting four rectangles into an R tree. After inserting the first three rectangles R_1, R_2, and R_3 only, the tree is as in Figure 7.19a. Inserting R_4 causes a split, resulting in the creation of two encompassing rectangles (Figure 7.19b). Inserting R_5 changes nothing, and inserting R_6 causes rectangle R_2 to be extended to accommodate R_6 (Figure 7.19c). Figure 7.19d shows another split after entering R_7 in the R tree. R_{12} is discarded, and R_{13} and R_{14} are created.

A rectangle *r* can be contained in more than one encompassing rectangle but it can be stored in only one leaf. The choice, which in need may take a wrong path at some level (before it becomes apparent) is made in favor of the first rectangle found in a node at that level. For example, rectangle R_{10} in Figure 7.19b is stored in both R_{13} and R_{14} since R_{10} is before R_{11} in the tree, the search chooses the middle child when looking for R_{10}. However, if R_{10} preceded R_{11} in the tree, following the path corresponding with R_{14} would be unsuccessful. For large and high-dimensional trees, the overlapping becomes excessive.

A modification of R trees called R^+ trees removes this overlap (Stonebraker, Sellis, and Hanson 1986; Sellis, Roussopoulos, and Faloutsos 1987). The encompassing rectangles are no longer overlapping, and each encompassing rectangle is associated with all the rectangles in the tree that it overlaps. Hence the data rectangle can be found in more than one leaf. For example, Figure 7.20 shows an R^+ tree constructed after the data rectangle R_3 was inserted into the R tree in Figure 7.19a. Figure 7.20 is identical to Figure 7.19c. Note that R_3 can be found in two leaves since it is contained by two encompassing rectangles, R_{12} and R_{13}. Operations on R^+ trees make it difficult to assure whether further manipulations of the nodes are at least 40% full.

7.1.7 2–4 Trees

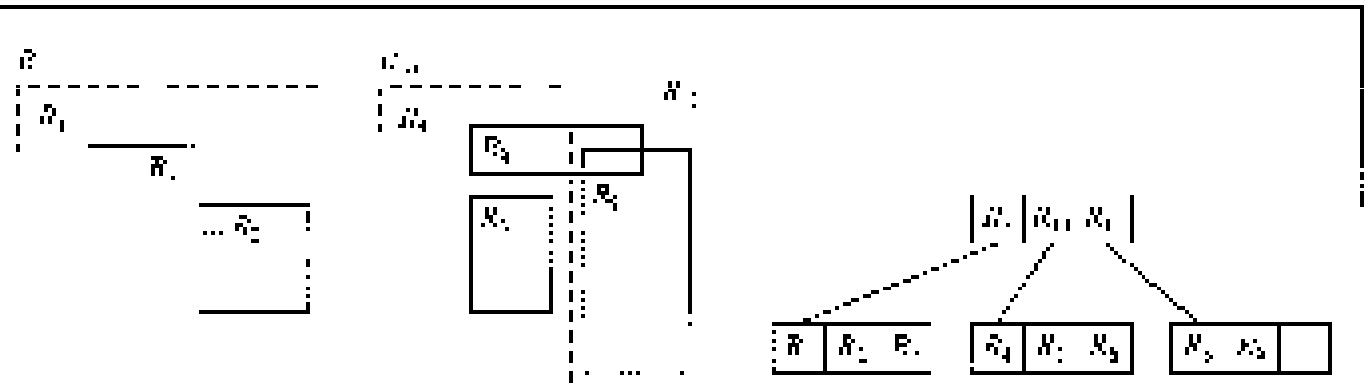
This section discusses a special case of B trees, 2–4 trees (of order 4). Type B trees was first discussed by Rudolf Bayer who called it a symmetric binary B tree (Bayer 1972), but it is more commonly called a 2–3–4 tree or a 2–4 tree. As the name implies, the nodes in 2–4 trees, on the other hand, contain one, two, or at most three keys and can be stored in arrays of size 3. Unlike the elements in very large B tree leaf nodes, which are stored on disk, there seems to be no reason to store the relatively small nodes of 2–4 trees on disk.



Although B-trees were introduced as a technique for handling data on secondary storage, it does not mean that they have to be used exclusively for that purpose.

We spend an entire chapter discussing binary trees, in particular, binary search trees, and develop algorithms that allow quick access to the information stored in them

FIGURE 7.20 A B-tree representation of the R-tree in Figure 7.19(a) after inserting the rectangle R_6 in the tree in Figure 7.19(b).



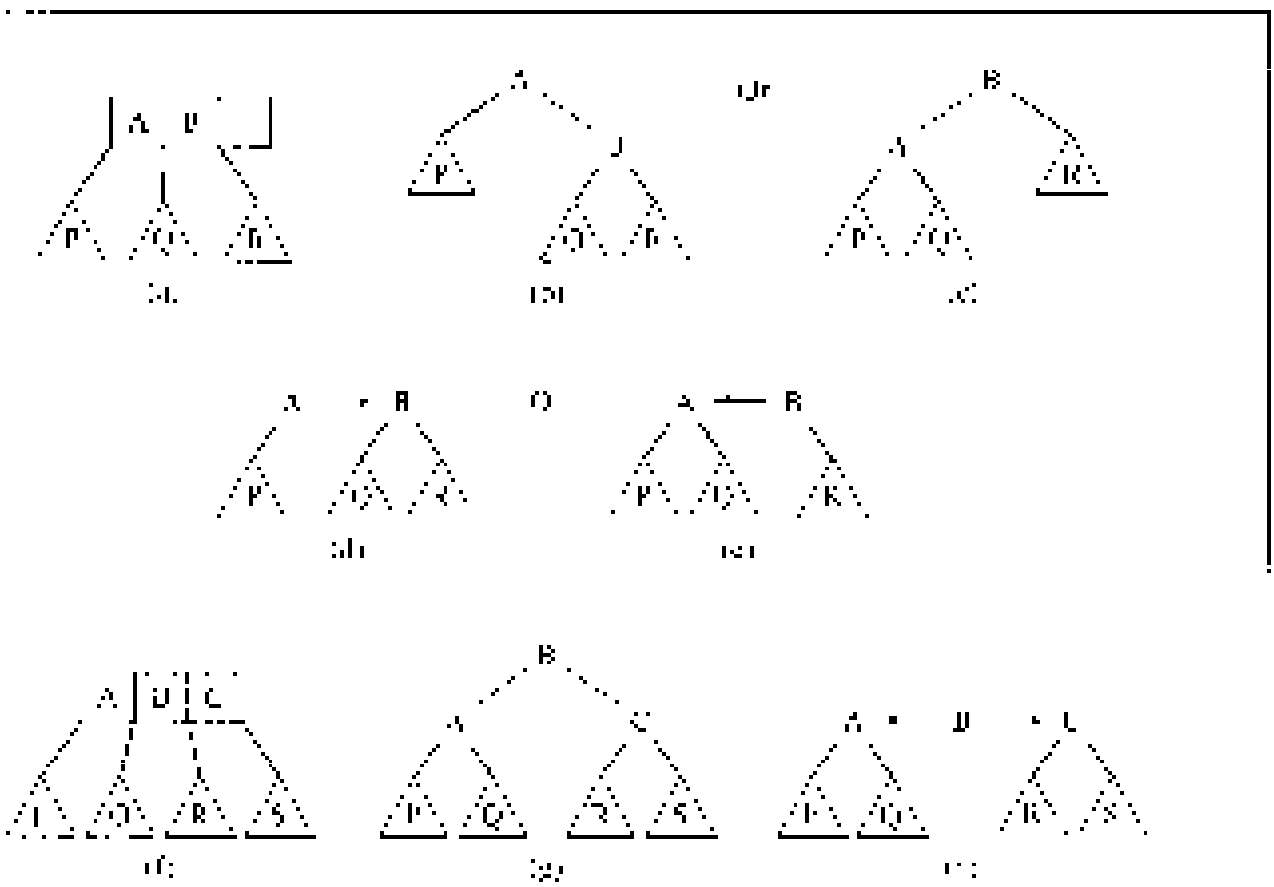
trees. Can B-trees offer a better solution to the problem of searching or traversing binary trees? We now return to the topic of binary trees and processing data in memory.

B-trees are well suited to challenge the algorithms used for binary search trees, since a B-tree's structure needs to be balanced. No special treatment is needed in addition to building a tree. Building to tree's balances for the most time. Instead of using binary search trees, we may use B-trees of small order such as 2–4 trees. However, since trees are implemented as structures similar to B-trees, there are three locations per node for more than three keys and five locations per node for more than four pointers in the worst case. Half of these are unused, and on the average, 69% are unused. Since space is much more expensive in main memory than in secondary storage, we would like to avoid this unused space. Therefore, 2–4 trees are transformed into binary tree form in which each node holds only one key. Of course, the transformation has to become a one-to-one and onto mapping to guarantee retrieval of the original B-tree form.

To represent a 2–4 tree as a binary tree, two types of links between nodes are used. One represents links between nodes representing keys belonging to the same node of a 2–4 tree, and another represents regular parent-child links. Several different terminologies for these two types of pointers are used. Typically, pointers and B-pointers (Guibas and Sedgewick in their discussion of red-black trees use the names red and black pointers. Not only are the names different in the trees, but also there is a bit difference in the pictures. Figure 7.21 shows nodes with two and three keys, which are called 3-nodes and 4-nodes, and their equivalent representations. Figure 7.22 shows a complete 2–4 tree and its binary tree equivalent. Note that the red links are drawn with dashed lines. The red-black tree terminology is the most common. In terms of a binary tree, the horizontal and vertical tree structure is retained, but the shape of the 2–4 tree is not as apparent. Although the red-black tree is shown as though it were on the same level, this is not necessarily the case. Red links could, in fact, be of any length. The red-black tree is not unique.

Both red-black trees and 2–4 trees are balanced binary trees. Each node has two pointers, which can be interpreted in two ways. We make a distinction between the two interpretations by giving each pointer a color. One color (the red one) is used

FIGURE 7.21 (a)–(c) A node represents 3-nodes in two possible ways by red black trees and 4-nodes in two possible ways by red-red vertical lines. (d)–(f) 4-node replacements by 3-nodes and 2-nodes by horizontal-vertical lines.



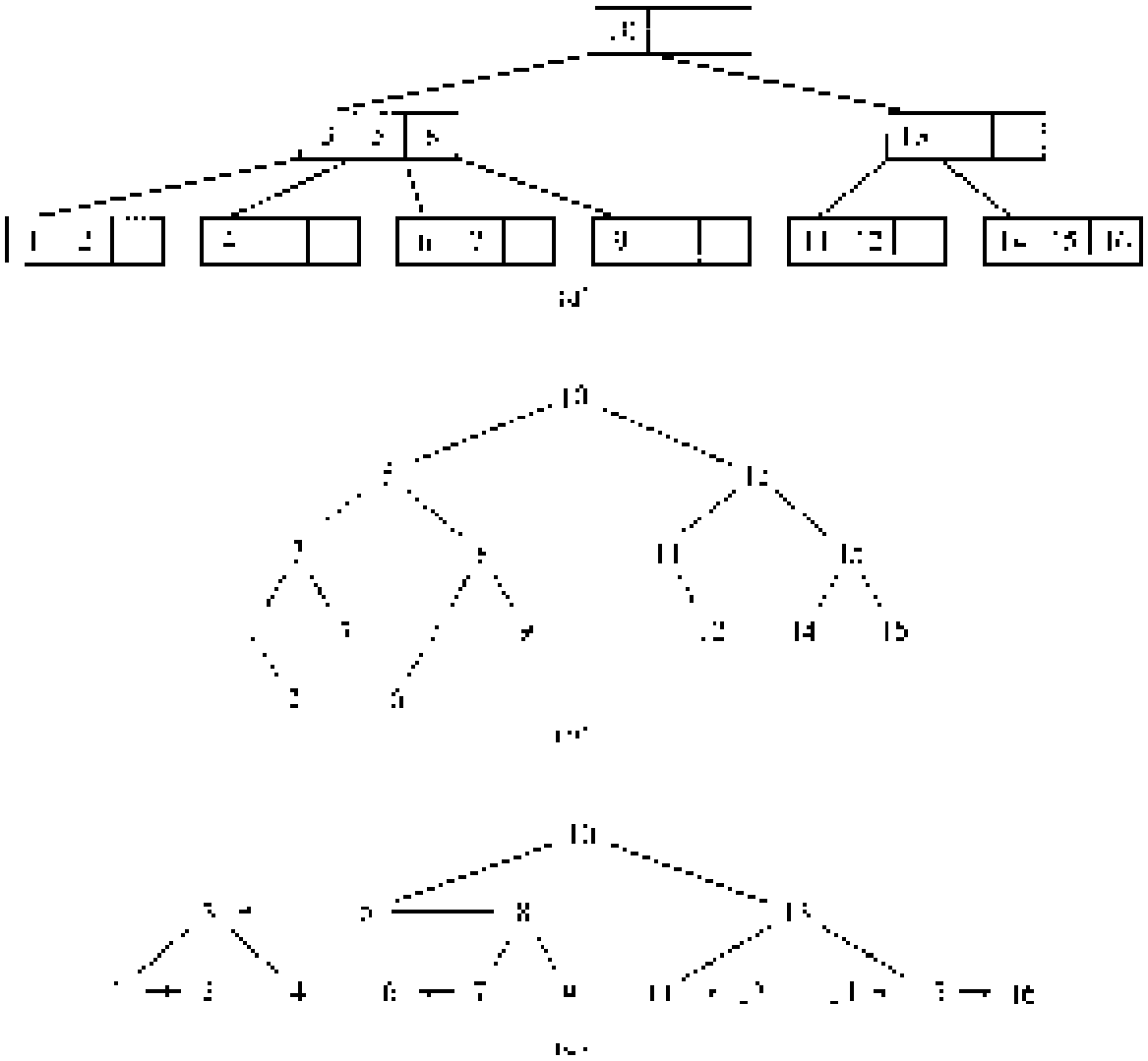
The operations we carried out on horizontal-vertical trees should be the same as on binary trees. Even though the implementation is much more involved, the only search we have to do to find a key is in the horizontal-vertical trees, no distinction is made between the different types of connections. We can use the same search algorithm as for binary search trees. If the key is found, stop. If the key in the current node is larger than the one we are looking for, move to the root of the subtree to the right subtree.

Before discussing insertion, note that horizontal-vertical trees have the following properties:

- The path from the root to any leaf contains the same number of vertical links.
- No path from the root can have two consecutive horizontal links in a row.

Insertion restructures the tree by adding one more node and one more link to the tree. Should it be a horizontal or vertical link? Deletion restructures the tree as well by removing one node and one link, but this may lead to two consecutive horizontal links. These constraints are not as straightforward as for binary search trees, since

FIGURE 7.22 (a) A 2-4 tree represented by a red-black tree. (b) The red-black tree with horizontal and vertical pointers.



(a)

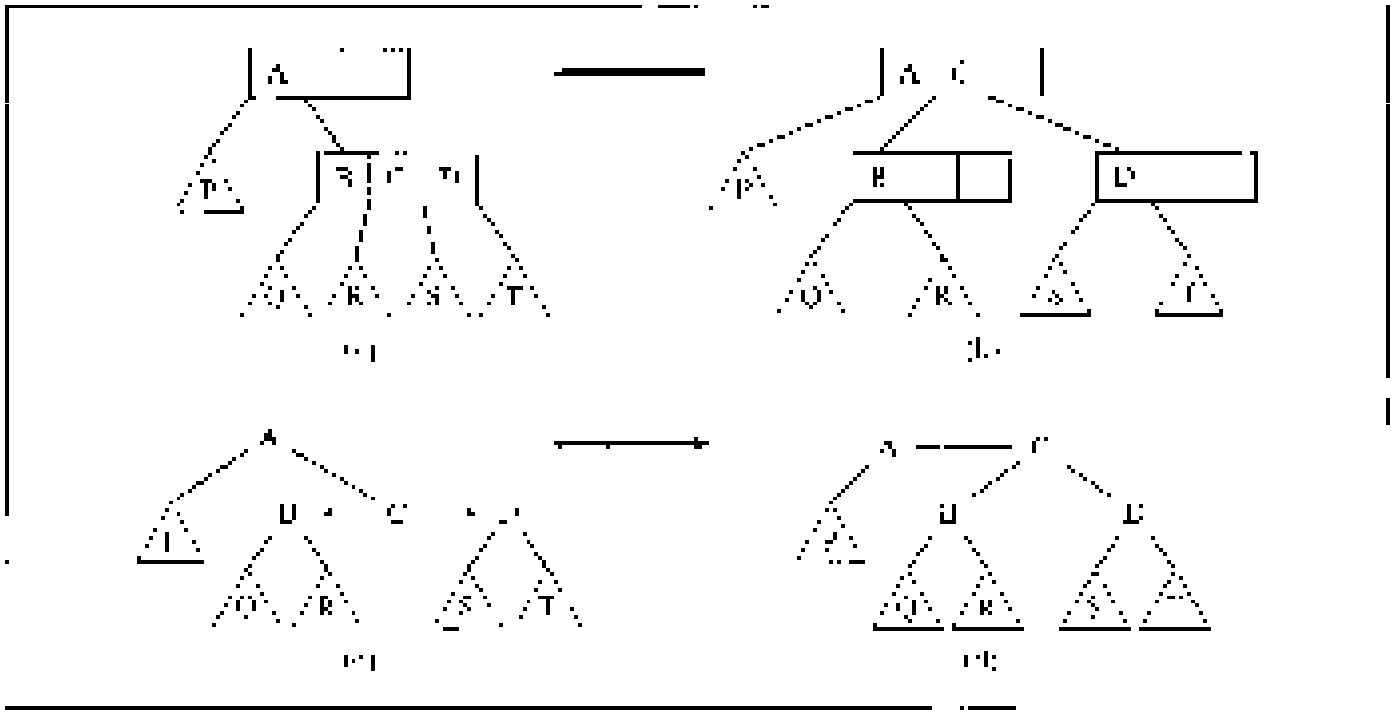
(b)

(c)

Sometimes, when we insert, splitting a node requires moving keys to the nodes' parents. Let's look at how.

Suppose we want to split node p in Figure 7.24a. We've already discussed how to split node p when its parent is a 2-node. We split p into two nodes, and the middle key moves up to the parent. The parent becomes a 3-node. But what if the parent is already a 3-node? Then we split the parent before we can move the middle key up. Because this splitting process may propagate all the way up to the root, we may have to split all the nodes on the path from the root to the node we originally wanted to split. Figures 7.23a and 7.23b show an example. Splitting node p with keys 4, 6, and 5 requires creating a new node. The two nodes that we split (Figure 7.23a) are the full and the nodes after splitting are not full (5 and 7). Properly colored pointers (red) indicate splitting nodes in the 2-4 tree results in properly colored nodes. However, the same operations are performed on the horizontal

FIGURE 7.23 (a)–(d) Splitting a 4-node attached to a node with one key and 2 children. (e)–(h) The same split as a horizontal-vertical tree, with splitting only in these two nodes.



vertical pointers, as in Figure 7.23e–h. Figure 7.23a and 7.23b illustrate splitting a 4-node attached to a 2-node, and the operation sequence changing only pointers from horizontal to vertical and back from vertical to horizontal. Only two bits are needed.

Restricting ourselves to operations on top-down B-trees that take the following steps (or a similar set), we show that such trees can be represented by splitting at the first node encountered that is horizontal and that is a pair node.

In our implementation (see Figure 7.24), the split node is the 4-node in Figure 7.23a before applying a flip-flopping of the two horizontal pointers. Figure 7.23b and 7.23c are equivalent representations that only use two bits, not three (Figures 7.24c and 7.24d).

Figure 7.23 indicates that there are four 4-node B-tree configurations in a horizontal-vertical tree. Therefore, Figure 7.23a can be attached not only to the tree in Figure 7.23b, but also by two more configurations as shown in Figure 7.25. Changing the labels in Figure 7.23b to the two configurations in Figure 7.25 results in all four configurations. Figure 7.23b and Figure 7.25a–c are the four configurations. In each case, the three pointers have to be followed to obtain the correct search path. Figure 7.23b and Figure 7.25a–c are the four configurations. In each case, the three pointers are flipped, and the result is Figure 7.23c or Figure 7.23d.

Figure 7.26a contains a node with a 4-node attached to it. Figure 7.26b is a 2-3 tree before splitting. Figure 7.26b is a 2-3 tree after splitting. Applying the flipping of the tree in Figure 7.26a yields the configuration in Figure 7.26b with two consecutive horizontal links. To restore the leaves-level property, two more pointer flips are needed. Note: One isolated horizontal link, which is followed by no

FIGURE 7.24 (a) Inserting 3 under a node with two keys in a 2-3 tree and (b) the similar operation performed on one possible horizontal representation of the two nodes.

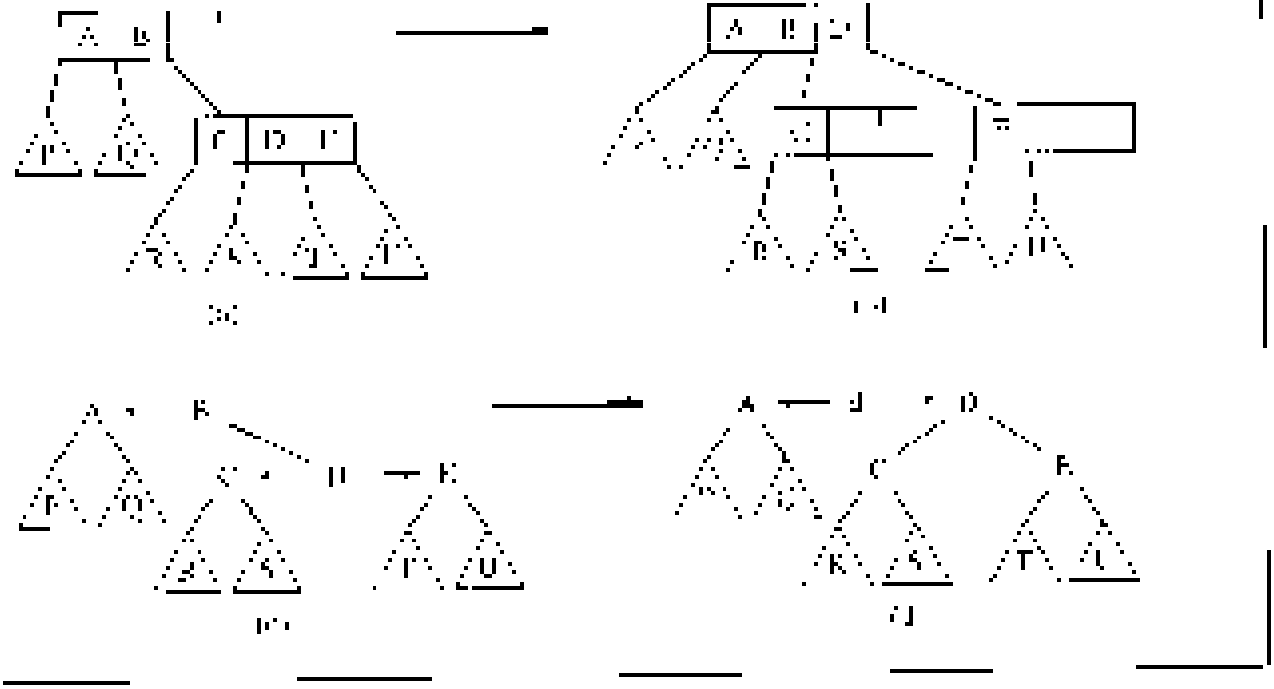


FIGURE 7.25 Fixing a horizontal-vertical conflict. (a) Incorrect horizontal link.

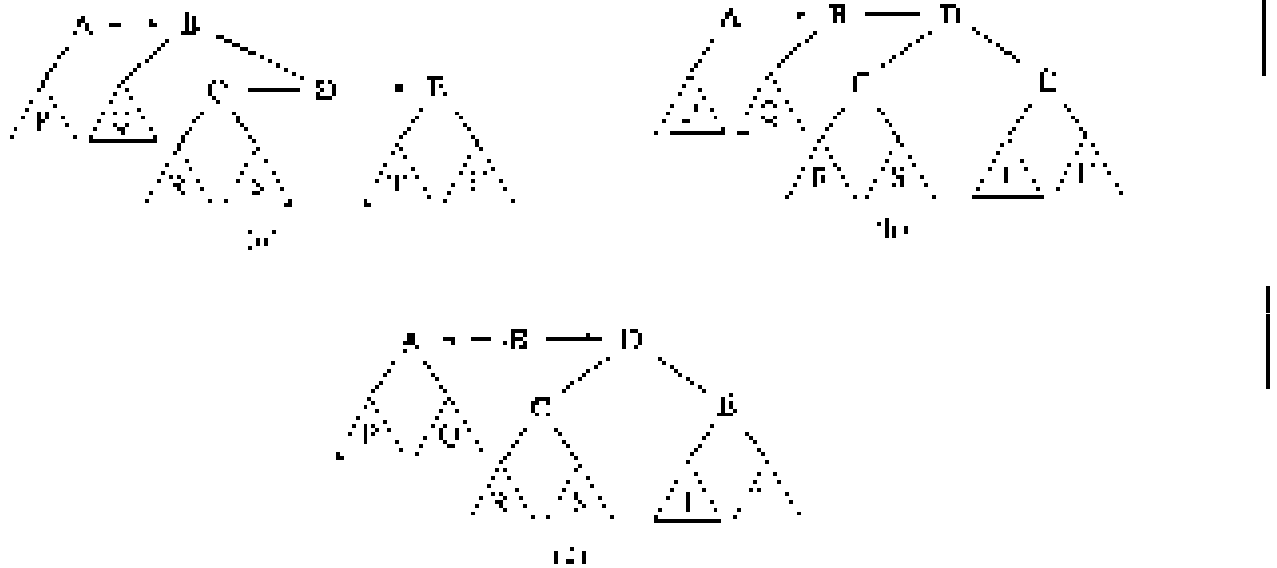
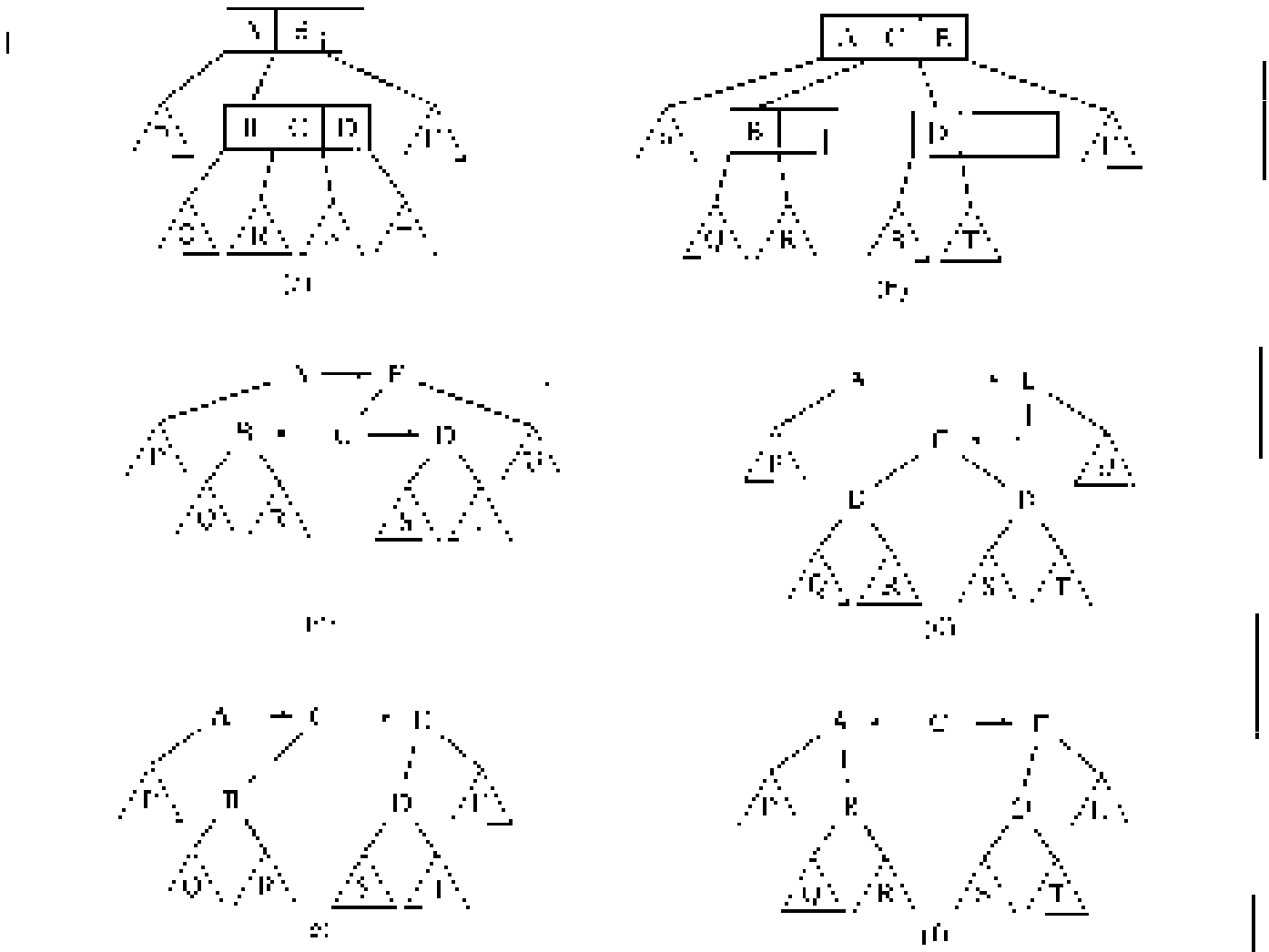


Figure 7.26 A 4-node associated with a 2-3 tree



(a) (b) (c) (d) (e) (f)

Figure 7.26c and then node C about node A, which is also full, as we go up the path. Figure 7.26d will lead to the tree in Figure 7.26f.

We prevent all these configurations by splitting a 4-node as we pass it on the way down the tree. This ensures that the parent of any node we split is not a 4-node. Thus, the number of consecutive nodes that must be updated is limited. This fact is reflected in the following insertion algorithm.

```
void TwoThreeDictionary::insert(int k)
{
  TwoThreeNode *p = root;
  if (root->isFourNode()) {
    root = new TwoThreeNode(root);
    root->split(0);
  }
  while (p->getChild(0) != 0) {
    p = p->findChildForKey(k);
    if (p->isFourNode()) {
      p->getParent()->split(p->whichChild());
      p = p->getParent();
    }
  }
```

```
            while stack[top].node != NULL {
                top++;
                stack[top].node = stack[top-1].node->left;
                stack[top].index = 0;
            }
            // backtrack to next node to be processed
            top--;
            while top >= 0 && stack[top].index == 1 {
                // visit node at stack[top].node
                top--;
            }
            if top >= 0 {
                // visit node at stack[top].node
                stack[top].index = 1;
                stack[top+1].node = stack[top].node->right;
                stack[top+1].index = 0;
            }
        }
    }
```

Figure 7.17 A nonrecursive inorder traversal using a stack

Figure 7.17 contains a nonrecursive inorder traversal algorithm. Note that a node is pushed onto the stack before its left subtree is traversed. When a node is popped from the stack, its left subtree (if any) has been traversed, and the node itself is visited. Then its right subtree is traversed.

How efficient is this algorithm? Like any algorithm that traverses a tree, it takes $O(n)$ time, where n is the number of nodes in the tree. Is it faster than a recursive traversal? Not by any significant amount. Every node is pushed onto the stack once and popped from the stack once. The maximum height of the stack is $h + 1$, where h is the height of the tree. Since the height of a tree with n nodes is at least $\lg n$ and at most $n - 1$, the space requirement for the stack is between $\lg n + 1$ and n. Thus, the space requirement for the stack is $\Omega(\lg n)$ and $O(n)$.

The space requirement for the stack in a recursive traversal is also $\Omega(\lg n)$ and $O(n)$, which is the same as for the nonrecursive traversal. The nonrecursive traversal may appear less elegant than the recursive traversal, but it is not significantly more complicated. The nonrecursive traversal algorithm is also faster by a constant factor than the recursive traversal algorithm, because the overhead of stacking and unstacking activation records is avoided. This difference is usually small enough to be unimportant, but if the algorithm is executed frequently, it may be worthwhile to use the nonrecursive version.

7.1.8 Sets and Multisets in the Standard Template Library

The set container in the Standard Template Library stores objects (known as keys) in order. Unlike the containers we have seen so far, it is not a sequence. Instead, it is an associative container. Associative containers are discussed in more detail in Section 7.3. However, we must discuss the set container briefly now because it is the most appropriate container to use for the task. Multiple copies of the same key are not allowed in a set. However, the multiset container is identical to the set container except that multiple copies of the same key are allowed. The insert() operation is used to add keys to a set or multiset. The erase() operation removes keys. Duplicate keys are ignored by insert() and erase() does not complain if you ask it to remove a key that is not present. The find() operation searches for a key and returns an iterator that points to the key if it is found. Otherwise, it returns an iterator that points to the end of the container. The iterator can be used to access the key. The following example illustrates the use of these operations:

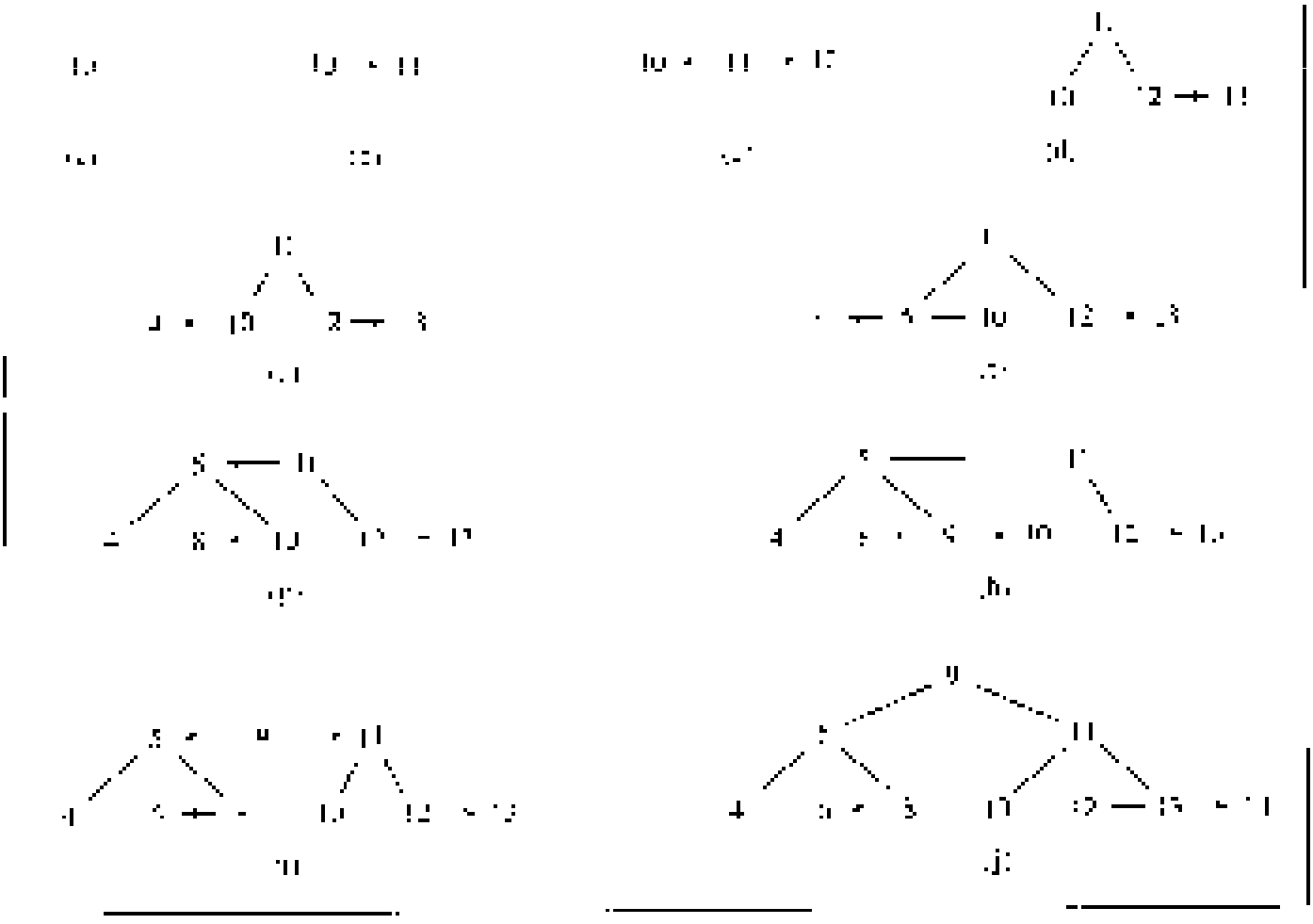


Figure 7.27 Building a horizontal version by inserting numbers in the sequence 8, 11, 2, 13, 4, 5, 8, 9, 5, 4.

whether an element is in the set or not. Only one copy of each element is stored in the set, but a new element requires allocating memory for the new element and placing it in a proper cell of the vector, and the complexity of the operation is the same as in the array case. Clearly, to speed up the use of vectors, we should keep the vector sorted. Such a data structure is called an ordered vector. The algorithms for insertion and deletion have the same complexity, but the flexibility of the vector...

Multisets (see Section 7.1.3) are similar to sets. Figure 7.28 shows the two versions. For each version, we provide a virtual base class that contains all of the same parameters. Second, instead of a pure virtual base class, we provide a template class with a member function `compare` that takes two objects of type `T` and returns `-1`, `0`, or `1` depending on whether the first is less than, equal to, or greater than the second. The `multiset` class allows for multiple copies of the same element. We can be used to check whether an insertion is successful or not. We always do. The implementation of some member functions for unordered and ordered multisets is left as an exercise. Figure 7.29 shows the ordered multiset class. The relation is specified by the user as a function object passed to the constructor. The relation is stored in the member function `key_compare`.

FIGURE 7.28 An example of a set with a multiway B-tree (top part) and two equivalent red-black trees (bottom part).

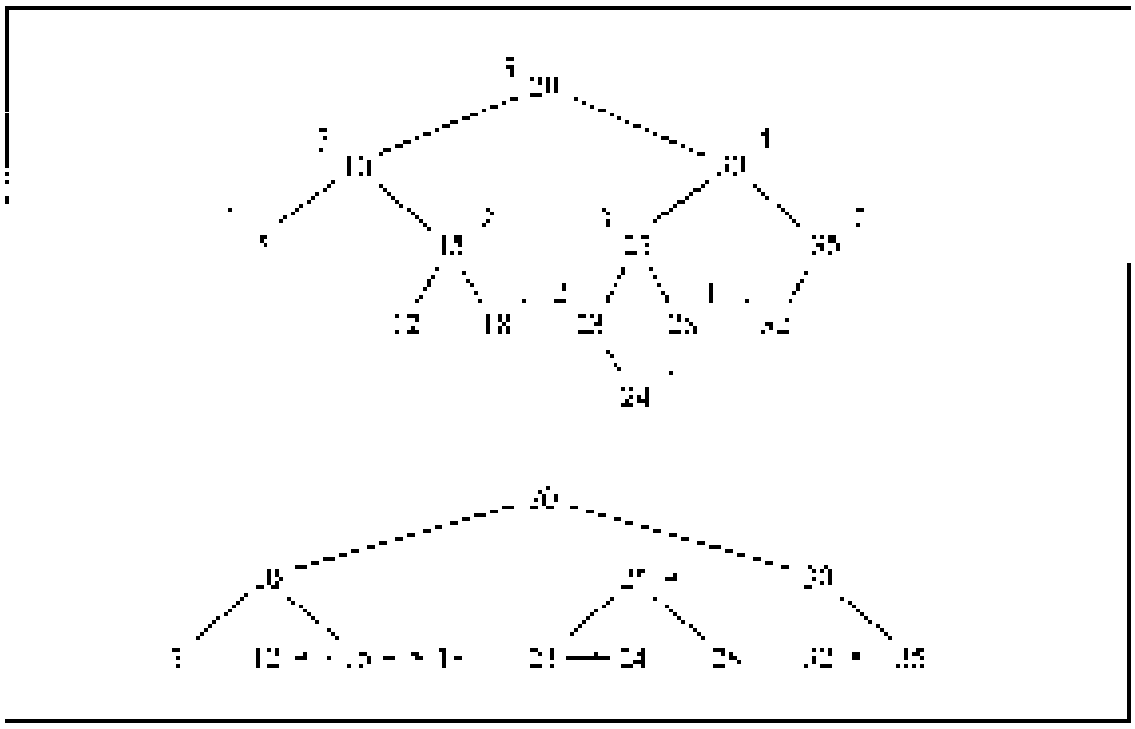


FIGURE 7.29 Member functions of the set class template.

Member Function	Operation
`const_iterator begin() const`	returns an iterator positioned at the first element of the set
`void clear()`	removes all the elements of the set
`size_type count(const ...) const`	returns the number of elements in the set (0 or 1)
`bool empty() const`	returns true if the set includes no elements and false otherwise
`const_iterator end() const`	returns an iterator positioned just past the last element of the set
`pair<const_iterator, const_iterator> equal_range(...) const`	returns a pair of iterators [lower_bound(k), upper_bound(k)) delimiting all the elements with key k
`const_iterator find(...) const`	returns an iterator positioned at the element with key k

Figure 7.29 (continued)

`local_iterator(void* arbitrary, bool);`	constructs a local iterator that may initially point to the specified bucket
`size_type erase(const key_type& k);`	removes the element(s) equivalent to k and returns the number removed
`const_local_iterator cbegin(size_type n) const;`	returns a const local iterator pointing to the first element in bucket n
`pair<const_iterator, const_iterator> equal_range(const key_type& k) const;`	returns a pair of const iterators i and j such that the keys in the range [i, j) are all those x that are equivalent to k, and no others
`const_local_iterator cend(size_type n) const;`	returns a const local iterator pointing to the end of bucket n
`void insert(initializer_list<value_type>);`	inserts elements from initializer_list il
`key_compare key_comp() const;`	returns the function object that compares keys
`const_iterator lower_bound(const key_type& k) const;`	returns a const iterator pointing to the first element with key not less than k
`size_type max_bucket_count() const;`	returns a number n such that n is greater than or equal to the number of buckets the container can ever have
`const_reverse_iterator crbegin() const;`	returns a const reverse iterator pointing to the last element in the container
`const_reverse_iterator crend() const;`	returns a const reverse iterator pointing to the end of the reversed container
`mutex* mp = new mutex();`	allocates a new mutex object, initializes it, and stores the pointer in mp
`mp->lock();`	locks the mutex
`bool b = mp->try_lock();`	tries to lock the mutex, returns immediately on failure
`mp->unlock();`	releases the lock held on the mutex
`const_iterator upper_bound(const key_type& k) const;`	returns a const iterator pointing to the first element with key greater than k
`value_compare value_comp() const;`	returns the function object that compares values

Figure 7.30 An example application of the set_union member function

```
#include <iostream>
#include <set>

using namespace std;

template<class St>
void Union(const St& st1, const St& st2, St& st3) {
     St tmp(st1);
     if (&st1 != &st2)
        for (St::iterator i = st2.begin(); i != st2.end(); i++)
          tmp.insert(*i);
     tmp.swap(st3);
}

int main() {
    ostream_iterator<int> out(cout," ");
    int a[] = {1,2,3,4,5};
    set<int> s1;
    set<int,greater<int> > s2;
    s1.insert(6); s1.insert(7); s1.insert(8);     // s1 = {6 7 8}
    s2.insert(6); s2.insert(7); s2.insert(8);     // s2 = {8 7 6}
    set<int> s3(a,a+5);                           // s3 = {1 2 3 4 5}
    set<int> s4(s3);                              // s4 = {1 2 3 4 5}
    pair<set<int>::iterator,bool> pr;
    pr = s1.insert(7);          // s1 = {6 7 8},   pr = {7 false}
    pr = s1.insert(9);          // s1 = {6 7 8 9}, pr = {9 true}
    set<int>::iterator i1 = s1.begin(), i2 = s1.begin();
    bool b1 = s1.key_comp()(*i1,*i1);             // b1 = false
    bool b2 = s1.key_comp()(*i1,*++i2);           // b2 = true
    bool b3 = s2.key_comp()(*i1,*i1);             // b3 = false
    bool b4 = s2.key_comp()(*i1,*i2);             // b4 = false
    s1.insert(s2.s1.insert(2)); s1.insert(1);
    Union(s1,s3); // s1 = {1 2 3 4 5 6 7 8 9} and s3 = {1 2 3 4 5} =>
                  // s4 = {1 2 3 4 5 6 7 8 9}

    multiset<int> ms1;
    multiset<int,greater<int> > ms2;
    ms1.insert(6); ms1.insert(7); ms1.insert(8); // ms1 = {6 7 8}
    ms2.insert(6); ms2.insert(7); ms2.insert(8); // ms2 = {8 7 6}
    multiset<int> ms3(a,a+5);                    // ms3 = {1 2 3 4 5}
```

Figure 7.30 *(continued)*

```
    multiset<Object>::iterator it = multiset.begin();
    multiset<Object>::iterator upper = multiset.upper_bound(it);
    multiset<Object>::iterator lower = multiset.lower_bound(it);
    multiset.erase(it);
    multiset<Object>::iterator it = multiset.begin();
    multiset.erase(it, upper);
    multiset<Object>::iterator it = multiset.begin(), it = multiset.begin();
    int i, j, k;
    bool b = multiset.empty();
}
```

A new multiset is created in the second line, already full. The third line attempts to insert a new object into the multiset at the location specified by the iterator. This is unsuccessful, which can be checked by comparing the iterator returned by the function insert():

```
pr = obj.insert(7);
```

where the iterator pr points to the 7. The components of the pair are accessed with the attributes first and second.

A more interesting situation arises when we wish to store objects whose order is determined by the value of some of their data members. Consider the class Person, declared in the file person.h:

```
class Person {
public:
    . . . . . .
    bool operator<(const Person& p) const {
        return name.compare(p.name) < 0;
    }
private:
    char name;
    int age;
    friend class hashtable;
};
```

With this definition we declare an array of objects of type Person:

```
Person p[] ={Person("Cheryl",25),Person("Ann",22),Person("Bill",22),
             Person("George",30),Person("Ray",20)};
```

We can declare a hash table and then initialize the table

```
set<Person> pSet1(...);
set<Person,LessSeniorAge> pSet2(...);
```

By default, the first set is ordered with the operator< overloaded for class Person, and the second is ordered with the relation defined by the function object.

```
class LessSenior {
public:
    int operator()(const Person& p1, const Person& p2) const {
        return p1.age < p2.age;
    }
};
```

to find

```
pSet1    {"Ann,'30'} {"Bill,'20'} {"Greg,'20'} {"Ray,'50'}
pSet2    {"Bill,'20'} {"Greg,'20'} {"Ann,'30'} {"Greg,'50'}
```

The first set pSet1, ordered on name and then age, pSet2 by age; therefore in pSet1, each name appears only once so that the object "Greg,'20'" is not inserted, whereas in pSet2, each age is unique so that object "Ray,'50'" is not inserted. An interesting point to mention is that in order to accomplish that, it is sufficient to form a set over which we have defined operator< that is used to order the elements of the set. This is possible as a consequence of implementation of multiset. As already mentioned, a set is implemented as a red-black tree. An iterator in such a tree structure indicates the current node of the tree. Along the way, the inequality test that is performed in order to find a position to insert the element that we want to add to the set (in the member function insert, for example) is equivalent to the expression !(x < y) && !(y < x), where x and y are two elements that already are in the tree in nodes that we have arrived at. If the result is true, we go down to the right. Then [see Josuttis (1999), page 107] we have equivalence if and only if x and y are the same element. Hence, if the order of the elements is now reversed, !(x < y) && !(y < x) is equivalent to (y < x), which means that we should not be allowed to insert the same value twice. Maintaining a pointer to a present node, then by traversing the tree through its parent link, the predecessor in an in-order traversal is referenced by --iter, as mentioned in

How declare and initialize a multiset to index from

```
multiset<Person> pSet1(pSet1.begin(), pSet1.end());
multiset<Person, LessSenior> pSet2(pSet2.begin(), pSet2.end());
```

The new result should be

```
pSet1 = {"Ann,'30'} {"Bill,'20'} {"Greg,'20'} {"Greg,'50'} {"Ray,'50'}
pSet2 = {"Bill,'20'} {"Greg,'20'} {"Greg,'50'} {"Ann,'30'} {"Ray,'50'}
```

With a set, the insertion of a new element is only accepted if the element is not already present in the set.

```
pair<int*,iterator<Comparable,vector<Comparable>>> range =
    tree.equal_range( "Bragg" );
for( int *ptr = range.first; ptr != range.second; ptr++ )
    cout << *ptr << endl;
```

This code produces the output 3 5 (line 3 of the file and the number of particular entries in the file).

```
cout << "Bragg: " << count( "Bragg" ) << " " << count( Person( "", 32 ) );
```

This code outputs 1 and 2. Note that member function count requires an object of the container class type, which is why the first count("Bragg") uses a constructor to build an object with which it calls count(Person("Bragg", 0)). This indicates that the age (the second attribute in the result of the search in a multiset pair) must be explicitly listed in the definition of operator<. Similarly, in case 2, name is irrelevant for the search, so it could be excluded in the request to find objects with a particular age.

It is important to notice that set related algorithms cannot be used on sets. For example, the call

```
set_union(s1, s2, s3, it1, it2);
```

causes a compilation error because "it" is not specified as an object. The reason for the message is that the set_union() algorithm takes non-repetitive elements from the range indicated by iterators s1 and s2 and iterators s3 and it1 and copies them to a set starting from a position indicated by iterator it1. Therefore, the implementation of set_union() includes an assignment

```
*it1++ = *s1++;
```

To ensure the assignment, the iterator it1 must not be constant, but the class set uses only constant iterators. The reason is that the set is implemented as a kind of binary search tree; therefore, a modification of information in a node of the tree could break the order of keys in the tree, which leads to improper output of many member functions of class set. Removed, set related algorithms are inapplicable to sets. To apply them, the set should be implemented differently (e.g., as an array, vector, list, or deque). An alternative solution is suggested in Figure 7.11 in a generic function called copy_unique(), which relies on the member function insert() and swap() and can be applied to objects of any type of class that support these two functions, including objects of type set.

7.1.9 Maps and Multimaps in the Standard Template Library

Maps are tables that can be indexed with any type of data. Hence, they are a generalization of arrays because arrays can be indexed only with constants and variables of ordinal types, such as characters and nonnegative integers, and not with real or double numbers.

Maps and keys form pairs used as indexes and elements to be accessed through the keys. The indexes in arrays (keys in maps) are unique in that any key is associated with exactly one element. In a generalization of this concept, maps can contain pairs that

...and black nodes. But unlike red-black trees, optimized top-down implementations for splay trees are difficult. Instead, we provide a bottom-up implementation that splays nodes by first searching for the node containing the key and then splaying that node to the root of the tree. Unlike most other self-adjusting search trees, splay trees are modified even by operations that would normally be considered queries, such as searches for keys that are not present in the tree. Despite this possibility of modification, we can still declare all of the methods in the splay tree interface to be const, so that they can be called on const splay trees.

FIGURE 7.31 Member functions of the multimap class

Member function	Operation
`iterator begin();`	returns an iterator that references the first element in the map
`const_iterator begin() const;`	returns a constant iterator that references the first element in the map
`void clear();`	removes all the elements in the map
`size_type count(const key_type& key) const;`	returns the number of elements with the specified key
`bool empty() const;`	returns true if the map contains no elements and false otherwise
`iterator end();`	returns an iterator that is past the last element of the map
`const_iterator end() const;`	returns a constant iterator that is past the last element of the map
`pair<iterator,iterator> equal_range(key);`	returns a pair of iterators that bound all keys equal to the specified key; if no keys are equal to the key, both iterators point to the location where the key would be inserted
`pair<const_iterator,const_iterator> equal_range(key) const`	returns a pair of constant iterators that bound all keys equal to the specified key; if no keys are equal to the key, both iterators point to the location where the key would be inserted
`iterator erase(iterator pos);`	removes the element at the iterator pos and returns the element that followed pos
`iterator erase(iterator first, iterator last);`	removes the elements in the range indicated by the iterators first and last
`size_type erase(key);`	removes the element with the key and returns their number
`bool find(key);`	returns true if the map contains an element with the key
`const_iterator find(key) const;`	returns a constant iterator that references the first element with the key

FIGURE 7.31 (continued)

`pair<iterator,bool> insert(const value_type& x);`	inserts pair x into the map and returns a pair consisting of a bidirectional iterator that points to x and a bool that is true if x was inserted and false if an element with the same key as x was already in the map
`iterator insert(iterator position, const value_type& x);`	inserts pair x into the map, where position is a hint indicating where the search for x should start
`void insert(input_iterator first, input_iterator last);`	inserts the range of pairs [first, last) into the map; the iterators must dereference to pairs that can be converted to value_type
`key_compare key_comp() const;`	returns the comparison function object for the map
`iterator lower_bound(const key_type& x);`	returns an iterator pointing to the first pair in the map whose key is not less than x
`const_iterator lower_bound(const key_type& x) const;`	returns a const iterator pointing to the first pair in the map whose key is not less than x
`map() : comp(Compare()) {}`	constructs an empty map using the specified comparison function; uses the default constructor for Compare
`map(const map<Key, T, Compare, Allocator>& x);`	copy constructor
`template <class InputIterator> map(InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator& = Allocator());`	constructs a map containing all the elements in the range [first, last); uses the specified comparison function and allocator
`size_type max_size() const;`	returns the size of the largest possible map
`T& operator[](const key_type& x);`	returns a reference to the object that is associated with the key x; inserts a default object if x is not already in the map
`bool operator==(const map<Key, T, Compare, Allocator>& x) const;`	returns true if the map is equal to x; false otherwise
`const_reverse_iterator rbegin() const;`	returns a const iterator pointing to the last element of the map
`reverse_iterator rbegin();`	returns a reverse iterator pointing to the last element in the map
`const_reverse_iterator rend() const;`	returns a const iterator pointing before the first element of the map
`size_type size() const;`	returns the number of elements in the map
`void swap(map&);`	swaps the contents of the map with the map passed as an argument
`iterator upper_bound(const key_type& x);`	returns an iterator pointing to the upper bound of the range of values with the key x
`const_iterator upper_bound(const key_type& x) const;`	returns a const iterator pointing to the upper bound of the range of values with the key x
`value_compare value_comp() const;`	returns the comparison function for the map

Program 7.34 An example of initialization of the map member variables

```
#include <iostream>
#include <map>

using namespace std;

void main() {
    public:Person *p[]
        {public:Person(char *n, int a, char *h),
         public:Person(char *n, int a, char *h),
         public:Person(char *n, int a, char *h)};
    map<Person,char*> children(p,p+3);
    children[Person("Kay",40)] = "New York";
    children["Jenny"] = "Chicago";
    children.insert(pair<Person,char*>(Person("Ray",42),
                                       "Detroit"));
    children.insert(public:Person(char*n, int a), "Austin");
    map<Person,char*>::iterator i;
    for (i = children.begin(); i != children.end(); i++)
        cout << (*i).first << " " << (*i).second << endl;
    // output
    //    (Ann ,30) Boston
    //    (Bill,20) Baltimore
    //    (Gregg,25) Pittsburgh
    //    (Jane,0) Phoenix
    //    (Kay, 40) New York
    children[Person("Kay",40)] = "Chicago";
    for (i = children.begin(); i != children.end(); i++)
        cout << (*i).first << " " << (*i).second << endl;
    // output
    //    (Ann ,30) Boston
```

An example program is shown in Program 7.34. The map `children` is indexed with objects of type `Person`. The map is initialized with three pairs `Person` objects. strings. The following assignment,

```
children[Person("Kay",40)] = "New York";
```

causes an insertion if the object `Person("Kay",40)` is not defined for the map that contains `children`. Otherwise, the existing value is changed, as in the case of this assignment. The next assignment

Figure 7-32 (continued)

```
//   (Tr 1,20): Delmore
//   (Gregg,25): Dittenhafer
//   (Henry,5): Roscoe
//   (Ray, 4): New York

multimap<Person, greater<Person>> pc(cities.begin(), cities.end());
pc.insert(pair<Person,Person>(Person("Ray",40),Person("Kristi",0)));
pc.insert(pair<Person,Person>(Person("Ray",40),Person("Kristi",0)));
pc.insert(pair<Person,Person>(Person("Ray",40),Person("Detroit",0)));
multimap<Person,greater<Person>>::reverse_iterator pi;
for (pi = pc.rbegin(); pi != pc.rend(); ++pi)
    cout << (*pi).first << " " << (*pi).second << endl;
// outputs:
//   (Ann,10) Canton
//   (Bill,20) Delmore
//   (Gregg, 5) Fitchburg
//   (Ray, 40) Auburn
//   (Ray, 40) Auburn
//   (Ray, 40) Detroit
pc[Person("Ray",40)] = "New York";
for (pi = pc.begin(); pi != pc.end(); ++pi)
    cout << (*pi).first << " " << (*pi).second << endl;
// output:
//   (Ann,10) Canton
//   (Bill,20) Delmore
//   (Gregg,25) Fitchburg
//   (Ray, 40) New York
//   (Ray, 40) Auburn
//   (Ray, 40) Detroit
```

```
multimap["Henry"] = "Roscoe";
```

multimap is a sequence, so you can't use operator[] with it. Person("Henry",0) is the key for the first element in the multimap, and the assignment changes the value of that element to "Roscoe".

Now you can see why the multimap doesn't support operator[]. Because it's legal to have multiple elements with the same key, there's no way to know which one you want to access. Even if you're sure there's only one element with a given key, the multimap doesn't know that, so it can't help you.

insert() statement if one is explicitly created. Both statements are examples of inserting a pair for an already existing key Person("Ray", 63); therefore, no insertion takes place, although the element is by name() is different from the one in the map with which the key is associated. To update an element in the map, an assignment has to be used, as illustrated in the program with the assignment

```
entries[p] = Person("Ray", 63);
```

Because the elements of the array pairs are pairs, a key such as the first element of the pair is accessed using the notation p[0].first. The way of accessing elements of pairs is also illustrated in the printing statements.

The program in Figure 7.21 is shown only to illustrate some concepts in indexing. It would probably be more natural to include a key as another data member in each object. A more useful example concerns social security numbers and objects of type Person. If we wanted to create an array-like vector or sequence of SSNs, it could be created as before, the array would need to be 10 cells because the largest SSN is nine 9s: 999999999. But with maps, we can have only as many entries as the number of Person objects used in the program. For example, we can define a map SSN

```
map<long,Person> SSN;
```

and then execute a number of assignments

```
SSN[122436789] = p[1].first;
SSN[987654321] = p[2].first;
SSN[222222222] = Person("Ray", 63);
SSN[111111111] = "Jones";
```

In this way, SSN has only four entries although keys are very large numbers

```
SSN = {[111111111,"Jones"],[122436789,"Mary"],
       [222222222,"Ray"],[987654321,"Gregg"]}
```

Information is now very easily accessible and modifiable by using SSNs as the access keys.

Multimap is a map natural also for duplicate keys. The associated class has the same member functions as map with a few exceptions. Map constructors are replaced by multimap constructors with the same parameters, and the member function `returnIterator, bool> insert(key_value)` is replaced by `iterator insert(key_value)`, which returns an iterator referencing the position in which the pair `<key,value>` was inserted. The subscript operator [] is not defined for multimaps. To modify an element in a map, the element can be found and then modified using the find() function as shown in Figure 7.22. Data is modified only one entry, the first found by the function. To modify all the entries for a particular key, we can use a loop such as

```
for (int ...  multiMap.lower_bound(Person("Ray", 63));
     it != multiMap.upper_bound(Person("Ray", 63)); ++it)
    (*it).second = "new value";
```

7.2 TRIES

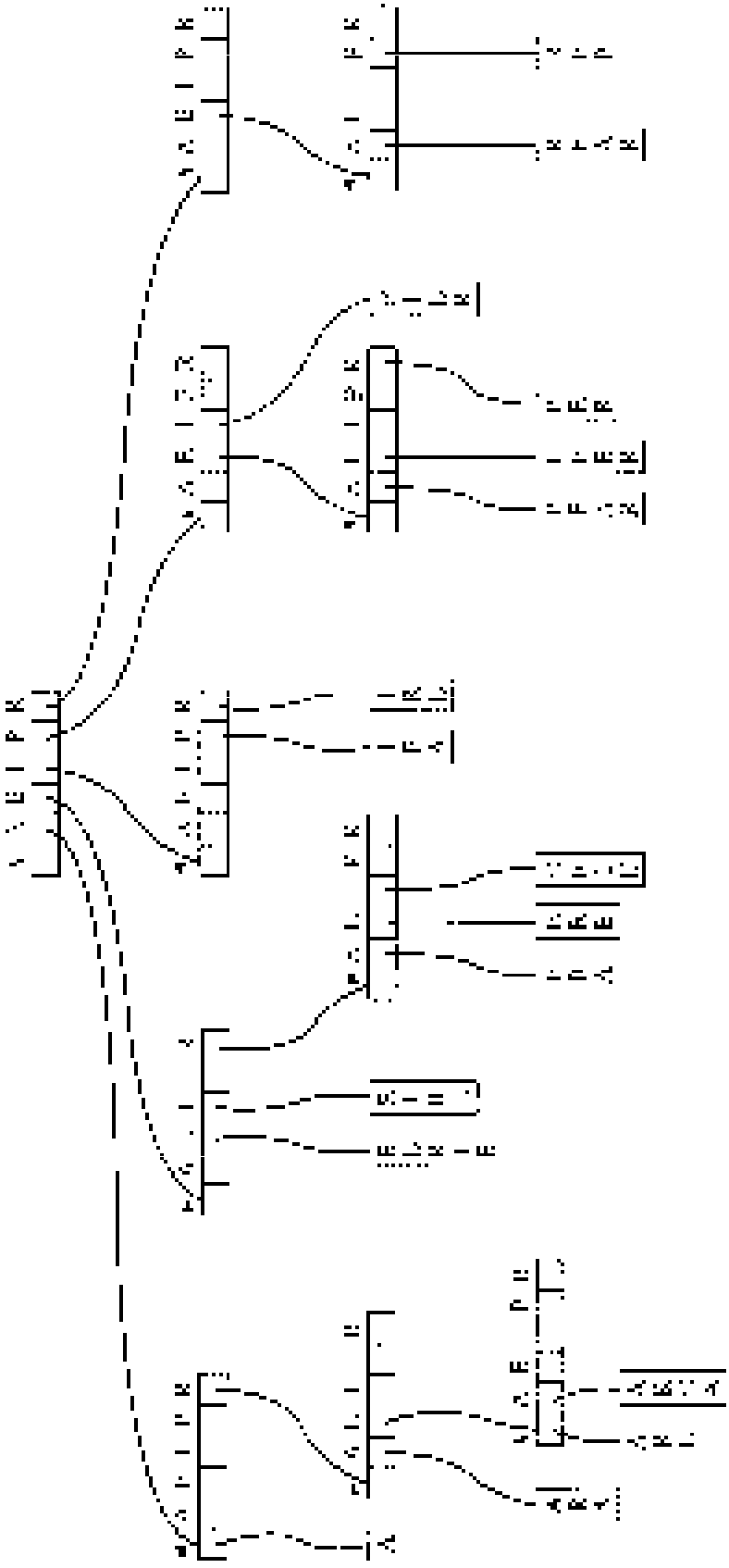
The previous chapter showed how binary search trees can be used to implement an ordered dictionary ADT. In this section, we examine a special tree data structure, called a *trie*, which is commonly used to store dictionaries of character strings over a fixed alphabet. Tries support standard dictionary operations such as insertion, deletion, and searching in time proportional to the length of the string. This is asymptotically faster than the $O(\log n)$ time required by binary search trees or balanced binary search trees, which must perform $O(\log n)$ string comparisons, each of which takes time proportional to the length of the string. Tries also support prefix searches, allowing for efficient retrieval of all strings in the dictionary that contain a given prefix. Tries are one of the few data structures that can efficiently support prefix searches, making them ideal for storing a dictionary of English words for use in a spell checker or thesaurus.

Standard tries store a collection of strings over an alphabet of size d. Each node in a standard trie has up to d children, one for each symbol of the alphabet. Given a node v in a standard trie, the i^{th} child of v stores the pointer to the node reached by following the i^{th} symbol of the alphabet from v. For example, in a standard trie for English words, each node has at most 26 children (one for each letter of the alphabet), and the edge leading to the second child from the root represents all words that begin with the letter 'B'. Similarly, each node in a standard trie for binary strings ($d = 2$) has at most two children (one for '0' and one for '1'). Figure 7.11 shows a standard trie for binary strings.

Figure 7.11 A standard trie for binary strings.

Each node in a standard trie, with the exception of the root, stores a character. The root stores nothing. Each path starting at the root corresponds to a string in the trie. For example, the path from the root to the second child of the root to the first child of that node corresponds to the string '01'. Similarly, the path from the root to the first child of the root to the first child of that node to the second child of that node corresponds to the string '010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node corresponds to the string '0100'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node corresponds to the string '01001'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '010010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '0100101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '01001010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '010010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '0100101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '01001010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '010010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '0100101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '01001010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '010010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '0100101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '01001010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '010010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '0100101010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '01001010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '010010101010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '0100101010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '01001010101010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '010010101010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '0100101010101010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '01001010101010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '010010101010101010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '0100101010101010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node corresponds to the string '01001010101010101010101010101'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node corresponds to the string '010010101010101010101010101010'. Note that the path from the root to the first child of the root to the first child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second child of that node to the first child of that node to the second

FIGURE 7.33 A trie of some words composed of the five letters A, E, I, K, and T. The shaded circles indicate the end of a word which can be a prefix of another word.



This last example illustrates another way in which it is useful to associate values with keys. For words in the trie, what is the length of the longest word starting with "ABC" and what is the position of the first character of that word? Answering these queries requires us to make a comparison between the key "ABC" and the characters stored in the trie. Any value stored in the trie can be associated with the key. Before we begin the discussion of the implementation, we should point out that only the most recent value associated with a key is stored in the trie. Thus if we associate the value 5 with the key "ABC" and then later associate the value 17 with the key "ABC", the value 17 will overwrite the value 5.

The amount of memory required by the trie data structure is proportional to the number of characters in the words stored in the trie. Each node in the trie requires 28 words of memory: one for each of the 26 letters of the alphabet and two pointers to help navigate the tree. Each word requires memory proportional to its length. Thus the memory requirements for a trie can be quite large. For example, storing the 10,000 words in the file `/usr/dict/words` requires more than 1 megabyte of memory on a 32-bit machine. Storing the 50,000 most frequently occurring words in the Brown corpus requires more than 5 megabytes of memory. Tries are best suited for storing relatively short words.

The height of the trie is determined by the length of the longest word stored in the trie. For example, all letters used in the words in the file `/usr/dict/words` are either lowercase or uppercase letters, so the height of the trie is 52. The height of the trie is determined by the longest prefix of any word in the trie. For example, if the trie contains only the words "a" and "antidisestablishmentarianism", then the height of the trie is 28, the length of "antidisestablishmentarianism". Thus tries are best suited for storing words with relatively short prefixes.

Tries are also known as *radix trees* because the branching at each node is based on the value of a character in the key. Thus the branching factor at each node is the radix of the alphabet. For example, if we are storing words consisting of the letters A, C, G, and T, then the radix is 4. If we are storing words consisting of the letters A through Z, then the radix is 26. If we are storing words consisting of the letters A through Z and the digits 0 through 9, then the radix is 36. The larger the radix, the more memory is required by each node in the trie. Thus, in situations where the speed of access is vital, a trie is a good choice for a relatively small alphabet.

Tries are also known as *digital search trees* because the branching at each node is based on the value of a digit of the key. Thus the branching factor at each node is the number of digits in the alphabet. For example, if we are storing words consisting of the letters A, C, G, and T, then the branching factor is 4. If we are storing words consisting of the letters A through Z, then the branching factor is 26. If we are storing words consisting of the letters A through Z and the digits 0 through 9, then the branching factor is 36. The larger the branching factor, the more memory is required by each node in the trie. Thus, in situations where the speed of access is vital, a trie is a good choice for a relatively small alphabet.

Tries are also known as *digital search trees* because the branching at each node is based on the value of a digit of the key. Thus the branching factor at each node is the number of digits in the alphabet. For example, if we are storing words consisting of the letters A, C, G, and T, then the branching factor is 4. If we are storing words consisting of the letters A through Z, then the branching factor is 26. If we are storing words consisting of the letters A through Z and the digits 0 through 9, then the branching factor is 36. The larger the branching factor, the more memory is required by each node in the trie. Thus, in situations where the speed of access is vital, a trie is a good choice for a relatively small alphabet.

Tries are also known as *digital search trees* because the branching at each node is based on the value of a digit of the key. Thus the branching factor at each node is the number of digits in the alphabet. For example, if we are storing words consisting of the letters A, C, G, and T, then the branching factor is 4. If we are storing words consisting of the letters A through Z, then the branching factor is 26. If we are storing words consisting of the letters A through Z and the digits 0 through 9, then the branching factor is 36. The larger the branching factor, the more memory is required by each node in the trie. Thus, in situations where the speed of access is vital, a trie is a good choice for a relatively small alphabet.

```
t***eInsert(R
   * = 0;
   p = the root;
   while *p!=nullptr
      * = R[i] == '\0' ?
         set the end-of-word number in p to true
      e** = if (p->next[x[i]] == 0)
         create a node containing R and put its address in p->next[R[i]];
      e** = if pointer p->next['X'-'A'] points to a node
         R_ = word in trie p->next['X'-'A']
         do {swap number and pointer addresses in p->next[R'-'A'];
            p = the new current node;
         while (R'-'A'      R_[i]-'A');
         create a node containing R and put its address in p->next[R[i] - 1]];
         if (R_[i]'       '\0')
            set the end-of-word number to true;
         else create a leaf containing R_ and put its address in p->next[R_[i]];
      else p = p->next[R[i] - 1];
```

The inorder traversal of this 2-3 tree visits the keys in the order A B C D E F G H I J K L M N O P Q R S T U V W X Y Z. Note that the keys in the internal nodes are used to direct the search, but they are not necessarily deleted when the search is complete. For example, the key 'E' is visited on the way to finding 'F' in Figure 7.11. The word 'SEARCH' is stored in the leaf node corresponding to the interval of the word 'SEARCH' in the sorted list. The word 'SEARCH' is not stored in the tree, although the key 'E' is. Because the search for 'SEARCH' ends at the third level of the tree, it takes four key comparisons to find the words 'SEARCH' and 'REPLACEMENT' in this tree.

If we compare this search tree to the binary search trees we saw earlier, the order in which keys are inserted is very important. Since this tree grows from the leaves toward the root, there is no way to force a key to the root except by the separator keys that are created when the root splits. Therefore, the height of the tree is equal to the length of the longest prefix of any of the words stored in the tree. Because the words 'SEARCH' and 'REPLACEMENT' are the longest words in the tree, the height of the tree is 8. Figure 7.11 has height five since the longest word is 'FUZZINESS'. It is not clear how to insert the words in order to build a perfect 2-3 tree.

The main problem with the tree in Figure 7.11 is that it wastes a substantial amount of disk space. Many nodes are only half full, and some nodes contain only a single pointer. There are only 33 pointers and 26 keys in the tree. There is a 1-to-1 correspondence between the number of keys and the number of unused pointers.

One way to reduce the wasted space is to apply a compression technique to the nodes of the tree, as in Figure 7.6 (Bayer 1972; Bayer and McCreight 1972). However, this would require a substantial amount of bookkeeping and would greatly complicate the implementation. An alternative is to use the unused pointers to store additional keys from the sorted list. Figure 7.12 shows the same tree as in Figure 7.11, but with the unused pointers and keys removed. The tree now has only 13 nodes instead of 26. Each node is now a variable-length record, which makes memory management more difficult. The space savings may be worthwhile if the tree is very large and will be stored for a long time. The savings will be even greater for trees of higher order.

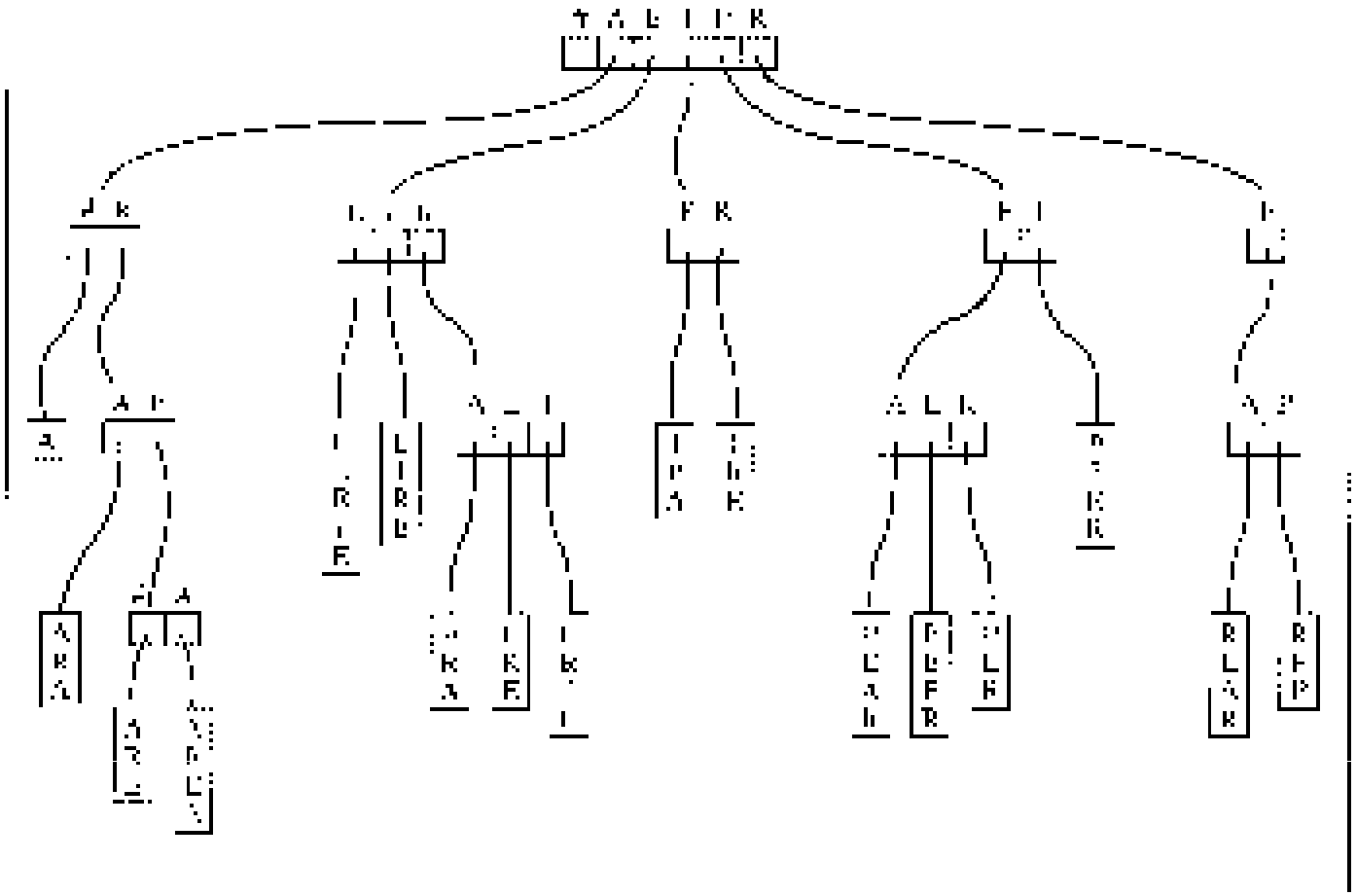
Another way to reduce the space requirements is to use a variable-length key compression technique such as front compression (Bayer and Unterauer 1977). In a front-compressed node, only enough of the key to distinguish it from its neighbors is stored. For example, the first word in the second leaf node of Figure 7.12 is 'FOR'. Only the first two letters are necessary to distinguish it from 'FIND', the last word in the first leaf node. The third letter of 'FOR' is necessary because 'FOOT' is the next word in the tree. Front compression is discussed briefly in Section 7.7.

Another way to reduce the space requirements is to use a data structure called a *trie* (Fredkin 1960; Knuth 1973b; Aho, Hopcroft, and Ullman 1974). The word 'trie' comes from the middle letters of the word 'retrieval', although it is pronounced like 'try' to avoid confusion with the more general use of the word 'tree'. Trie algorithms were first discussed in the context of computer typesetting by de la Briandais (1959). Unlike a B-tree, which stores entire keys in its nodes, a trie breaks the key into pieces and stores individual pieces in its nodes. Each node in a trie of order m contains m pointers and a flag to indicate whether the node is a leaf. Leaf nodes contain the keys, and internal nodes contain the pointers. For example, the first letter of a key is stored in the root of the trie. The ith pointer in the root node points to a subtrie containing all keys beginning with the ith letter of the alphabet. Subtries are defined recursively in the obvious way. The second level of the trie stores the second letter of each key, and so on. Figure 7.13 shows a trie containing the words 'A', 'AN', 'ANT', 'ALL', 'ALLOT', 'ALLOY', 'ALLOYS', 'ALLY', 'BE', 'BEAR', 'BEARD', 'BEARS', 'BEAT', 'BEATEN', 'BEATER', 'BEATS', 'I', 'IS', 'ISLAND', 'ISLANDS', 'IT', 'ITS', 'ITSELF', 'SAND', 'SANDS', 'SET', 'SETS', 'SETTER', 'SETTERS', 'SETTING', 'SETTINGS', 'THE', 'THEIR', 'THEM', 'THEME', 'THEMES', 'THEN', 'THERE', 'THESE', 'THEY', 'THIS', 'TO', 'TOO', 'TOOL', 'TOOLS'. Leaf nodes are shown as boxes containing the keys. Internal nodes are shown as boxes containing the pointers. Empty pointers are shown as dots. Note that the trie is not balanced, and that the height of the trie is equal to the length of the longest word in the trie.

A trie has the property that all of the descendants of a node have a common prefix of the string associated with that node, and the root node represents the empty string. Thus, the first branching in the trie in Figure 7.13 is 3-way, corresponding to the first letters of the words that are not prefixes of other words in the trie ('I', 'S', and 'T'). The next branching is 2-way, corresponding to the second letters of the words beginning with 'A' ('L' and 'N'). The next branching is 3-way, corresponding to the third letters of the words beginning with 'AL' ('L', 'LO', and 'LY'). The next branching is 2-way, corresponding to the fourth letters of the words beginning with 'ALLO' ('T' and 'Y'). Finally, the next branching is 2-way, corresponding to the fifth letters of the words beginning with 'ALLOY' ('S' and the blank character). Figure 7.14 shows the trie of Figure 7.13 with the unused pointers and keys removed. Figure 7.15 shows the trie of Figure 7.14 with the pointers implemented as offsets relative to the beginning of the array. Figure 7.16 shows the trie of Figure 7.15 with the pointers implemented as relative offsets and the keys stored in a separate array. Figure 7.17 shows the trie of Figure 7.16 with the keys stored in the array in sorted order. Figure 7.18 shows the trie of Figure 7.17 with the keys stored in the array in reverse order.

Tries have the property that no balancing is required as keys are inserted or deleted. However, some of the nodes may contain only a single pointer, leading to inefficient use of space. Trie algorithms are discussed in Section 7.7.

Figure 7.34 The trie in Figure 7.33 with all unused pointers to 0 removed



array occupies substantially less space than all nonnull nodes combined. In this example, all nonleaf nodes require $3 \times 4 = 12$ cells, and the cellArray has 11 cells, so the compression ratio is $11/12 = 91.6\%$. However, if the data are stored as in Figure 7.10c, the compression ratio is $11/9 = 122.2\%$.

It turns out that the algorithm which compresses the trie is exponential in the number of nodes and inapplicable for large tries. Other algorithms may not render the optimal compression ratio but are faster. Al-Suwaiyel and Horowitz [1983] give one such algorithm in `compressTrie1`.

```
compressTrie1
   allocate cellArray[0..numberOfNodes-1]
   for each node
      for each position i of cellArray
         if node.pointer[i] = cellArray[j] for some cellArray[j] then
            set cellArray[node.pointer[i]] to point to node with pointer
            node.pointer[i] changed to point to cellArray[j]
         if node is the root then set root to point to cellArray[j]
         break;
```

FIGURE 7.35 The tree from Figure 7.34 reconstructed as a binary tree.

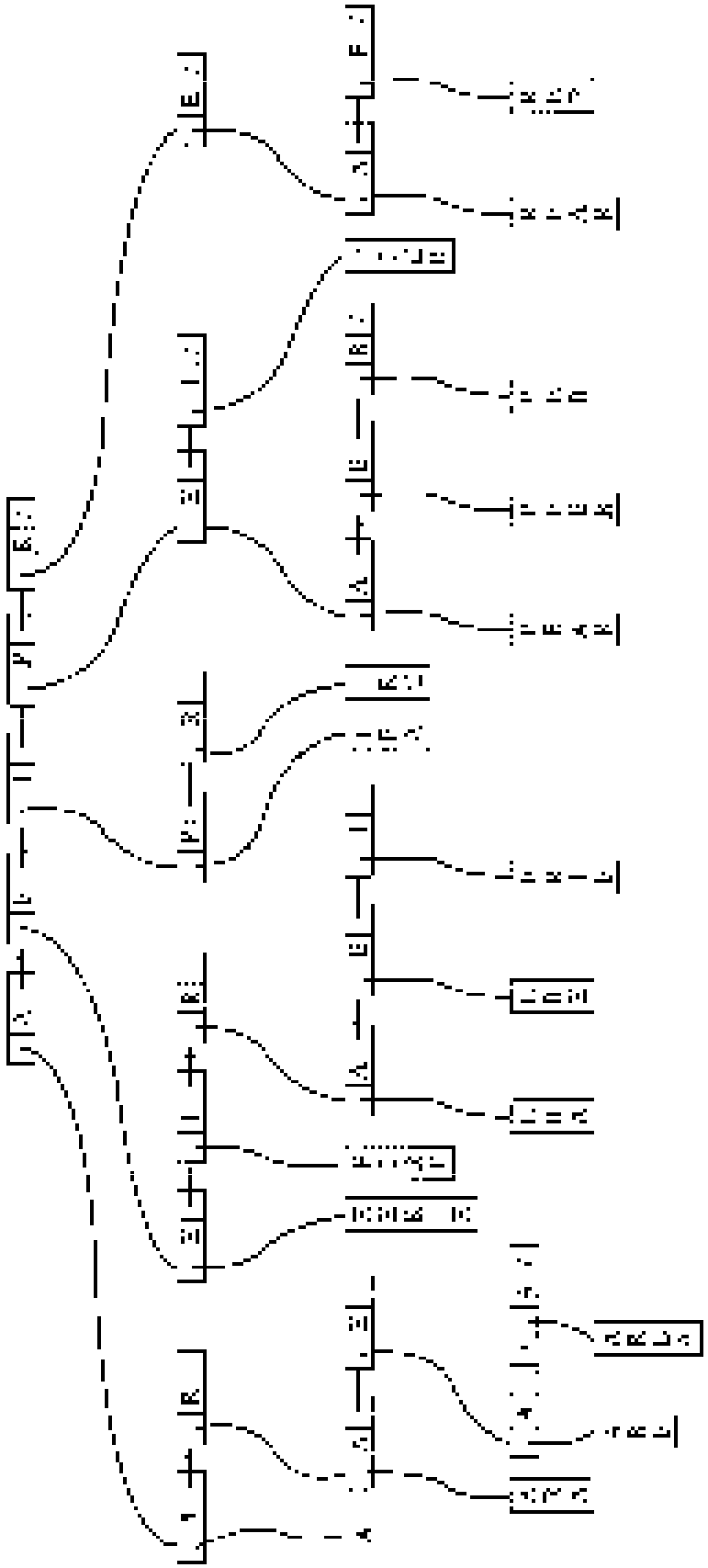
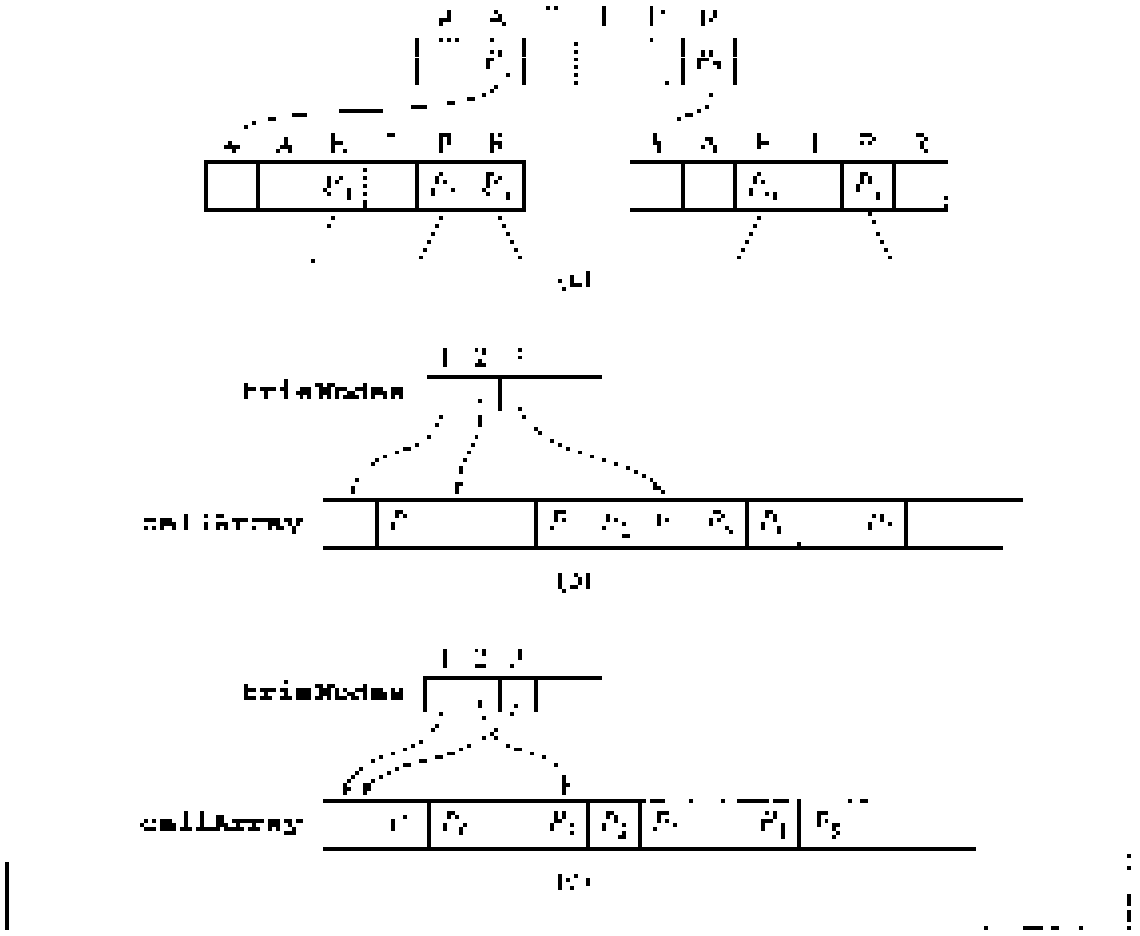


FIGURE 7.36 A possible trie (a) before and (b) after compression using the compressed trie algorithm and (c) after compressing it in an optimal way.

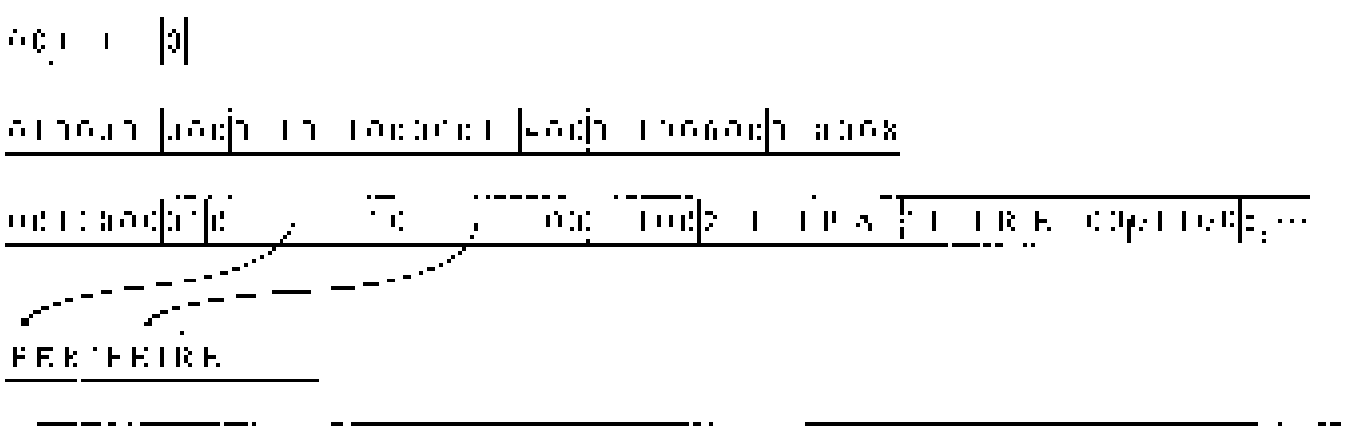


This is the algorithm that was applied to the trie in Figure 7.35 to create the one in Figure 7.36(b). Searching in such a compressed trie is similar to searching in a regular trie. However, node accesses are modified to take the `skip` field into account. The query rules to follow the path-node pointers have to be modified in the same way, and can be used in the `cellArray`.

The problem with using a compressed trie is that the search key is stored, for instance, a search for a word starting with the letter P is immediately directed to the trie in Figure 7.36(a) since the `skip` field corresponding to this letter points to a node (node 2 in the figure). On the other hand, in the compressed version of the same trie (Figure 7.36(b)), this field corresponding to P is empty (0), so that the algorithm will be directed only after some unsuccessful pointer chasing, after reaching a leaf, by comparing the key in this leaf with the key used in searching.

One possible way to compress tries is by creating a Patricia trie (a compressed version of the original trie (Morly 1968)). In Patricia, chains of one-level tries are removed in order to save space and memory, and by the use of the `skip` and `count`

Figure 7.37 A fragment of the C data representation of the tree shown in Figure 7.36.



leaf and a pointer to the next leaf in sequence. Internal nodes contain particular values that indicate the path of descent through the tree. Each internal node contains a number of pointers to other nodes and a corresponding number of keys that indicate which path should be taken to reach a particular value. For example, in a B+ tree of order 5, every node (except the root and the leaves) has at most 5 children (pointers to other nodes) and at least 3 children; the root has at least 2 children (unless it is a leaf); and a leaf has at most 4 values. In order to maintain these constraints, nodes may occasionally have to be split or combined as new values are added or removed. Splitting and combining nodes will be discussed shortly.

Each node contains a field named `count` that gives the number of values (or pointers) currently stored in the node. For example, the B+ tree in Figure 7.36 has a root node with two values (and three pointers) and a left child node with three values (and four pointers). The right child node has only two values (and three pointers), because it contains the minimum number of values allowed for a node that is not a leaf or the root. Nodes are not allowed to contain fewer than the minimum number of values (except for the root) because then they would be more than 50% empty. Maintaining a minimum level of node occupancy guarantees that the tree will never become unbalanced to the point where performance degrades substantially.

Each node also contains an array of keys named `key` and an array of pointers named `ptr`. The number of keys is always one less than the number of pointers. The relationship between keys and pointers is as follows: `ptr[0]` points to a node containing values less than `key[0]`, `ptr[1]` points to a node containing values greater than or equal to `key[0]` and less than `key[1]`, and so on. Finally, `ptr[count-1]` points to a node containing values greater than or equal to `key[count-2]`. Figure 7.38 illustrates this relationship.

Each node also contains a Boolean field named `leaf` that indicates whether the node is a leaf. Leaf nodes do not have pointers to other nodes, so their pointers are used to point to the next leaf in sequence. For example, the pointer `ptr[4]` in the left child node of the root in Figure 7.36 points to the right child node. The pointer `ptr[4]` in the right child node is `NULL`, indicating that it is the last leaf in the tree.

Searching for a key in a B+ tree is similar to searching for a key in a binary search tree. Here is an outline of the algorithm:

```
Search for key k:
   Set p = pointer to root node; found = false;
   while p is not NULL
      if p is a leaf
         if k is a value in p
            found = true
         else failure;
      else if k == key[i] for some i
         if value[i] is not NULL
            success;
         else failure;
      else if k is between key[i-1] and key[i] for some i
         failure;
      else p = pointer to the next node to check;
```

```
-C fanout(p) size(root node)  // to skip all children of nodes
                              // in front of p on level 1;
+{pointer(i) fanout(p) size(p) p.child(p) | i in [1,p.size]}
  corresponding to K[1] | p in disappearnode.  // and children of p;
```

For example, to find "KARL" in the example B-tree of Figure 7.17, we first look for the root of the first entry. Since the first entry is not the root, a half-page read is performed. Only the root node is then accessed to find the root node of the second entry. Since the address of the root node is the same as that of the first entry, the last page read is skipped. The root node is searched to find the pointer to the second entry. The address of the root node is the same as that of the first entry, so

7.4 Case Study: Spell Checkers

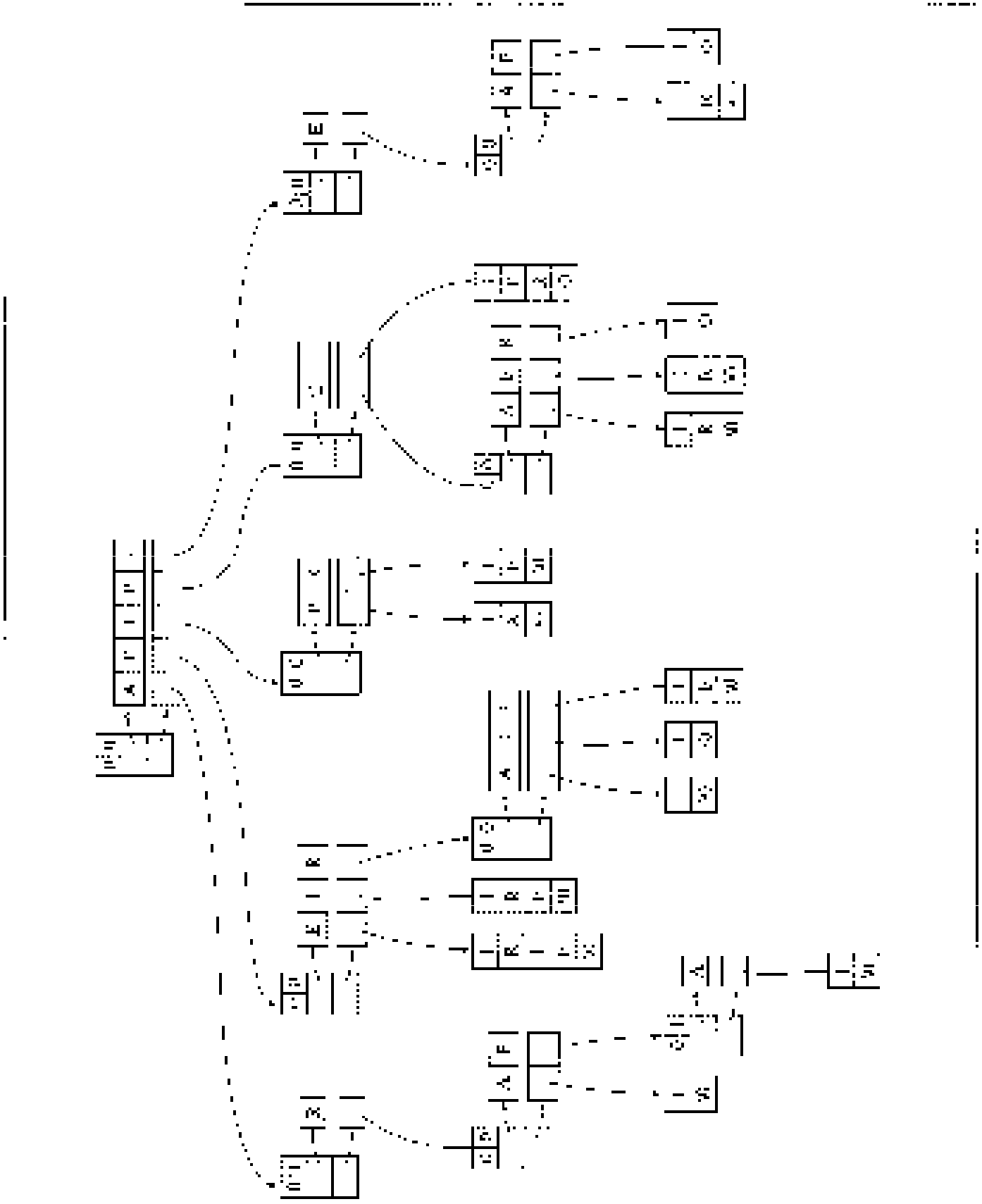
An objective of [illegible] word processor is to spell check the words in a document. The user is told about misspelled words and possibly corrections. Depending on the program, the spell checker can be run in batch mode on a complete document or interactively as words are typed. Batch spell checkers usually make a listing of misspelled words and possibly their corresponding line numbers in the document. Interactive word processing programs highlight misspelled words as they are typed, allowing the user immediate correction.

The core of a spell checker is a data structure containing all the words in the dictionary. Such a dictionary might likely have [illegible] words, and it needs to be very fast to process queries. Typically a query is a word and the response is a simple yes/no indicating whether or not the word is in the dictionary. Sometimes a list of words with similar spellings is also returned. The user can then select the desired word from the list, and replacement takes place.

Typical [illegible] of spelling errors include typing two letters in the wrong order, omitting a letter, typing an extra letter, and typing the wrong letter. Studies have been done on [illegible] with [illegible] (see Exercise 7.11). Most misspelled words are within two errors of the correct spelling. That is, the correct word can be obtained from the misspelled word by two insertions, deletions, or transpositions of letters. Sometimes two errors result in another valid word (for example, "their" and "there"), but usually more than two errors are required. Thus, a useful heuristic when searching for words with similar spellings is to look at words that are within two errors of the misspelled word. Searching for words within two errors can be a time-consuming process for a large dictionary, so careful thought must be given to the design of the data structure and algorithm.

One data structure that can be used for the dictionary is a trie. Figure 7.16 shows a trie containing the words "a," "all," "ape," "can," "cap," "car," "cat," "data," "date," "dot," "mat," and "may." Each node contains 27 pointers, one for each of the letters in the alphabet and one for the end of a word. Pointers that are not used are shown as dashes. To determine if a word is in the trie, we start at the root and follow the pointer for the first letter of the word. We then follow the pointer for the second letter of the word from the node we arrived at by following the first letter. We continue in this manner until we either fall off the trie (because a pointer we want to follow is NULL) or we have used the last letter of the word. In the latter case, we must check whether the node we are at corresponds to a complete word. This is done by setting a flag in the node that indicates whether or not it corresponds to a word. For example, the node we reach by following the letters "c" and "a" in the trie in Figure 7.16 has a flag value of true, indicating that "ca" is a word. The node we reach by following the letters "c," "a," and "r" has a flag value of true, indicating that "car" is a word. The node we reach by following the letters "c," "a," and "t" has a flag value of true, indicating that "cat" is a word. The node we reach by following the letters "c," "a," and "p" has a flag value of true, indicating that "cap" is a word. The node we reach by following the letters "c," "a," and "n" has a flag value of false, indicating that "can" is not a word. The node we reach by following the letters "c," "a," and "d" has a flag value of false, indicating that "cad" is not a word. The node we reach by following the letters "c," "a," and "b" has a flag value of false, indicating that "cab" is not a word. The node we reach by following the letters "c," "a," and "x" has a flag value of false, indicating that "cax" is not a word. The node we reach by following the letters "c," "a," and "z" has a flag value of false, indicating that "caz" is not a word. The node we reach by following the letters "c," "a," and "q" has a flag value of false, indicating that "caq" is not a word. The node we reach by following the letters "c," "a," and "w" has a flag value of false, indicating that "caw" is not a word. The node we reach by following the letters "c," "a," and "e" has a flag value of false, indicating that "cae" is not a word. The node we reach by following the letters "c," "a," and "r" has a flag value of true, indicating that "car" is a word. The node we reach by following the letters "c," "a," and "s" has a flag value of false, indicating that "cas" is not a word. The node we reach by following the letters "c," "a," and "t" has a flag value of true, indicating that "cat" is a word. The node we reach by following the letters "c," "a," and "u" has a flag value of false, indicating that "cau" is not a word. The node we reach by following the letters "c," "a," and "v" has a flag value of false, indicating that "cav" is not a word. The node we reach by following the letters "c," "a," and "w" has a flag value of false, indicating that "caw" is not a word. The node we reach by following the letters "c," "a," and "x" has a flag value of false, indicating that "cax" is not a word. The node we reach by following the letters "c," "a," and "y" has a flag value of false, indicating that "cay" is not a word. The node we reach by following the letters "c," "a," and "z" has a flag value of false, indicating that "caz" is not a word.

Figure 7.38 An implementation of a trie that uses pseudodeletible arrays. The trie has the same word as the trie in Figure 7.33.



Also, the discussion of tries in this chapter assumed that the keys of the tries were full keys. This is not necessary because the suffixes of all words are implicitly stored in the trie and can be reconstructed by going up all the letters on the path leading to the leaf. For example, to access the leaf with the word "a.bit," two pointers have to be passed through pointers corresponding to the letters "t," and "t." Here, the trie's weight becomes the suffix "ff" in the leaf instead of the entire word "a.bit." Typically, the number of suffixes of these words is very small, and these leaves are one of the most compressed nodes of the trie. Figure 7.11 shows a more compressed version.

We also included the function `visitEachKey`, which prints the contents of a trie sideways. The output generated by this function when applied to the trie in Figure 7.10 is as follows:

```
        >>>ear
        >>>eBA  |
    >>>t.|sh
        >>>e'|
        >>>ee|t
        >>>eA|'
    >>>th|e
    >>>i |A
        >>>sh|s
        >>>sh|
        >>>sh|
    >>>st|ck
    >>>st|ll|e
            >>>ll|s
           >>>ll|
          >>>ll|
    >>>s
```

Three apple trackers in the words `a`, `all`, and `an` are marked with a `*` to indicate that they have been set in the corresponding nodes. For example, the word `an` has apple trackers `a` and `n` in the trie. Some nodes have two characters in their `*` marks. The vertical bar `|` separates the prefix reconstructed when traversing the trie from the suffix that will be reconstructed from a leaf.

Spell checking works much as before. We first break the string `each` word into letters and print up to all misspelled words along with their line numbers where the misspelled words are found. Figure 7.12 contains the complete code of the spell checker.

Figure 7-39 Implementation of a spell checker using tries

```
//*************************  Trie.h  ***********************************

class Trie;

class TrieNonLeafNode {
friend ostream& operator<<(ostream&, const Trie&);
friend Trie;
private:
    TrieNonLeafNode(int, const Trie&);
    TrieNonLeafNode(const TrieNonLeafNode&);
    void operator=(const TrieNonLeafNode&);
    Trie letters;
};

class TrieLeafNode {
public:
    TrieLeafNode() {
    }
    TrieLeafNode(char);
private:
    bool leaf;
    char value;
    friend Trie;
};

class Trie {
public:
    Trie(int = initialSize) {
    }
    Trie(char);
    void display() {
        cout << "print";
        display(cout, 0);
    }
    void insert(char);
    bool member(char) const;
private:
    TrieNonLeafNode *trie;
```

Listing 7.39 (continued)

```
    const int notFound;
    char prefix[80];
    int  position(TrieNonLeafNode*,char);
    void addChild(char,TrieNonLeafNode*,int);
    void splitLeaf(char,char*,TrieNonLeafNode*);
    void splitFix(int,TrieNonLeafNode*,char*);
};

//************************  Trie.cpp  ************************

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "trie.h"

TrieLeafNode::TrieLeafNode(char *suffix) {
    leaf = true;
    word = new char [strlen(suffix)+1];
    if (word == 0) {
        cout << "Out of memory!\n";
        exit(-1);
    }
    strcpy(word,suffix);
}

Trie::TrieLeafNode::Trie(char c) {
    root = new TrieNonLeafNode;
    int* data = new int[1];
    if (root == 0 || data == 0) {
        cout << "Out of memory!\n";
        exit(-1);
    }
    leaf = false;
    root->full = false;
    root->ptrs = 0;
    root->letters = c;
    root->prefix[0] = '\0';
}

Trie::Trie(Trie& word) : notFound(-1) {
```

Figure 7.39 (continued)

```
    root = new TrieNode(L'\0'); // initialize the root
    create(root,word,0,end);    // recursively insert words
}

void Trie::suggest(int depth, TrieNode *p, char *prefix) {
    register int i;             // each node is either a leaf
    if (p->leaf) {              // node or is internal
        TrieLeafNode *lf = (TrieLeafNode*) p;
        for (i = 1; i <= depth; i++)
            cout << '  ';
        cout << lf->word << " " << lf->count << endl;
    }
    else {
        for (i = strlen(p->letters)-1; i >= 0; --i)
            if (p->ptrs[i] != 0) {          // add the letter
                prefix[depth] = p->letters[i]; // corresponding to
                prefix[depth+1] = '\0';     // position i to prefix
                suggest(prefix,depth+1,p->ptrs[i]);
            }
        if (p->endOfWord) {
            prefix[depth] = '\0';
            for (i = 1; i <= depth+1; i++)
                cout << '  ';
            cout << "DOG" << prefix << '\n';
        }
    }
}

int Trie::position(TrieNonLeafNode *p, char ch) {
    for (int i = 0; i < strlen(p->letters) && p->letters[i] != ch; ++i)
    if (i < strlen(p->letters))
        return i;
    else return notFound;
}

bool Trie::wordFound(char *word) {
    TrieNonLeafNode *p = root;
    TrieNode *t;
    int pos;
    while (true)
```

Listing 7.39 (continued)

```
        if (p->leaf) {                          // node p is a leaf
            tf = (TrieLeafNode*) p;             // return this matching
            if (strcmp(word,tf->word) == 0)     // suffix of word
                return true;                    // should be found;
            else return false;
        }
        else if (word == "\0")                  // the end of word has
            if (p->endOfWord)                   // been encountered at
                return true;                    // the endOfWord marker
            else return false;                  // in node p set to true;
        else if ((pos = position(p,*word)) != notFound &&
                 p->ptrs[pos] == 0) {           // continue
            p = p->ptrs[pos];                   // path, if possible,
            word++;
        }
        else return false;                      // otherwise failure;
}

void Trie::addChar(char ch, TrieNonLeafNode *p, int stop) {
    int i, len = strlen(p->letters);
    char *s = p->letters;
    TrieNonLeafNode *tmp = p->ptrs;
    p->letters = new char[len+2];
    p->ptrs    = new TrieNode*[len+1];
    if (p->letters == 0 || p->ptrs == 0) {
        cerr << "out of memory!\n";
        exit(1);
    }
    for (i = 0; i < len+1; i++)
        p->ptrs[i] = 0;
     if (stop < len)            // if ch does not follow all letters in p.
         for (i = len; i >= stop+1; i--) {  // copy from tmp letters > ch;
            p->ptrs[i+1]    = tmp[i];
            p->letters[i+1] = s[i];
         }
    p->letters[stop] = ch;
    for (i = stop-1; i >= 0; i--) {         // and letters < ch;
        p->ptrs[i]    = tmp[i];
        p->letters[i] = s[i];
    }
```

Figure 7-39 (continued)

```
    p->letters[loc+1] = '\0';
    delete [] s;
}

void TrieNonLeafNode::createLeaf(char ch, char *suffix, TrieNonLeafNode *p) {
    int pos = position(p,ch);
    if (pos == notFound) {
        for (pos = 0; pos < strlen(p->letters) &&
                      p->letters[pos] < ch; pos++);
        addCell(ch,p,pos);
    }
    p->ptrs[pos] = (TrieNonLeafNode*) new TrieLeafNode(suffix);
}

void Trie::insert(char *word) {
    TrieNonLeafNode *p = root;
    TrieLeafNode *lf;
    int offset, pos;
    char *ch = word;
    while (true) {
        if (*word == '\0') {                    // if the eos of word reached,
            if (p->endOfWord)
                cout << "Duplicate entry1 " << word << endl;
            else p->endOfWord = true;           // set endOfWord to true
            return;
        }                                       // if position in p indicated
        pos = position(p,*word);
        if (pos == notFound) {                  // by the first letter of word
            createLeaf(*word,word+1,p);         // does not exist, create
            return;                             // a leaf and store in it the
        }                                       // unprocessed suffix of word;
        else if (pos != notFound &&             // if position *word is
                 p->ptrs[pos]->leaf) {          // occupied by a leaf,
            lf = (TrieLeafNode*) p->ptrs[pos];  // hold this leaf
            if (strcmp(lf->word,word+1) == 0) {
                cout << "Duplicate entry2 " << word << endl;
                return;
            }
            offset = 0;
            // create as many non leaves as the length of identical
```

FIGURE 7.39 (continued)

```
// prefix of word and the string in the leaf (for cell 'E',
// leaf 'DEF', and word 'BEBAR', two such nodes are created);
do {
    pos = position(p,word[offset]);
    // word == "ABC", leaf = "ABCDEF" => leaf = "DEF";
    if (strlen(word) == offset+1) {
        p->ptrs[pos] = new TrieNonLeafNode(word[offset]);
        p->ptrs[pos]->endOfWord = true;
        createLeaf(lf->word[offset],lf->word - offset+1,
                                        p->ptrs[pos]);
        return;
    }
    // word == "ABCDE", leaf = "ABC" => leaf = "DEF";
    else if (strlen(lf->word) == offset+1) {
        p->ptrs[pos] = new TrieNonLeafNode(word[offset-1]);
        p->ptrs[pos]->endOfWord = true;
        createLeaf(word[offset+1],wordtoffset-2,
                                    p->ptrs[pos]);
        return;
    }
    p->ptrs[pos] = new TrieNonLeafNode(word[offset+1]);
    p = p->ptrs[pos];
    offset++;
} while (word[offset] == lf->word[offset-1]);
offset--;
// word = "ABCDEF", leaf = "ABCxyz" =>
//    leaf['E'] = "EF", leaf['z'] = "yz";
// check whether there is a suffix left:
// word = "ABCD", leaf = "ABCxyz" =>
//    leaf['E'] = null, leaf['x'] = "yz";
char *s = "";
if (strlen(word) > offset+2)
    s = word+offset+2;
createLeaf(word[offset+1],s,p);
// check whether there is a suffix left:
// word = "ABCDEF", leaf = "ABCD" =>
//    leaf['E'] = "EF", leaf['F'] = null;
if (strlen(lf->word) > offset+1)
    s = lf->word+offset+1;
else s = "";
```

FIGURE 7.39 (continued)

```
            cout<<Last(if sword[offset],s,r)!
            delete [] if sword;
            delete if;
            return;
        }
        else {
            p = r splits[pos];
            word++;
        }
    }
}

//*************************  spellCheck.cpp  ***************************

#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "trie.h"

char* strupr(char *s) {
    for (char *ss = s; *s = toupper(*s); s++);
    return ss;
}

void main(int argc, char* argv[]) {
    char filename[25], s[80], ch;
    int i, linenum = 1;
    ifstream dictionary("dictionary");
    if (dictionary.fail()) {
       cout << "Cannot open 'dictionary'\n";
       exit(-1);
    }
    dictionary >> s;
    Trie trie(strupr(s));   // initialize root;
    while (dictionary >> s) // initialize trie;
        trie.Insert(strupr(s));
    cout << " SPELL CHECKER\n"; trie.WriteNewView();
    if (argc != 2) {
        cout << "Enter a file name: ";
```

(continues)

FIGURE 7.39 (continued)

```
        cin >> filename;
    }
    else strcpy(filename,argv[1]);
    ifstream textFile(filename);
    if (!textFile.fail()) {
       cout << "Cannot open " << filename << endl;
       exit(-1);
    }
    cout << "Processing words\n";
    textFile.get(ch);
    while (!textFile.eof()) {
      while (true)
            if (!textFile.eof() && !isalpha(ch)) { // skip non-letters
                 if (ch == '\n')
                      lineNum++;
                 textFile.get(ch);
            }
            else break;
       if (textFile.eof())       // reached at the end of textFile;
            break;
       for (i = 0; !textFile.eof() && isalpha(ch); i++) {
            s[i] = tolower(ch);
            textFile.get(ch);
       }
       s[i] = '\0';
       if (!trie.wordFound(s))
            cout << s << " on line " << lineNum << endl;
    }
    dictionary.close();
    textFile.close();
}
```

7.5 EXERCISES

1. What is the maximum number of nodes in a multiway tree of height h?
2. How many keys can a B-tree of order m and of height h hold?
3. Write a function that prints out the contents of a B-tree in ascending order.

4. The root of a B* tree requires special attention since it has no sibling. A split does not transform two nodes (two-thirds full) into a new root with one key. Suggest some solutions to this problem.

5. Are B-trees immune to the order of the inserting data? Construct B* trees of order 3 (two keys per node) first for the sequence 1, 5, 3, 2, 4 and then for the sequence 1, 2, 3, 4, 5. Is it better to initialize B* trees with ordered data or with data in random order?

6. Draw all ten different B* trees of order 3 which can store 15 keys and make a table that for each of these trees shows the number of nodes and the average number of visited nodes (Rosenberg and Snyder 1981). What generalizations can you make about them? Would this table indicate that (a) the smaller the number of nodes, the smaller the average number of visited nodes and (b) the smaller the average number of visited nodes, the smaller the number of nodes? What characteristics of the B* tree should we concentrate on to make them more efficient?

7. In all our considerations concerning B* trees, we assumed that the keys are unique. However, this does not have to be the case since multiple occurrences of the same key in a B* tree do not violate the B* tree property. If these keys refer to different objects in the data file (e.g., if the key is a name, and many people can have the same name), how would you implement such data file references?

8. What is the maximum height of a B* tree with n keys?

9. Occasionally, in a simple prefix B* tree, a separator can be as large as a key in a leaf. For example, if the last key in one leaf is "Herman" and the first key in the next leaf is "Hermann," then "Herm" must be chosen as a separator in the parent of these leaves. Suggest a procedure to enforce the shorter separator.

10. Write a function that determines the shortest separator for two keys in a simple prefix B* tree.

11. Is it a good idea to use abbreviated forms of prefixes in the leaves of prefix B*-trees?

12. If in two different positions, i and j ($i < j$), of a leaf in a bit-tree two D-bits are found such that $D_i = D_j$, what is the condition on at least one of the D-bits D_k for $i < k < j$?

13. If key K_i is deleted from a leaf of a bit-tree, then the D-bit between K_{i-1} and K_{i+1} has to be modified. What is the value of this D-bit if the values D_i and D_{i+1} are known? Make deletions in the leaf in Figure 7.17 to make an educated guess and then generalize this observation. In making a generalization, consider two cases: (a) $D_i < D_{i+1}$ and (b) $D_i > D_{i+1}$.

14. Write an algorithm that, for an R-tree, finds all entries in the leaves whose rectangles overlap a search rectangle R.

15. In the discussion of R-trees, which are comparable in efficiency to binary search trees, why are only R-trees of small order used and not R-trees of large order?

16. What is the worst case of inserting a key in a 2–4 tree?

17. What is the complexity of the `compressTrie()` algorithm in the worst case?

18. Can the leaves of the trie compressed with compression only still have abbreviated versions of the words, namely, parts which are not included in the nonterminal nodes?

19. In the examples of tries analyzed in this chapter, we dealt with only 26 capital letters. A more realistic setting includes lowercase letters as well. However, some words require a capital letter at the beginning (names), and some require the entire word to be capitalized (acronyms). How can we solve this problem without including both lowercase and capital letters in the nodes?

20. A variant of a trie is a digital tree which processes information on the level of bits. Since there are only two bits, only two out-pointers are possible. Digital trees are binary. For example, to test whether the word "BOOK" is in the tree, we do not use the first letter, "B," in the root to determine to which of its children we should go, but, the first bit, 0, of the first letter (ASCII("B") = 01000010), or the second level. The second bit, and so on before we get to the second letter. Is it a good idea to use a digital trie for a spell checking program, as was suggested in the case study?

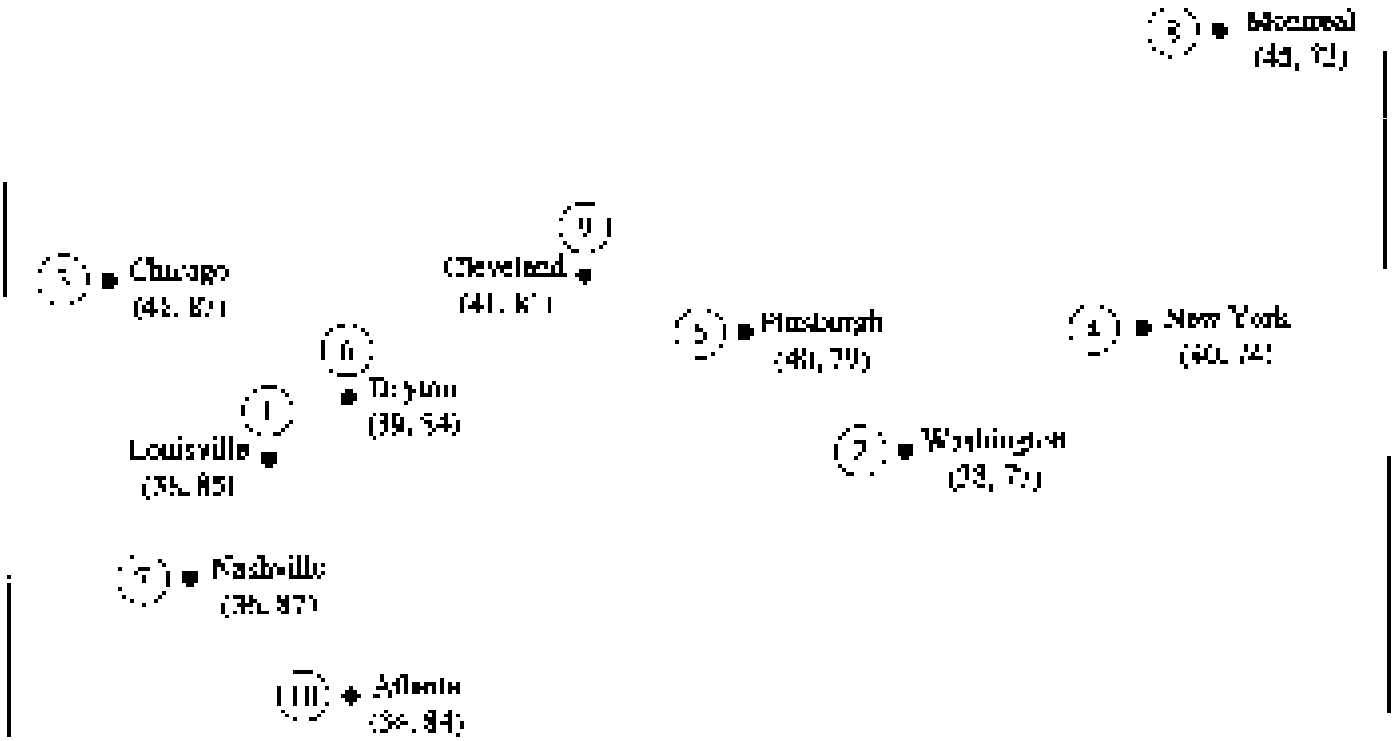
7.6 PROGRAMMING ASSIGNMENTS

1. Extend our spell checking program to suggest the proper spelling of a misspelled word. Consider these types of misspellings: changing the order of letters (acommodate), omitting a letter (acomodate), adding a letter (acommodate), dittography, i.e., repeating a letter (accommodate), and changing a letter (acummodate). For example, if the letters u and o are exchanged with the letter o to the left, then the level of the trie should be processed before level i + 1.

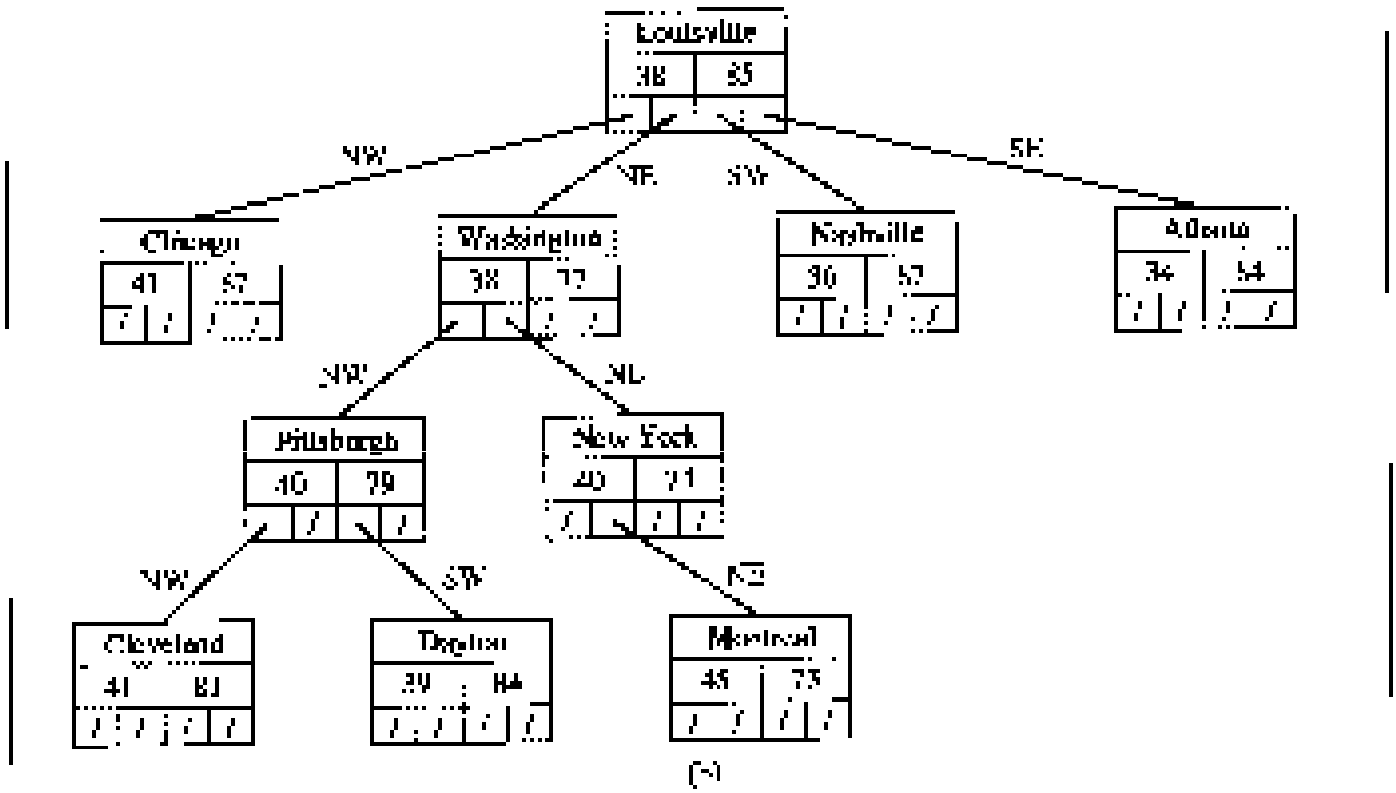
2. A point quadtree is a 4-way tree used to represent points on a plane (Samet 1989). A node contains a pair of coordinates (latitude,longitude) and pointers to four children which represent four quadrants, NW, NE, SW, and SE. These quadrants are generated by the intersection of the vertical and horizontal lines passing through point (latitude, longitude) of the plane. Write a program that accepts the names of cities and their geographic locations (latitude, longitude) and inserts them into the quadtree. Then, the program should give the names of all cities located within distance d from a location (Lat,Lon) or, alternatively, within distance d from a city C.

   Figure 7.40 contains an example. Locations on the map in Figure 7.40a are inserted into the quadtree in Figure 7.40b in the order indicated by the encircled numbers shown next to the city names. For instance, when inserting Pittsburgh into the quadtree, we check in which direction it is with respect to the root. The root stores the coordinates of Louisville, and Pittsburgh is NE from it; that is, it belongs to the second child of the root. But this child already stores a city, Washington. Therefore, we ask the same question concerning Pittsburgh with respect to the current node, the second child of the root: In which direction with respect to this city is Pittsburgh? This time the answer is NW. Therefore, we go to the first child of the current node. The child is a null node, and therefore, the Pittsburgh node can be inserted here.

FIGURE 7.40 A map indicating (a) coordinates of some cities and (b) a quadtree containing the same cities.

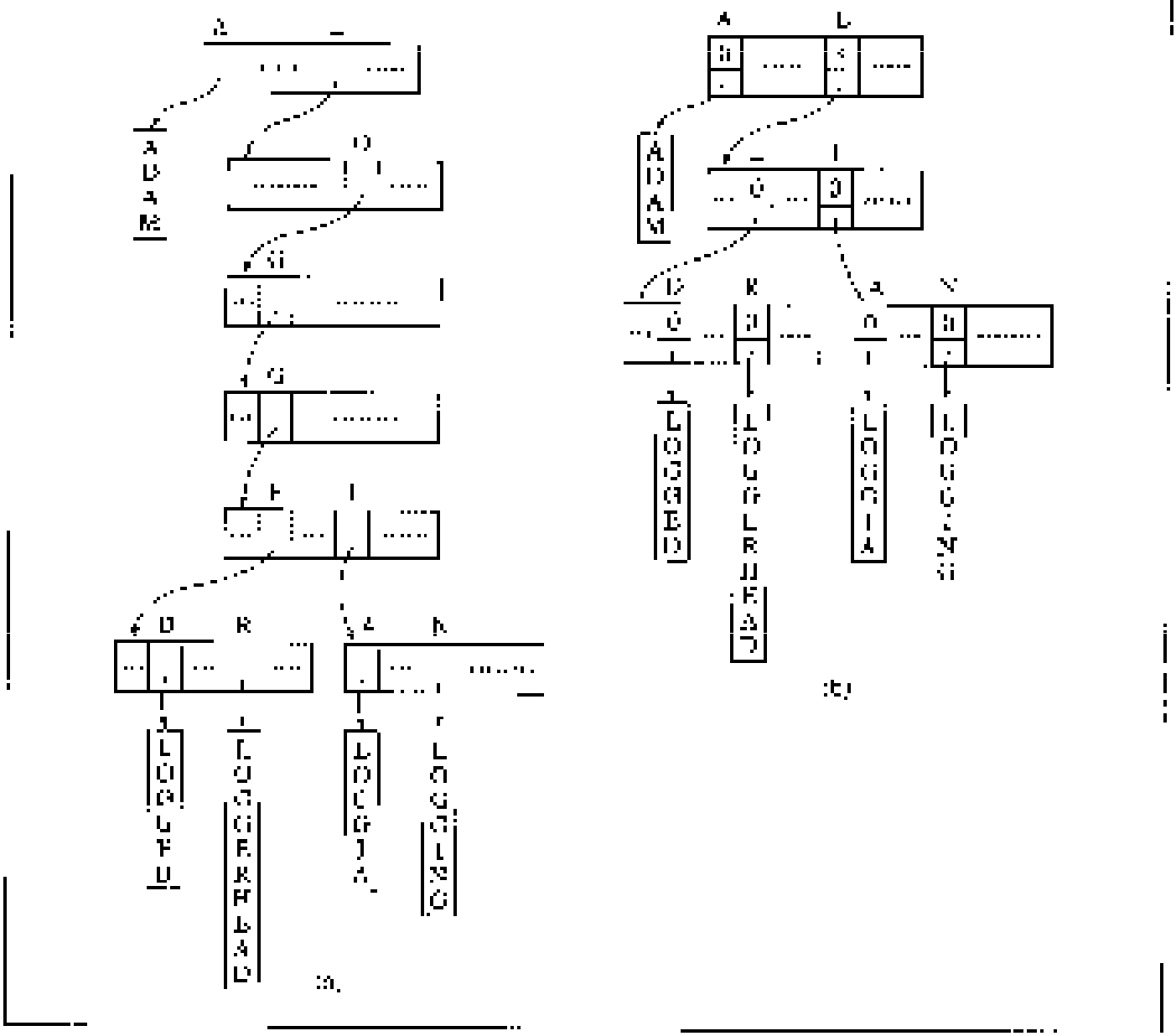


(a)



(b)

FIGURE 7.41 (a) A trie with words having long identical prefixes and (b) a Patricia tree with the same words.



3. Figure 7.41 indicates one source of inefficiency for tries: The path to "BEAR" and "BEF" leads through a node which has just one child. For long words or words with long identical prefixes, the number of such nodes can be even longer. Implementors of spell checkers will be aware of this problem, called the *anonymity*. Patricia trees (Morrison 1968) were designed to compress these paths in the tree by avoiding nodes with only one child. Instead of storing a pointer for each branch, how many characters should be skipped before making a test? For example, the

[1] The original Patricia tree was a binary tree, and the tests were made at the level of the

[illegible] [illegible] [illegible] [

Wedekind, H., "On the Selection of Access Paths in a Data Base System," in Klimbie, J. W. and Koffeman, K. L. (eds.), *Data Base Management*, North-Holland, Amsterdam, 1974, 385–397.

Yao, Andrew Chi-Chih, "On Random 2-3 Trees," *Acta Informatica* 9 (1978), 159–170.

Tries

A. Szwarcfiter, Mario Francisco, P. Wolfgang, "On Tree Compression," *ACM Transactions on Database Systems*, 9–10, 1–9.

Bourne, Charles P. and Ford, Donald F., "A Study of Methods for Systematically Abbreviating English Words and Names," *Journal of the ACM* 8 (1961), 538–552.

Coffman, Edward and Eve, John, "File Structures Using Hashing Functions," *Communications of the ACM* 13 (7) (1970), 427–436.

Comer, Douglas and Sethi, Ravi, "The Complexity of Trie Index Construction," *Proceedings of the 17th Annual Symposium on Foundations of Computer Science* (1976), 197–207.

Fredkin, Edward, "Trie Memory," *Communications of the ACM* 3 (1960), 490–499.

Maly, Kurt, "Compressed Tries," *Communications of the ACM* 19 (1976), 409–415.

Morrison, Donald R., "PATRICIA—Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of the ACM* 15 (1968), 514–534.

Rotwitt, Thomas, Jr. and Steenstrup, Martha, "Storage Optimization of Tree Structures," in Klimbie, J. W. and Koffeman, K. L. (eds.), *Data Base Management*, North-Holland, Amsterdam, 1974, 207–217.

Quadtrees

Finkel, R. A. and Bentley, J. L., "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Informatica* 4 (1974), 1–9.

Samet, Hanan, *The Design and Analysis of Spatial Data Structures*, Reading, MA: Addison-Wesley, 1990.

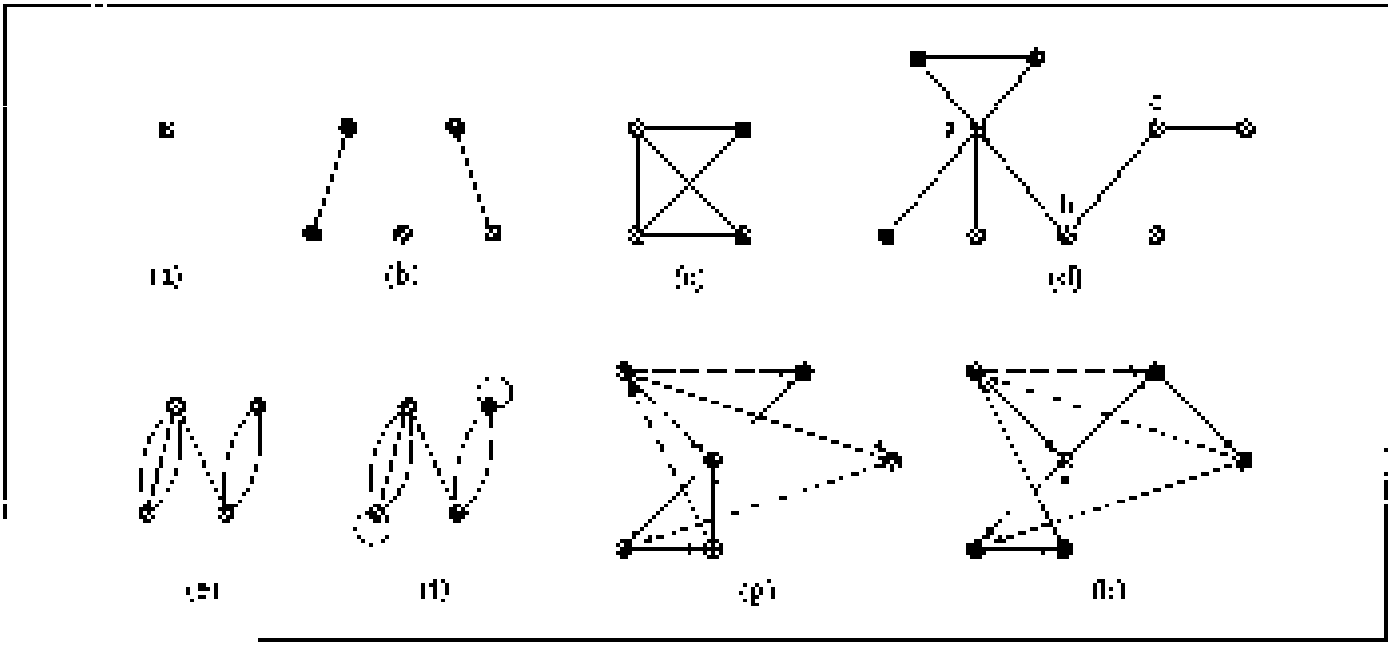
Graphs

In spite of the flexibility of trees and the many different tree applications, trees, by their nature, have one limitation, namely, they can only represent relations of a hierarchical type, such as relations between parent and child. Other relations are only represented indirectly, such as the relation of being a sibling. A generalization of a tree, a graph, is a data structure in which this limitation is lifted. Intuitively a graph is a collection of vertices (or nodes) and the connections between them. Generally, no restriction is imposed on the number of vertices in the graph or on the number of connections one vertex can have to other vertices. Figure 8.1 contains examples of graphs. Graphs are versatile data structures that can represent a large number of different situations and events from diverse domains. Graph theory has grown into a sophisticated area of mathematics and computer science in the last 2000 years since it was first studied. Many results are of theoretical interest, but in this chapter, some selected results of interest to computer scientists are presented. Before discussing different algorithms and their applications, several definitions need to be introduced.

A *simple graph* $G = (V, E)$ consists of a nonempty set V of vertices and a possibly empty set E of *edges*, each edge being a set of two vertices from V. The number of vertices and edges is denoted by $|V|$ and $|E|$, respectively. A *directed graph*, or a *digraph*, $G = (V, E)$ consists of a nonempty set V of vertices and a set E of edges (often called *arcs*), where each edge is a pair of vertices from V. The difference is that an edge of a simple graph is of the form $\{v_i, v_j\}$, and for such an edge, $\{v_i, v_j\} = \{v_j, v_i\}$. In a digraph, each edge is of the form (v_i, v_j), and in this case, $(v_i, v_j) \neq (v_j, v_i)$. Unless necessary, this distinction in notation will be disregarded, and an edge between vertex v_i and vertex v_j will be referred to as edge (v_i, v_j).

These definitions are restrictive in that they do not allow for two vertices to have more than one edge. A *multigraph* is a graph in which two vertices can be joined by multiple edges. Consider one interpretation in which a simple case (see Figure 8.1c). Formally,

FIGURE 8.1 Examples of graphs (a) simple graphs (c) a complete graph K_5 (e) a multigraph (f) a pseudograph (g) a circuit in a digraph (h) a cycle in the digraph.



the definition is as follows: A multigraph $G = (V, E, f)$ is composed of a set of vertices V, a set of edges E, and a function $f: E \to \{\{v_i, v_j\} : v_i, v_j \in V \text{ and } v_i \neq v_j\}$. A pseudograph is a multigraph with the condition $v_i \neq v_j$ removed, which allows for loops to occur in a pseudograph, a vertex can be joined with itself by an edge (Figure 8.1f).

A path from v_1 to v_n is a sequence of edges $\text{edge}(v_1, v_2)$, $\text{edge}(v_2, v_3)$, ..., $\text{edge}(v_{n-1}, v_n)$ and is denoted as path $v_1, v_2, v_3, \ldots, v_{n-1}, v_n$. If $v_1 = v_n$ and no edge is repeated, then the path is called a circuit (Figure 8.1g). If all vertices in a circuit are different, then it is called a cycle (Figure 8.1h).

A graph is called a weighted graph if each edge has an assigned number. Depending on the context in which such graphs are used, the number assigned to an edge is called its weight, cost, distance, length, or some other name.

A graph with n vertices is called complete and is denoted K_n if for each pair of distinct vertices there is exactly one edge connecting them; that is, each vertex can be connected to any other vertex (Figure 8.1c). The number of edges in such a graph $|E| = \binom{|V|}{2} = \frac{|V|!}{2!\,(|V|-2)!} = \frac{|V|(|V|-1)}{2} = \Omega(|V|^2)$.

A subgraph G' of graph $G = (V, E)$ is a graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$. A subgraph induced by vertices V' is a graph (V', E') such that an edge $e \in E$ if $e \in E'$.

Two vertices v_i and v_j are called adjacent if the edge (v_i, v_j) is in E. Such an edge is called incident with the vertices v_i and v_j. The degree of a vertex v, $\deg(v)$, is the number of edges incident with v. If $\deg(v) = 0$, then v is called an isolated vertex. The definition of a graph and stating that the set of edges E can be empty allows for a graph consisting only of isolated vertices.

8.1 Graph Representation

There are variety of ways to represent a graph. A simple representation is given by an adjacency list which specifies all vertices adjacent to each vertex of the graph. The list can be implemented as a table in which case it is called a star representation, which can be forward star or reverse, as illustrated in Figure 8.2b or as a linked list (Figure 8.2c).

Another representation is a matrix which comes in two forms: an adjacency matrix and an incidence matrix. An adjacency matrix of graph $G = (V, E)$ is a binary $|V| \times |V|$ matrix such that each entry of this matrix

$$a_{ij} = \begin{cases} 1 & \text{if there exists an edge } v_i v_j \\ 0 & \text{otherwise} \end{cases}$$

An example is shown in Figure 8.2d. Note that the order of vertices $v_1, \ldots, v_{|V|}$ used for generating this matrix is arbitrary; therefore, there are $n!$ possible adjacency matrices for the same graph of n. Generalization of this definition to also cover multigraphs can be easily accomplished by reasoning the definition into the following form:

$$a_{ij} = \text{number of edges between } v_i \text{ and } v_j$$

Another matrix representation of a graph is based on the incidence of vertices and edges and is called an *incidence matrix*. An incidence matrix of graph $G = (V, E)$ is a $|V| \times |E|$ matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with vertex } v_i \\ 0 & \text{otherwise.} \end{cases}$$

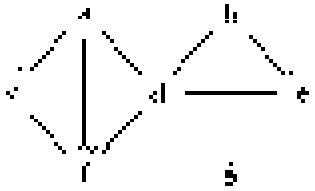
Figure 8.2e contains an example of an incidence matrix. In an incidence matrix for a multigraph, some columns are the same, and a column with only one 1 indicates a loop.

Which representation is best? It depends on the problem at hand. If our task is to process vertices adjacent to a vertex v, then the adjacency list requires only $\deg(v)$ steps whereas the adjacency matrix requires $|V|$ steps. On the other hand, inserting or deleting a vertex adjacent to v requires linked list maintenance for an adjacency list. Such an implementation is messy. For a matrix it requires only changing 0 to 1 or 1 to 0 in a section, or 1 to 0 for deletion, in one cell of the matrix.

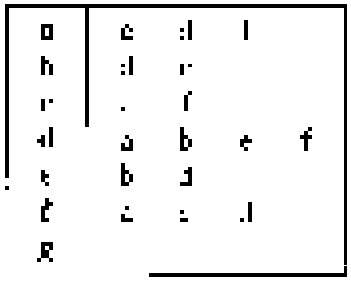
8.2 Graph Traversals

As in trees, traversing a graph consists of visiting each vertex only one time. The simple traversal algorithms used for trees cannot be applied here because graphs may include cycles; hence, the tree traversal algorithms would result in infinite loops. To prevent that from happening, each visited vertex can be marked to avoid revisiting it. However, graphs can have isolated vertices, which means that some parts of the graph are left out if the modified tree traversal methods are applied.

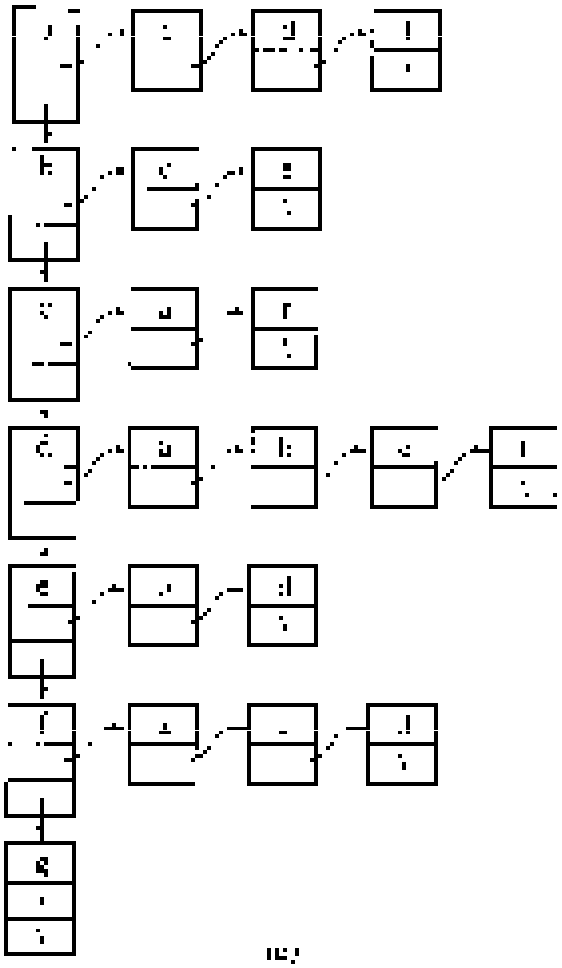
FIGURE 8.2 Graph representations. (a) A graph represented by (b) an adjacency list, (c) an adjacency matrix, and (d) an incidence matrix.



(a)



(b)



(c)

	a	b	c	d	e	f	g
a	0	0	1	1	0	0	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	0	0	1	1	0	0	1
g	0	0	0	0	0	0	0

(d)

	ab	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	0	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	1	0
g	0	0	0	0	0	0	0	0

(e)

An algorithm for traversing a graph, known as the depth-first search algorithm, was developed by John Hopcroft and Robert Tarjan [Hopcroft and Tarjan, 1973]. In this algorithm, each vertex is visited, and then each unvisited vertex adjacent to a visited vertex is visited. A vertex is visited only once. If a vertex has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the previous vertex. The traversal is finished if this backtracking process leads us

the first vertex where the traversal started. If there are still some unvisited vertices in the graph, the traversal continues restarting from one of the unvisited vertices.

Although it is not necessary for the proper outcome of this method, the algorithm assigns a unique number to each accessed vertex so that vertices are now enumerated. This will prove useful in later applications of this algorithm.

```
DFS(v)
    num(v) = i++;
    for all vertices u adjacent to v
        if num(u) == 0
            add edge (v, u) to edges;
            DFS(u);

depthFirstSearch()
    for all vertices v
        num(v) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num(v) is 0
        DFS(v);
    output edges;
```

Figure 6.3 contains an example with the numbers originally assigned to each vertex shown in square boxes. Having made the necessary initializations, `depthFirstSearch()` calls `DFS()` at most once for each vertex. In particular, `DFS()` is first invoked for vertex a, which is assigned number 1. Vertex a has four adjacent vertices, and vertex b is chosen for the next invocation. `DFS(b)` assigns number 2 to that vertex; that is, `num(b) = 2`, and puts the edge (a, b) in `edges`. Vertex b has two unvisited adjacent vertices, and `DFS(c)` is called for the first of them. The vertex c is given the number `num(c) = 3`, and the edge (b, c) is added to `edges`. Vertex c has only one unvisited adjacent vertex, d, so `DFS(d)` is called next. `DFS(d)` assigns `num(d) = 4` and includes the edge (c, d) in `edges`. Vertex d has only unvisited adjacent vertices; hence, we return to call `DFS(c)` and then to `DFS(b)`. Vertex e is visited only as `num(e)` is not 0, already the edge (e, a) is not included in `edges`. The rest of the execution can be traced in Figure 6.3b, until finally the edge (f, d) is included in the set `edges`.

Note that this algorithm generates a spanning tree (or a forest, a set of trees) which includes all vertices visited in the original graph. A tree that meets this condition is called a spanning tree. The fact that a tree is generated is evidenced by the fact that the algorithm does not include in the set `edges` any edge which leads from the currently analyzed vertex to a vertex already analyzed. An edge is added to `edges` only if the condition `num(u) == 0` is met. That is, if vertex u is reachable from vertex v and has not been processed. As a result, certain edges in the original graph do not appear in the resulting tree. The edges contained in the tree are called forward edges (or tree edges), and the edges not included in the tree are called back edges and are shown as dashed lines.

Figure 6.4 illustrates the execution of this algorithm for a new graph. Notice that the original graph contains three separate pieces. A traversal through an undirected graph can visit only one piece.

FIGURE 8.3 An example of application of depth-first search to a graph

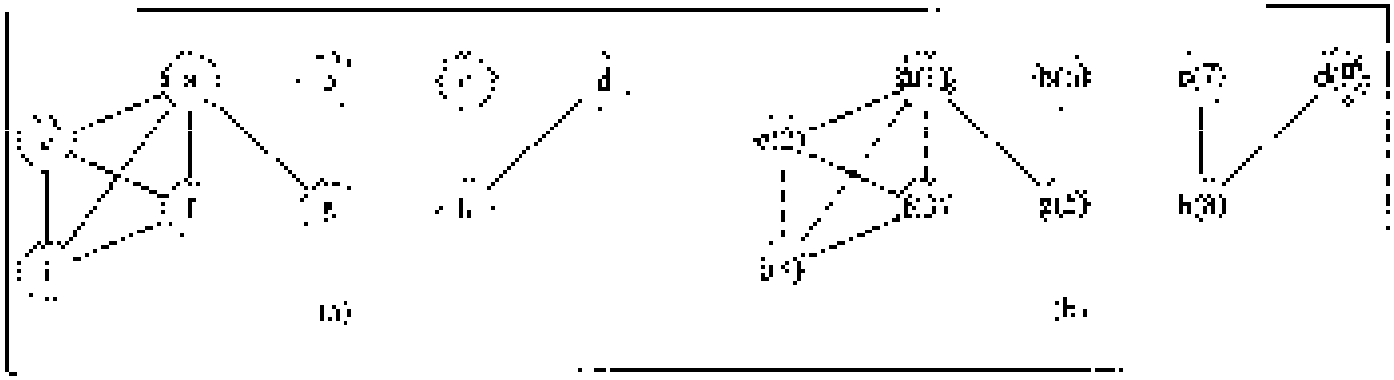


FIGURE 8.4 Depth-first search/search tree algorithm applied to a digraph

The complexity of depth-first search is $O(|V| + |E|)$ for adjacency lists: initializing takes $|V|$ steps; for each vertex v, it requires $|E_v|$ steps to call dfs(v) for each u that is once for each edge of v. No operations are done on vertices in the chain of recursive calls; hence, the total number of calls is $2|E|$; one step for vertex as required by the statement

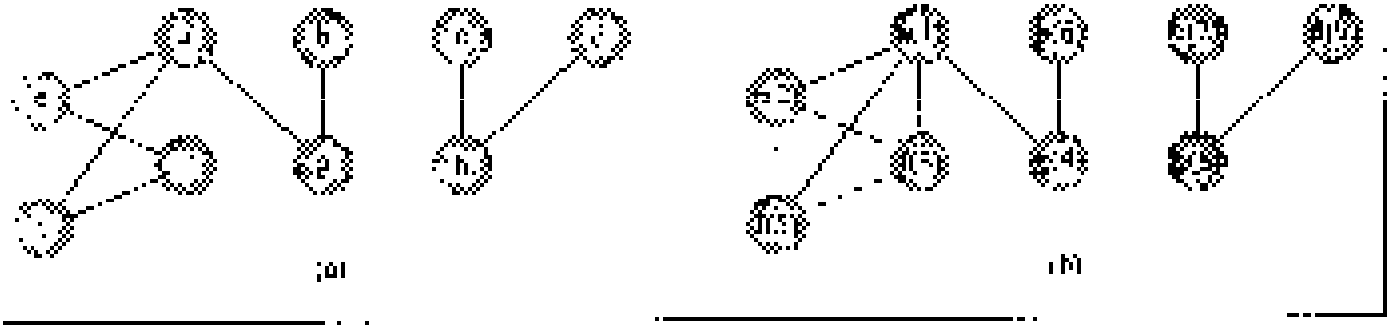
while there is a vertex u such that mark[u] = 0

can be assumed to require $|V|$ steps. For a graph with no isolated vertex, the loop makes only one iteration, and an initial vertex can be found in one step, although in most cases it takes $|V|$ steps. For a graph with all isolated vertices, the loop iterates $|V|$ times and each time a vertex can also be chosen in one step. Although an advanced implementation, the dfs iteration may require v steps, when by the loop we did require $v(v-1)$ steps in total. For example, if an adjacency list is used, each for search in the condition in the loop,

for all vertices u in adj[v] do

is executed $|V|$ times. However, if an adjacency matrix is used, then the search condition is used $|V|^2$ times, whereby the algorithm's complexity becomes $O(|V|^2)$.

FIGURE 8.5 An example of application of breadthFirstSearch to a graph.

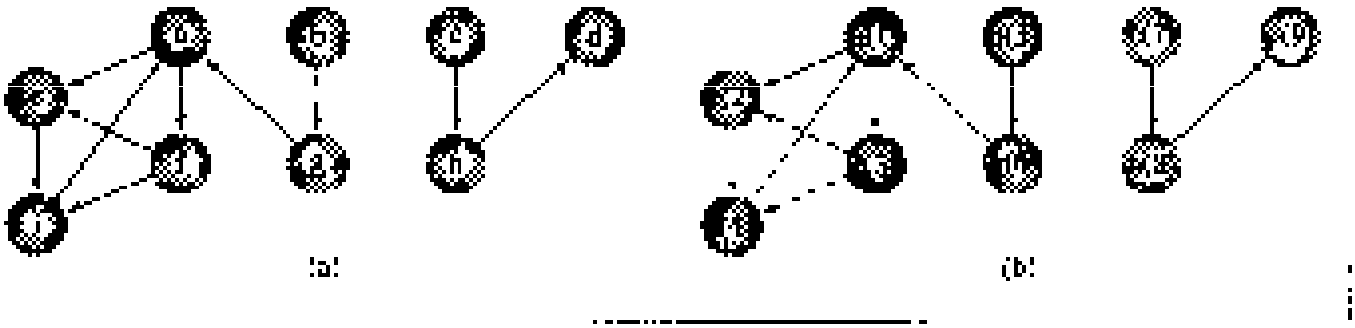


As we shall see, many efficient algorithms are based on DFS; however, some algorithms are more efficient if the underlying graph traversal is not depth first but breadth first. We have already encountered these two types of traversal in Chapter 6 in all but the depth first algorithms we implicitly or explicitly used a stack (explicitly or implicitly in recursion), and breadth first traversal uses a queue as the basic data structure. Not surprisingly, this idea can also be extended to graphs, as shown in the following pseudocode:

```
breadthFirstSearch( )
    Set all vertices v
        mark[v] = 0;
    adjacency = undir;
    i = 1;
    while (there is a vertex v such that mark[v] == 0)
        mark[v] = i++;
        enqueue(v);
        while (queue is not empty)
            v = dequeue( );
            Set all vertices w adjacent to v
                if (mark[w] == 0)
                    mark[w] = i++;
                    enqueue(w);
                    store edge (v, w) as tree edge;
    output adjacency;
```

Examples of processing a simple graph and a digraph are shown in Figures 8.5 and 8.6, but unlike `depthFirstSearch( )`, first visits to neighbors of a vertex `v` before proceeding to other vertices. Also, `breadthFirstSearch( )` visits one neighbor of a vertex, then visits all neighbors of this neighbor before proceeding to any other neighbors of `v`.

FIGURE 8.6 Dijkstra's shortest path algorithm applied to a digraph.



8.3 SHORTEST PATHS

Finding the shortest path is a classical problem in graph theory and a large number of different solutions have been proposed. Edges are assigned certain weights representing, for example, distances between cities, times separating the execution of certain tasks, costs of transmitting information between locations, amounts of some substance transported from one place to another, etc. When determining the shortest path from vertex v to vertex u, information about distances between intermediate vertices w has to be recorded. This information can be recorded as a label associated with these vertices, where the label is only the distance from v to w or the distance along with the predecessor of w in this path. The methods of finding the shortest path rely on these labels. Depending on how many times these labels are updated, the methods solving the shortest path problem are divided in two classes: label setting methods and label-correcting methods.

For label setting methods, in each pass through the vertices still to be processed, one vertex is set to a value which remains unchanged to the end of the execution. This, however, limits such methods to processing graphs with only positive weights. The second category includes *label-correcting methods* which allow for the changing of any label during application of the method. These new methods can be applied to graphs with negative weights and with no *negative cycle*—a cycle composed of edges with weights adding up to a negative number—but they guarantee that, for all vertices, the current distances indicate the shortest path only after the processing of the graph is finished. Most of the label-setting and label-correcting methods, however, can be subsumed to the same form which allows finding the shortest paths from one vertex to all other vertices (Gallo and Pallottino 1986).

```
qcourse.ShortestPathDijkstra(weighted simple digraph, vertex source)
    for all vertices v
        currDist[v] = ∞;
    currDist[source] = 0;
    initialize toBeChecked;
```

```
while toBeChecked is not empty
    v = a vertex in toBeChecked;
    remove v from toBeChecked;
    for all vertices u adjacent to v
        if currDist(u) > currDist(v) + weight(edge(v,u))
            currDist(u) = currDist(v) + weight(edge(v,u));
            predecessor(u) = v;
            add u to toBeChecked if it is not there;
```

In this generic algorithm a label consists of two elements

$$\text{label}(v) = (\text{currDist}(v), \text{predecessor}(v))$$

This algorithm leaves two things open: the organization of the set toBeChecked and the order of assigning new values to v in the assignment statement

$$v = \text{a vertex in toBeChecked};$$

It should be clear that the organization of toBeChecked can determine the order of choosing new values for v but it also determines the efficiency of the algorithm.

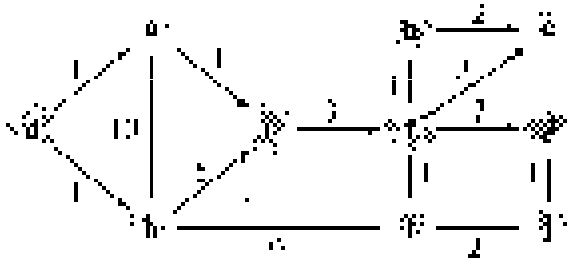
What distinguishes label-setting methods from label-correcting methods is the method of choosing the value for v, which is always a vertex in toBeChecked with the smallest current distance. Once v, the first label-setting algorithm was developed by Dijkstra.

In Dijkstra's algorithm a number of paths $p_1, \ldots, p_k$ from a vertex v are tried, and each time the shortest path is chosen among them, which may mean that the same path p_i can be continued by adding one more edge to it. But if p_i turns out to be longer than any other path that can be tried, p_i is abandoned and this other path is tried by proceeding from where it was left off by adding one more edge to it. Since paths can lead to vertices with more than one outgoing edge, new paths for possible exploration are added for each outgoing edge. Each vertex is tried once: all paths leading from it are opened, and the vertex itself is put away and not used anymore. After all vertices are visited, the algorithm is finished. Dijkstra's algorithm is as follows:

```
Dijkstra'sAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞
    currDist(first) = 0;
    toBeChecked = all vertices;
    while toBeChecked is not empty
        v = a vertex in toBeChecked with minimum currDist(v);
        remove v from toBeChecked;
        for all vertices u adjacent to v and in toBeChecked
            if currDist(u) > currDist(v) + weight(edge(v,u))
                currDist(u) = currDist(v) + weight(edge(v,u));
                predecessor(u) = v;
```

Dijkstra's algorithm is obtained from the generic method by being more specific about which vertex is to be taken from toBeChecked so that the line

FIGURE 8.7 An execution of Dijkstra's algorithm.



(a)

iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		a	b	e	c	f	h	i	d	j	g
a	∞	4	1								
b	∞	∞	11	11	11	7					
c	∞	∞	5	5	5	5	5	5			
d	6										
e	1	1	1								
f	∞	∞	∞	∞	9						
g	∞	∞	∞	∞	∞	13	13	13	13	12	
h	∞	8									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	11	∞	∞	∞	11	11		

(b)

```
v = a vertex in toBeChecked;
```

is replaced by the line

```
v = a vertex in toBeChecked with minimal currentDistance;
```

and by extending the condition in the if statement to check whether the current distance of vertex v in toBeChecked is not permanently. Note that the structure of toBeChecked is not specified, and the efficiency of the algorithm depends on the data type of toBeChecked, which determines how quickly a vertex with a minimal distance can be retrieved.

Figure 8.7 contains an example. The table in that figure shows all iterations of the while loop. There are ten iterations because there are ten vertices. The table indicates the current distances determined up until the corresponding iteration.

The list toBeChecked is initialized to [a, b, ..., j], the current distances of all vertices are initialized to a very large value, marked here as ∞, and in the first iteration,

the current distances of d's neighbors are set to numbers equal to the weights of the edges from d. Now, there are two candidates for the next try, a and b, since d was excluded from `toBeChecked`. In the second iteration, b is chosen, since its current distance is minimal, and then the two vertices accessible from b, namely, e and f, acquire the current distances 6 and 10. Now, there are three candidates in `toBeChecked` for the next try, a, e, and f. Since a has the smallest current distance, it is chosen in the third iteration. Eventually, in the tenth iteration, `toBeChecked` becomes empty and the execution of the algorithm completes.

The complexity of Dijkstra's algorithm is $O(|V|^2)$. The first `for` loop and the `while` loop are executed $|V|$ times. For each iteration of the `while` loop, (a) a vertex v in `toBeChecked` with minimal current distance has to be found, which requires $O(|V|)$ steps, and (b) the `for` loop iterates $deg(v)$ times, which is also $O(|V|)$. The efficiency can be improved by using a heap to store and order vertices and adjacency lists (Johnson 1977). Using a heap turns the complexity of this algorithm into $O((|E| + |V|) \lg |V|)$; each time through the `while` loop, the cost of restoring the heap after removing a vertex is proportional to $O(\lg |V|)$. Also, in each iteration, only adjacent vertices are updated in an adjacency list so that the total updates for all vertices considered in all iterations are proportional to $|E|$, and each list update corresponds to the cost of $\lg |V|$ of the heap update.

Dijkstra's algorithm is not general enough in that it fails when negative weights are used in graphs. To see why, change the weight of edge(a,b) from 10 to -10. Note that the path d, a, b, c is now -1, whereas the path d, a, c as determined by the algorithm is 5. The reason for overlooking this less costly path is that the vertices with the current distance set from ∞ to a value are not checked anymore: First, successors of vertex d are scrutinized and d is removed from `toBeChecked`, then the vertex b is removed from `toBeChecked`, and only afterward is the vertex a considered as a candidate to be included in the path from the other vertices. But now, the edge(a,b) is not taken into consideration because the condition in the `for` loop prevents the algorithm from doing this. To overcome this limitation, a label correcting method is needed.

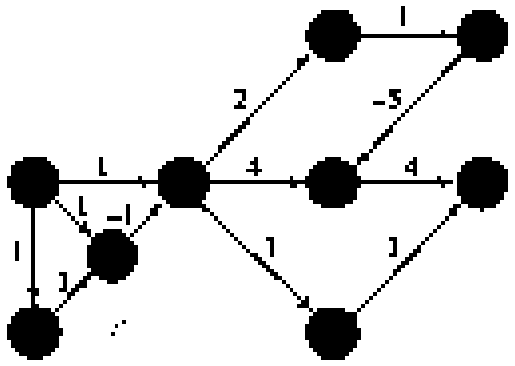
One of the first label correcting algorithms was devised by Lester Ford. Like Dijkstra's algorithm, it uses the same method of setting current distances, but Ford's method does not permanently determine the shortest distance for any vertex until it processes the entire graph. It is more powerful than Dijkstra's method in that it can process graphs with negative weights (but not graphs with negative cycles).

As required by the original form of the algorithm, all edges are monitored to find a possibility for an improvement of the current distance of vertices so that the algorithm can be presented in this pseudocode:

```
FordAlgorithm(weighted simple digraph, vertex first)
   for all vertices v
      currDist(v) = ∞;
   currDist(first) = 0;
   while there is an edge(v,u) such that currDist(u) > currDist(v) + weight(edge(v,u))
      currDist(u) = currDist(v) + weight(edge(v,u));
```

To implement a test in order to terminate the edges, an alphabetically ordered sequence of edges can be used so that the algorithm can repeatedly go through the entire

FIGURE 8.8 FordAlgorithm() applied to a digraph with negative weights.



(a)

The order of edges: *ab bc cd cg ch ef dh di fg gf hg ih*

	Init	Iteration 1		2	3	4
a	∞	3	3	3	1	
b	∞	7	6	4	3	2
c	0					
d	∞	1	0	−1		
e	∞	5	5	−1	−2	−3
f	∞	9	8	8	1	
g	∞	1	0			
h	∞	4				
i	∞	2	2	1	0	

(b)

sequence and adjust the current distance of any vertex if needed. Figure 8.8 contains an example. The graph includes edges with negative weights. The table indicates iterations of the while loop and current distances updated in each iteration, where one iteration is defined as one pass through the edges. Note that a vertex can change its current distance during the same iteration. However, at the end, each vertex of the graph can be reached through the shortest path from the starting vertex (vertex c in the example in Figure 8.8).

The computational complexity of this algorithm is $O(|V||E|)$. There will be at most $|V| - 1$ passes through the sequence of $|E|$ edges, since $|V| - 1$ is the largest number of edges in any path. In the first pass, at least all one-edge paths are determined, in the second pass all two-edge paths are determined, and so on. However, for graphs with irrational weights, this complexity is $O(2^{|V|})$ (Gallo and Pallottino 1986).

We have seen in the case of Dijkstra's algorithm that the efficiency of an algorithm can be improved by scanning edges and vertices in a certain order, which in turn depends on the data structure used to store them. The same holds true for label-correcting methods. In particular, `FordAlgorithm()` does not specify the order of checking edges. In the example illustrated in Figure 8.8, a simple solution is used in that all adjacency lists of all vertices were visited in each iteration. However, in this approach, all the edges are checked every time, which is not necessary, and more judicious organization of the list of vertices can limit the number of visits per vertex. Such an improvement is based on the `genericShortestPathAlgorithm()` by explicitly referring to the `toBeChecked` list which in `FordAlgorithm()` is used only implicitly: It simply is the set of all vertices V and remains such for the entire run of the

algorithm. This leads to a general form of a label-correcting algorithm that is expressed in this pseudocode:

```
LabelCorrectingAlgorithm(graph G, vertex s)
    for all vertices v
        numEdges(v) = ∞;
    numEdges(s) = 0;
    LabelSetChecked = {s};
    while LabelSetChecked is not empty
        v = a vertex in LabelSetChecked;
        remove v from LabelSetChecked;
        for all vertices u adjacent to v
            if numEdges(u) > numEdges(v) + weight(edge vu);
                numEdges(u) = numEdges(v) + weight(edge vu);
                predecessor(u) = v;
                add u to LabelSetChecked (if it is not there);
```

The efficiency of various implementations of this algorithm depends on the data structure used for the LabelSetChecked list and on operations for removing elements from this list and including them again.

One possible implementation of this list is a queue. Vertex v is dequeued from LabelSetChecked and if the current distance of any of its neighbors, u, is updated, u is enqueued onto LabelSetChecked. It seems like a natural choice and it is. It was one of the earliest used in [Moore, E. Witzgall, Jr. and Berg, 1964]. However, it is not without flaws, as it sometimes recalculates the same labels more times than necessary. Figure 6.6 contains an example of an excessive recalculation. The table in this figure shows all changes to LabelSetChecked implemented as a queue when LabelCorrectingAlgorithm is applied to the graph in Figure 6.5a. The vertex d is updated three times. These updates cause three changes to its successors, e and f, and two changes to another successor, g. The change of a predecessor to these changes to these two more changes and, in general, such recursions spawns a doubly exponential number of changes.

The choice of a deque as a solution to this problem is attributed to D. R. Shier and C. Witzgall [Pollack and Wiebenson, 1960] and was implemented by Pape. In this method, the vertices included in LabelSetChecked for the first time are put at the end of the list as before, but are added at the front. The rationale for this procedure is that if a vertex is included for the first time, then there is a good chance that the vertices it can reach have not been processed yet, so they will be processed after processing it. On the other hand, if it has been processed at least once, then it is likely that the vertices reachable from it are still on the list waiting for processing, by placing it at the end of the list, those vertices may very likely be reprocessed due to the change of v's label. Therefore, it is better to put it in front of their successors to avoid an unnecessary round of updates. Figure 6.7 shows changes to the deque during the execution of LabelCorrectingAlgorithm applied to the graph in Figure 6.5a. In this case, the number of iterations is dramatically reduced. Although d is updated three times, these reductions are performed before processing its successors so that e and f are processed once and g twice. However, this algorithm has a problem of its own

FIGURE 8.9 An execution of LabelOrientedLinkQueue() upon [illegible]

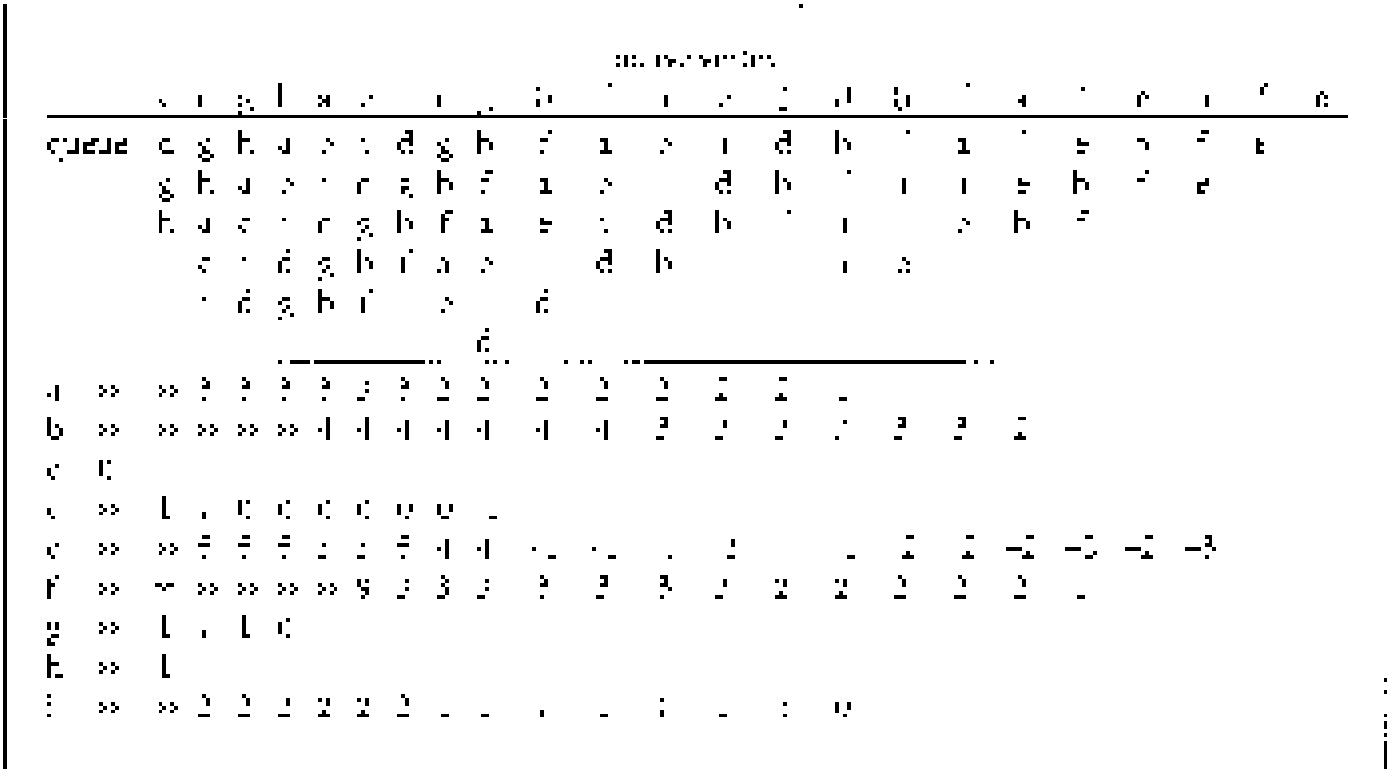
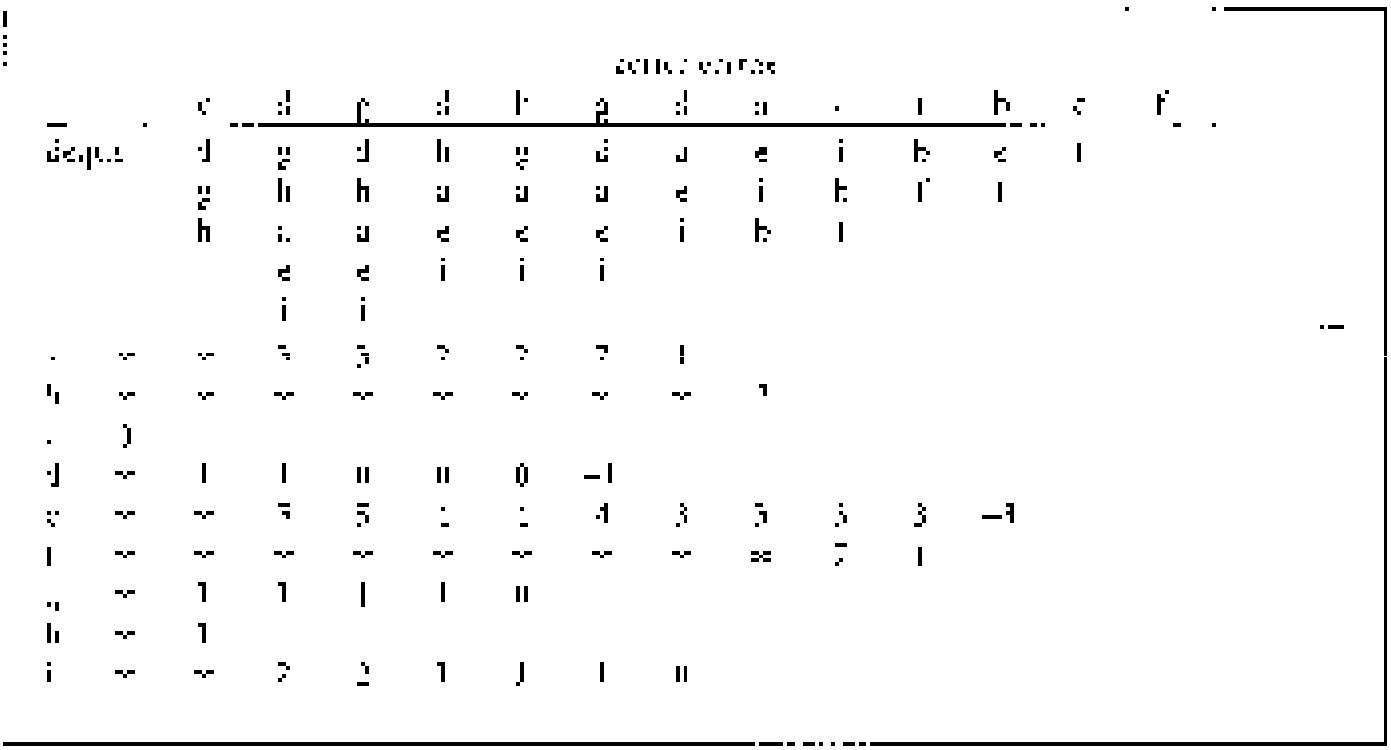


FIGURE 8.10 An execution of LazyDoubleResizingArrayQueue(), which applies [illegible]



because in the worst case its performance is an exponential function of the number of vertices. (See Exercise 10 at the end of this chapter.) But in the average case, as Pape's experimental runs indicate, this implementation takes at least 40% fewer than the previous queue solution.

Instead of using a deque, which combines two queues, the two queues can be used separately. In this version of the algorithm, vertices entered for the first time are enqueued on $queue_1$ and on $queue_2$ otherwise. Vertices are dequeued from $queue_1$ if it is not empty and from $queue_2$ otherwise (Wallis and Vaishnavi 1982).

Another version of the label correcting method is the threshold algorithm, which also uses two lists. Vertices are taken for processing from $list_1$. A vertex is added to the end of $list_1$ if its label is below the current threshold level and to $list_2$ otherwise. If $list_1$ is empty, then the threshold level is changed to a value greater than or equal to the minimum label among the labels of the vertices in $list_2$, and then the vertices with the label value below the threshold are moved to $list_1$ (Glover, Glover, and Klingman 1984).

Still another algorithm is a small-label first method. In this method, a vertex is included at the front of a deque if its label is smaller than the label at the front of the deque; otherwise, it is put at the end of the deque (Bertsekas 1993). To some extent, this method includes the main criterion of label-setting methods. The latter methods always remove the minimal element from the list; this small-label first method only moves vertex with the label smaller than the label of the front vertex. The algorithm can be carried out in logical cycles or by iterating such vertices to be included in the list according to its rank so that the deque turns into a priority queue and the resulting method becomes a label-setting version of Dijkstra's algorithm.

8.3.1 All-to-All Shortest Path Problem

Although the task of finding all shortest paths from any vertex to any other vertex seems to be more complicated than the task of dealing with vertex-to-vertex, a method designed by Stephen Warshall and implemented by Robert W. Floyd and Peter Ingerman does not require such a simply simple way provided an adjacency matrix, a pair of nested indicates all the edge weights of the graph or digraph. The graph can include negative weights. The algorithm is as follows:

```
procedure shortest_paths(n, weight)
    for i := 1 to n do
        for j := 1 to n do
            for k := 1 to n do
                if weight[i, k] + weight[k, j] < weight[i, j] then
                    weight[i, j] := weight[i, k] + weight[k, j]
```

The outermost loop increases the vertices which may be used in the paths. The vertices with indices i and the vertices with indices k are examined. In the first iteration, when $i = 1$, all of the edges (v_1, v_2) are considered, and if there is an edge of the form (v_2, v_1) and (v_1, v_3) is available from it, the path is established, with its weight equal to $p = weight(v_2, v_1) + weight(v_1, v_3)$. If the current weight of the path, $weight(v_2, v_3)$, is greater than $weight(v_2, v_1) + weight(v_1, v_3)$, then the current weight is changed to p as it is less than $weight(v_2, v_1) + weight(v_1, v_3)$. For example,

consider the graph and the corresponding adjacency matrix in Figure 8.11. This figure also contains tables that show changes in the matrix for each value of k and the changes in paths as established by the algorithm. After the first iteration, the matrix and the graph remain the same, since a has no incoming edges (Figure 8.11a). They also remain the same in the last iteration, when $k = 5$; no changes are introduced to the matrix because vertex e has no outgoing edges. A better path, one with a lower combined weight, is always chosen, if possible. For example, the direct one edge path from a to e in Figure 8.11d is shorter than the path found after the two edge path from b to e is found with a lower weight, as in Figure 8.11e.

This algorithm also allows us to detect cycles if the diagonal is initialized to ∞ and not to zero. If any of the diagonal values are changed, then the graph contains a cycle. Also, if an initial value of ∞ between two vertices in the matrix is not changed to a finite value, it is an indication that one vertex cannot be reached from another.

The simplicity of the algorithm is reflected in the ease with which its complexity can be computed. Since all three `for` loops are executed $|V|$ times, its complexity is $|V|^3$. This is a good efficiency for dense, nearly complete graphs, but in sparse graphs, where there is no need to check for all possible connections between vertices, for example, it may be more beneficial to use a one-to-all method $|V|$ times, that is, apply it to each vertex separately. This should be a label-setting algorithm, which as a rule has better complexity than a label-correcting algorithm. However, a label-setting algorithm cannot work with graphs with negative weights. To solve this problem, we have to modify the graph so that it does not have negative weights and it guarantees to have the same shortest paths as the original graph. Fortunately, such a modification is possible (Johnson and Sedgewick).

Observe first that, for any vertex v, the length of the shortest path to v is never greater than the length of the shortest path to any of its predecessors u plus the length of edge from u to v:

$$\text{dist}[v] \leq \text{dist}[u] + \text{weight}[\text{edge}(u, v)]$$

for any vertices u and v. This inequality is equivalent to the inequality

$$0 \leq \text{weight}[\text{edge}(u, v)] + \text{dist}[u] - \text{dist}[v]$$

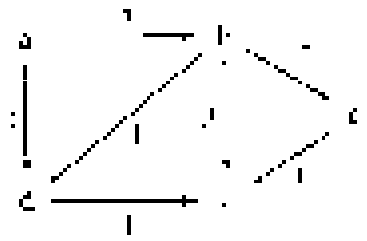
Hence, changing weights to $\text{weight}[\text{edge}(u, v)] + \text{dist}[u] - \text{dist}[v]$ for all edges renders a graph with nonnegative edge weights. Now note that the shortest path $v_1, v_2, \ldots, v_k$ is

$$\sum_{i=1}^{k-1} \text{weight}[\text{edge}(v_i, v_{i+1})] = \sum_{i=1}^{k-1} \text{weight}[\text{edge}(v_i, v_{i+1})] + \text{dist}[v_1] - \text{dist}[v_k]$$

Therefore, if the length L' of the path from v_1 to v_k is found in terms of nonnegative weights, then the length L of the same path in the sparse graph using the original weights, some possibly negative, is $L = L' - \text{dist}[v_1] + \text{dist}[v_k]$.

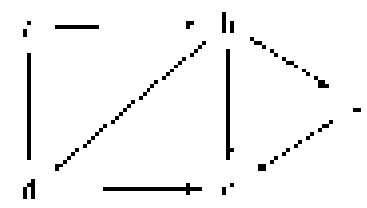
But, because the shortest paths have to be known to make such a transformation, the graph has to be preprocessed by one application of a label-correcting method. Only afterward are the weights modified, and then a label-setting method is applied $|V|$ times.

FIGURE 8.13 An execution of Algorithm Dijkstra.



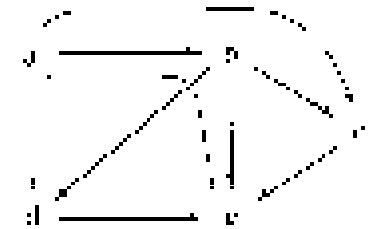
(a)

	a	b	c	d	e
a	0	2	1	4	∞
b	∞	3	2	5	7
c	∞	∞	3	4	6
d	∞	∞	∞	7	5
e	∞	∞	∞	∞	9



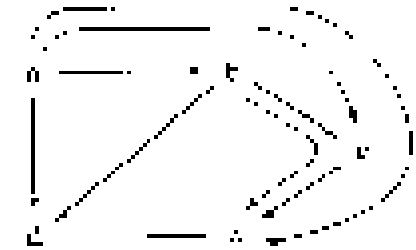
(b)

	a	b	c	d	e
a	0	2	1	4	∞
b	∞	3	2	5	7
c	∞	∞	3	4	6
d	∞	∞	∞	7	5
e	∞	∞	∞	∞	9



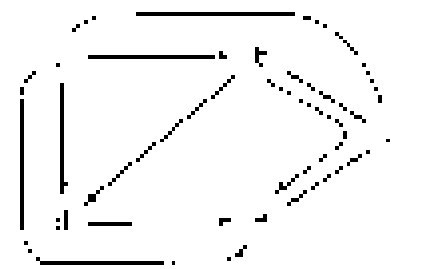
(c)

	a	b	c	d	e
a	0	2	1	4	∞
b	∞	3	2	5	7
c	∞	∞	3	4	6
d	∞	∞	∞	7	5
e	∞	∞	∞	∞	9



(d)

	a	b	c	d	e
a	0	2	1	4	∞
b	∞	3	2	5	7
c	∞	∞	3	4	6
d	∞	∞	∞	7	5
e	∞	∞	∞	∞	9



(e)

8.4 Cycle Detection

Many algorithms rely on detecting cycles in graphs. We saw in Section 8.3.1 how topological sorting algorithms fail if the graph contains a cycle. However, it is also sometimes desirable to explicitly find a cycle in a graph. Unfortunately, the depth-first search algorithm we developed in Section 8.3.2 is not sufficient. Therefore, a different cycle detection method is presented in this section.

The search algorithm is also implemented directly in `depthFirstSearch()`, because we need to add only one line in `DFS()` to detect cycles. Which one? Here is a simple hint:

```
type nTimeArrayDFS(n)
    num(n) = i++;
    for nWeight e in myGraph(v) do
        if num(w) = 0
            stack.push(w); to edges!
            cycleDetectionDFS(w);
        else if e.isBackEdge() then edges!
            cycleDetected!
```

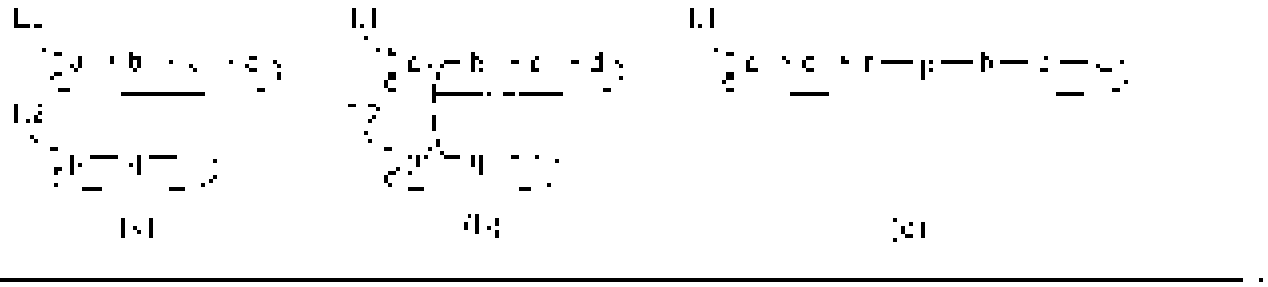
For digraphs, cycle detection is a bit more complicated, since there may be edges between different spanning subtrees called *side edges* (see edge g in Figure 8.3b). An edge leading to a cycle if it joins two vertices already included in the same spanning subtree. This is the case only if the assigned number is higher than any number generated in a recursive call whose box is assigned to a vertex being currently visited. After all the descendants have been visited, in this way a vertex is about to be finished by an edge with a vertex having a lower number, we declare a cycle detection. The algorithm is now

```
digraphCycleDetectionDFS(v);
    num(v) = i++
    for all nWeight w in myGraph(v) do
        if num(w) = 0
            stack.push(w); to edges;
            digraphCycleDetectionDFS(w);
        else if num(w) < num(v) then
            cycleDetected;
    num(v) = ∞
```

8.4.1 Union-Find Problem

Let us recall how the preceding section's depth-first search guaranteed generating a spanning tree in which no element of edges is used by `depthFirstSearch()` to form a cycle with all or some of the edges. This was due to the fact that if vertices would be joined by edges, then the edge was disregarded by `depthFirstSearch()`. A problem arises when `depthFirstSearch()` is modified so that it can detect whether a specific edge e is part of a cycle (see Exercise 20). Should such a modified depth-first search be applied to each edge separately, then one total run would be

FIGURE 8.12 Concatenating two circularly linked lists.



(a) (b) (c)

Gibbons [1985] gives an algorithm that solves the cycle detection problem. However, better methods are known.

The task is to determine if we can reach a node from itself by following pointers. A straightforward approach is to traverse the list, marking each node as it is visited. If we ever reach a node that is already marked, then we have found a cycle. Otherwise, we will eventually reach the end of the list. This is known as a *depth-first* algorithm.

The algorithm to solve the union-find problem can be improved with a technique called *path compression*. Suppose we have a node v deep in a tree, and we find the root r by following a sequence of k links. Path compression will change the parent pointers of all k nodes encountered on the path from v to r so that they point directly to r. Figure 8.13 shows an example of path compression.

Path compression makes the algorithm for `find` more expensive. In particular, `find` now makes two passes up the tree: one to find the root, and a second to update the parent pointers. Since `find` is now more than twice as expensive, it is not clear that path compression is a good idea if we are only doing a few `find` operations. However, if we are doing many `find` operations, then path compression is a very good idea, because most of the nodes we encounter will be pointing directly to the root.

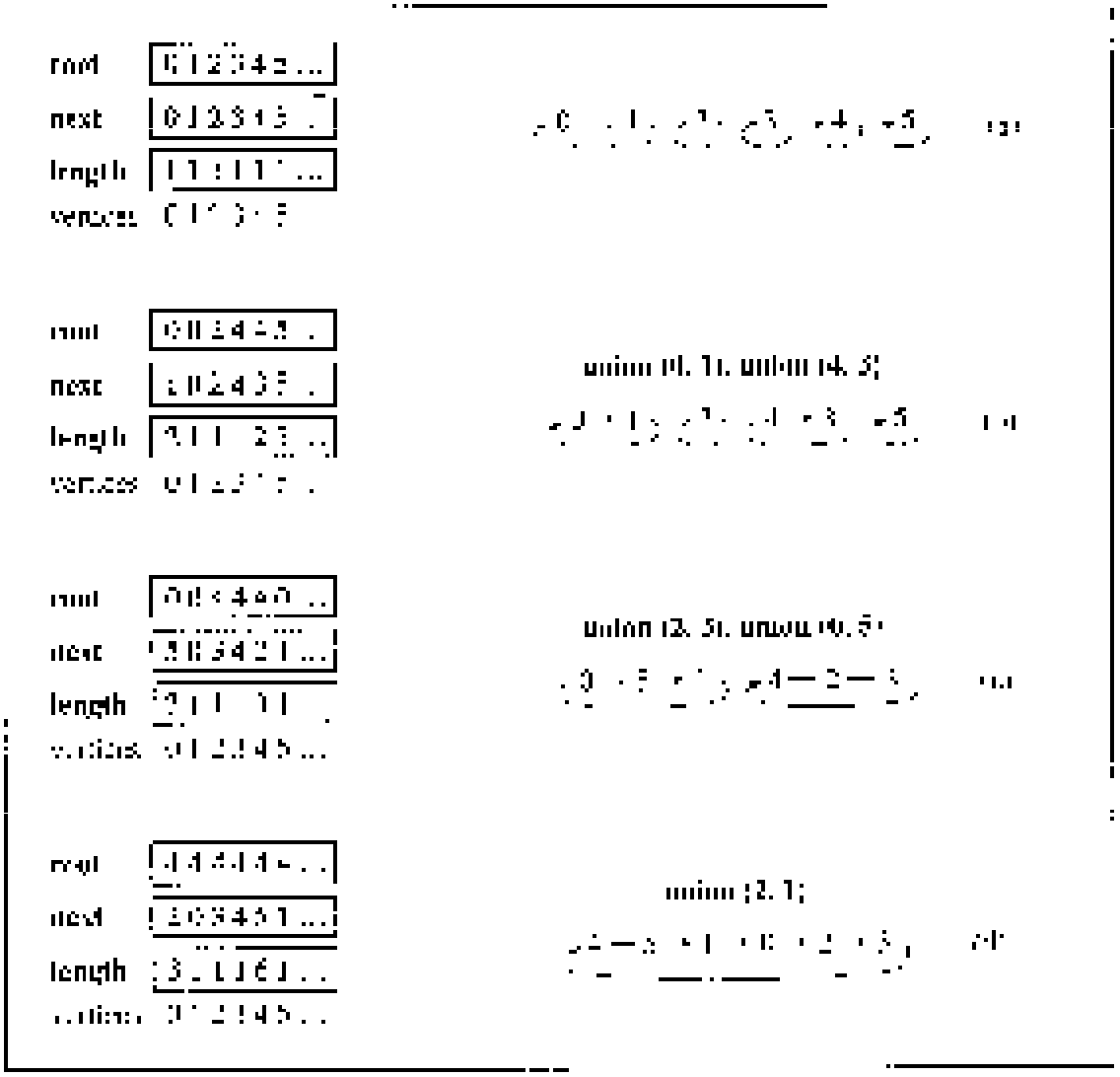
Path compression is particularly easy to code if we use recursion. Our implementation of `find` will first find the root. Then it will make a second recursive pass down the tree, changing the parent pointer of each node encountered to point to the root. The code is shown in Figure 8.14.

The `union` operation performs all the necessary tasks, so the `find` operation becomes trivial. Before we write the `union` function, we must decide what to do with the `size` array. Since we only care about whether the sizes are relatively large or small, we can afford to be sloppy and not update the `size` information for nodes that are no longer roots. This will cause the `size` information to be inaccurate, but we will see that we only need to know that a node was large at some point in the past, so this inaccuracy is not a problem. We will change the `size` array to store the natural logarithm of the number of nodes in the tree rooted at each node. This will require only changing the initialization in `main` to

```
for i = 0 to |V| - 1
    parent[i] = size[i] = 1
    height[i] = 1
```

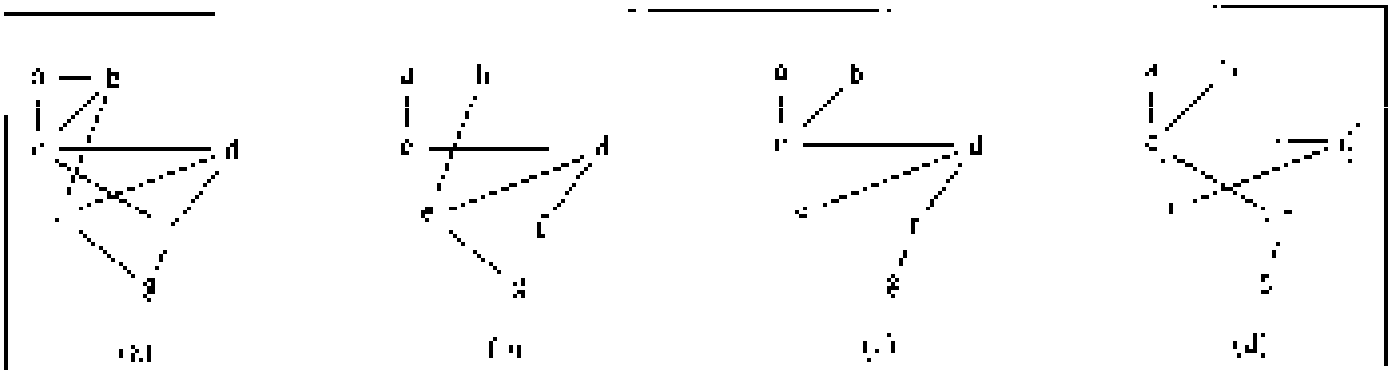
`union` is coded as follows:

FIGURE 8.13 An example of application of union() to merge lists.




```
union(edge(v,w));
   if (root[v] == root[w])                    // discarded this edge;
      return;                                 // since v and w are in
   else if (length[root[v]] < length[root[w]]) // the same set; combine
      v = root[v];                            // two sets into one;
      length[root[v]] += length[w];
      root[w] = root[v];                      // update root of root and
      for (i = next[w]; i != w; i = next[i]); // then update vertices
         root[i] = root[v];                   // in w's circular list;
      swap(next[v],next[root[w]]);            // merge two lists;
      add edge(v,w) to spanningTree;
   else // if length[root[v]] >= length[root[w]]
        // proceed as before, with v and w interchanged;
```

FIGURE 8.14 A graph representing the airline connections between seven cities and three possible sets of connections.



An example of the application of Kruskal's algorithm is shown in Figure 8.15. After initialization, there are 10 unary sets of vertex-node linked lists, as in Figure 8.13a. Edge (a, b) connects two of these sets, so their linked lists are merged into a larger one, as in Figure 8.13b. The new situation is pictured in the array as shown in Figure 8.15a.

The complexity of Kruskal's algorithm depends on the number of vertices that have to be updated when merging two sets, specifically, on the number of vertices on the shortest list, since that is the one for which we may have to do a union operation. Since this number can be between 1 and $|V|/2$, the complexity of merging is given by $O(|V|)$.

8.5 Spanning Trees

Consider the graph representing the airline connections between seven cities (Figure 8.14a). If the cost of maintaining flights is the same between any two cities, we want to keep as few connections as possible while still making sure that it is possible to reach any city from any other city. A set of connections is indicated by the graph in Figure 8.14b. City c is no longer reached from city d using the path d, c, but it is also possible to use the path d, b, a, c. Since the number of retained connections is the same, there is still the possibility we can reduce this number. It should be clear that the minimum number of such connections forms a tree because additional edges create cycles as a result of cycles in the graph. Hence, to ensure the minimum number of connections, a spanning tree should be created. And each spanning tree is the result of doing a depth-first search (or breadth-first search) on the graph. Clearly, we can create different spanning trees depending on which vertex we start the search from and which edges we traverse first. The airline uses the set of these connections to choose the best connections among the

optimum cost. This can be achieved by laying new roads only between the connections. This problem can now be phrased as finding a minimum spanning tree which is a spanning tree in which the sum of the weights of its edges is minimal. The previous problem of finding a spanning tree in an unweighted graph is a case of the minimum spanning tree problem in that the weights for each edge are assumed to be equal to one. Therefore each spanning tree is a minimum cost spanning tree.

The minimum spanning tree problem has many solutions and only a handful of them are presented here. For a review of these methods, see Graham and Hell 1985. These algorithms can be divided into the following categories:

1. Creating and expanding trees from one vertex to the entire set of vertices (Borůvka's algorithm).
2. Expanding a set of trees to form one spanning tree (Kruskal's algorithm).
3. Creating and expanding only one tree by adding new branches to it (Prim's algorithm).
4. Creating and expanding only one tree by adding new branches to it and possibly removing branches from it (Dijkstra's method).

8.5.1 Borůvka's Algorithm

Probably the first algorithm for finding the minimum spanning tree was devised in 1926 by Otakar Borůvka (pronounced: boh roof ka). In this method, we start with $|V|$ one-vertex trees and for each vertex in we look for an edge of minimum weight connecting all edges outgoing from it and create small trees by including these edges. Then we look for edges of minimal weight that can connect the resulting trees to larger trees. This process is finished when one tree is created. Here is a pseudocode for Borůvka's algorithm:

```
Borůvka's algorithm(weighted connected undirected graph)
    make each vertex into a one-vertex tree
    while there is more than one tree
        for each tree in the forest
            find a minimum-weight edge from a vertex in the tree to a vertex not in the tree
            create a new tree by combining the two trees and the edge
        replace the trees with the new tree
```

For example, in the graph in Figure 8.15a, out of seven one-vertex trees two trees are created connecting vertices a and c. Edge (a, c) is chosen for vertex a, edge (c, a) is chosen for vertex c, edge (b, c) is chosen for vertex b, edge (c, b) is chosen, and for vertex d edge (d, c) is chosen (Figure 8.15b). Afterward, for the two trees left, edge (b, c) is selected. Since it is the shortest edge that connects these two trees, we get a single spanning tree.

How many iterations are required? In each iteration of the while loop, each of the existing trees is joined with an edge to at least one tree. In the worst case, $|V|$ trees remain after the first iteration, one tree is generated. In the subsequent iteration, these $|V|/2$ trees are joined, and so on. In the worst case, $\log_2 |V|$ iterations are needed, where $\log_2$ is the binary logarithm of one vertex trees.

Jarník's method lends itself very nicely to parallel processing, since for each tree a minimal edge has to be found independently.

8.5.2 Kruskal's Algorithm

One popular algorithm was described by Joseph Kruskal. In this method, all edges are ordered by weight, and then each edge in this ordered sequence is checked to see whether it can be used to extend one of the trees under construction. It is added to the tree if no cycle is created by its addition. Kruskal's algorithm can be summarized as follows:

```
Kruskal's Algorithm for weighted connected undirected graphs
    trees = ∅;
    edges = sequence of all edges of graph sorted by weight;
    for (i = 1; i <= |V| and |trees| < |V| - 1; i++)
        if edge found adjoins no vertex in a cycle with edges in trees
            add edge to trees;
```

Figure 8.5.2 Kruskal's algorithm

The complexity of this algorithm is determined by the complexity of the sorting method employed, which for an efficient sorting is $O(|E| \log |E|)$, which dominates the complexity of the union and find used for cycle detection. If we use a heap to implement Kruskal's algorithm, then the for loop of `KruskalAlgorithm` becomes

```
for (i = 1; i <= |V| and |trees| < |V| - 1; i++)
    minEdge = edges[i];
```

Although this loop can be called as many as $|E|$ times, it is exited after one if the first test is always executed and it performs a sift, which is of complexity $O(\log |E|)$, only for $|V| - 1$ edges addition to trees. Hence the complexity of these operations is $O(|E| + (|V| - 1) \log |E|)$, which is $O(|E| \log |E|)$. Therefore, the complexity of Kruskal's algorithm is determined by the complexity of a sorting algorithm which is $O(|E| \log |E|)$, but $O(|E| \log |V|)$.

8.5.3 Jarník-Prim's Algorithm

Another algorithm was discovered by Vojtech Jarník (pronounced: yar-neek) in 1930 and later rediscovered by Robert Prim. In this method, all of the edges are also initially ordered, but a candidate for inclusion in the spanning tree is an edge which not only does not form a cycle in the tree, but also is incident to a vertex already in the tree:

```
Jarník-Prim's Algorithm for weighted connected undirected graphs
    trees = ∅;
    edges = sequence of all edges of graph sorted by weight;
    for (i = 1; i <= |V| - 1; i++)
        for (j = 1; j <= |E|; j++)
            if edge[j] adjoins no vertex in a cycle with edges in trees and
               is incident on a vertex in trees
                add edge[j] to trees;
                break;
```

Figure 8.15 A spanning tree of graph G found by Borůvka's algorithm, a tree found by Kruskal's algorithm, and a tree found by Prim's algorithm, each of which is also found with Dijkstra's method.

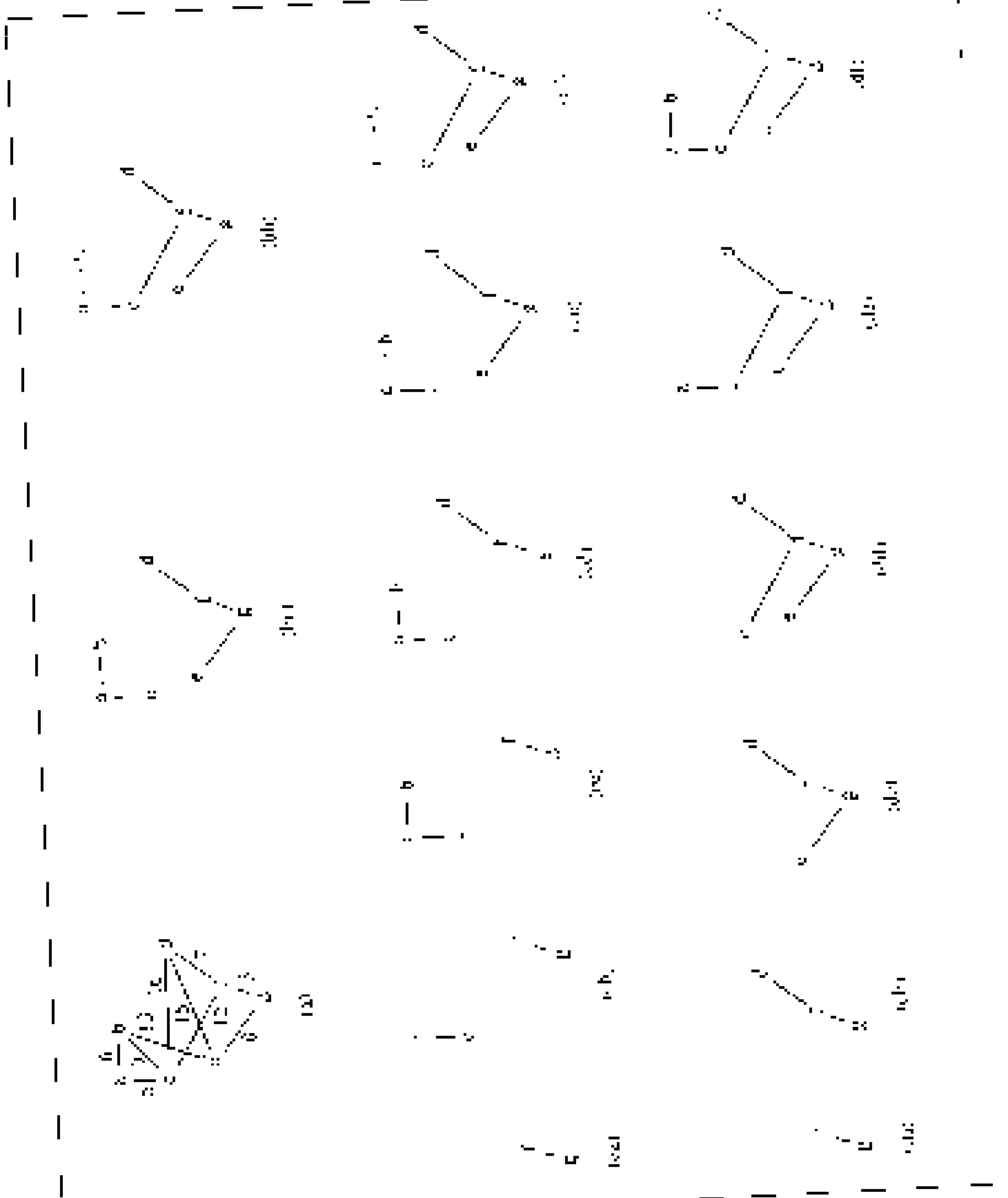
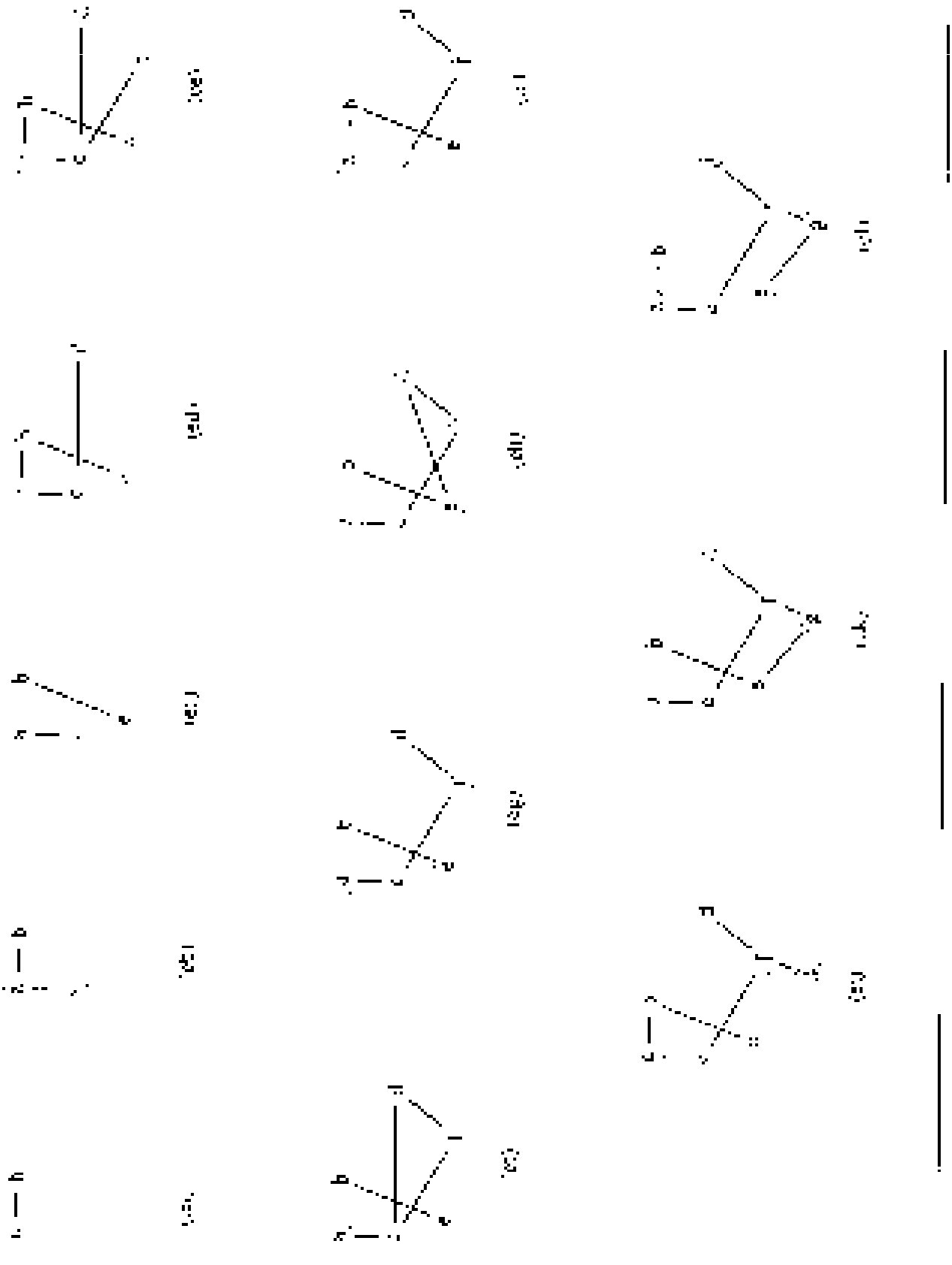


FIGURE 8.15 (continued)



Figures 8.15d–f contain a step-by-step example of Jarník-Prim's algorithm. The spanning tree resulting from this algorithm is the same as that produced by the Kruskal algorithm. However, the order in which edges have been added to the tree is different. The inner loop of `GenericSpanningTree()` can be $O(E)$ in the worst case, and since the outer loop iterates $|V| - 1$ times, the inner loop may execute $O(V \cdot E)$ times in total. However, this complexity can be substantially improved by a careful implementation of edges.

The difference between the Kruskal algorithm and the Jarník-Prim algorithm is that the latter always keeps the tree being constructed in one piece so that it is a tree at all stages of application of this algorithm. The Kruskal algorithm is more concerned about the outcome, so it considers it irrelevant that in the middle of its execution the spanning tree may not be a tree at all but a collection of trees. Nevertheless, the Kruskal algorithm guarantees that, in the end, there is only one spanning tree. Therefore, the Jarník-Prim algorithm may be considered more elegant as we have a tree being expanded at all times. The price for this elegance is that it's only edges that can be added to the tree are the ones which emanate from the tree built so far, so that certain edges may need to be reconsidered several times. In the Kruskal algorithm each edge needs to be considered only once, since if it leads to a cycle at one stage, it will then would lead to a cycle at a later stage, and hence, it does not have to be reconsidered anymore. Thus, the Kruskal algorithm is faster.

8.5.4 Dijkstra's Method

Kruskal's and Jarník-Prim's algorithms require that all the edges be ordered before beginning to build the spanning tree. This, however, is not necessary; it is possible to build a spanning tree by using any order of edges. A method was proposed by Dijkstra (1960) and independently by Robert Kruskal, and because no portion of an edge is required here, their method is more general than the other two.

```
Dijkstra's Method (weighted connected undirected graph)
    tree = null
    edges = an auxiliary structure of all edges of graph
    for i = 1 to |E|
        get e, a min edge in edges
        if there is a cycle in tree
            remove an edge with maximum weight from this cycle
```

In this algorithm, the tree is being expanded by adding to it edges one at a time, and if a cycle is detected, then an edge in this cycle with maximum weight is discarded. An example of building the minimum spanning tree with this method is shown in Figures 8.15a–c.

8.6 Connectivity

Many problems involve finding a path from one specified vertex to another or to any other vertex. For unweighted graphs, this means that the only criterion is the

or subgraphs of the graph for a digraph. We may find that we have several places in the graph to which we can get from some of the vertices but are not necessarily able to get to the starting vertex.

8.6.1 Connectivity in Undirected Graphs

An undirected graph is called connected when there is a path between any two vertices of the graph. The depth first search algorithm can be used for recognizing whether a graph is connected precisely that the loop heading

```
while there is a vertex v such that num[v] = 0
```

is terminated. Then, after the algorithm is finished, we have to check whether the list acyclist includes all vertices of the graph or simply check if t is equal to the number of vertices.

Connectivity comes in degrees. A graph can be more or less connected, and it depends on the number of different paths between its vertices. A graph is called *n*-connected if there are at least *n* different paths between any two vertices; that is, there are *n* paths between any two vertices that have no vertices in common. A special type of graph is a 2-connected or biconnected graph for which there are at least two different paths between any two vertices. A graph is not biconnected if a vertex can be found which always has to be included in the path between at least two vertices *a* and *b*. In other words, if this vertex is removed from the graph (along with incident edges), then there is no way to find a path from *a* to *b*, which means that the graph is split into two separate subgraphs. Such vertices are called *articulation points*, or *cut vertices*, and vertices *a* and *b* in Figure 8.15a are examples of articulation points. If an edge causes a graph to break down into two subgraphs, it is called a *bridge*. In the example, the edge (*a*, *b*) in Figure 8.15b is a bridge. The subgraphs that result from removing an articulation point or a bridge are called *blocks*, or *biconnected components*. It is important to know how to decompose a graph into biconnected components.

Articulation points can be detected by extending the depth first search algorithm. This algorithm generates a tree with forward edges (the graph edges included in the tree) and back edges (the edges not included). A vertex *v* in this tree is an articulation point if it has at least one subtree unconnected with any of its predecessors by a back edge, because it is then certainly none of the predecessors is reachable from any of its successors by a path. For example, the graph in Figure 8.16a is transformed into a depth first search tree (Figure 8.16b), and this tree has four articulation points, *A*, *d*, *b*, and *e*, since there is no back edge from any node below *d* to any node above it in the tree, *d* has back edges from any vertex in the right subtree of it to any vertex above it, but vertex *g* can be an articulation point because its successor *h* is connected to vertex above it. The four vertices divide the graph into the five blocks indicated in Figure 8.16c by dotted lines.

A special case for an articulation point is when a vertex is a root with more than one descendant in the tree. In Figure 8.16d, the vertex chosen as the root, *a*, has three incident edges, but only two of them have been followed (edges (*a*, *b*) and (*a*, *c*)), since the other was processed by depth-first search. Therefore, if the algorithm starts from a vertex previously processed, there is no unmarked edge. If we start from an articulation point, there

FIGURE 8.16 Finding blocks and articulation points using the DFS algorithm

would have exactly one child in the depth-first spanning tree. This indicates that v is an articulation point. The root is a special case that must be handled separately.

1. v is the root of the depth-first spanning tree and has more than one child, as discussed in this section.
2. v is not the root of T and some child of v cannot reach an ancestor of v by following zero or more tree edges and then one back edge (in this order).

To find articulation points, a parameter $\text{predt}[v]$ is used, defined as $\min(\{\text{numt}[v]\} \cup \{\text{numt}[p] \mid (v, p) \text{ is a back edge}\} \cup \{\text{predt}[c] \mid c \text{ is a child of } v\})$, where numt is as before. In other words, predt is the lowest number reachable from v by following zero or more tree edges and then one back edge (or by staying at v itself). Because the higher a predecessor of v is, the lower its number is, choosing a minimum number means choosing the highest predecessor. For the tree in Figure 8.16(a), $\text{predt}[2] = \text{predt}[3] = 1$, $\text{predt}[4] = 4$, and $\text{predt}[5] = 5$.

The algorithm uses a stack to store all currently processed edges. After an articulation point is identified, the edges corresponding to a block of the graph are popped. The algorithm is given as follows:

```
BlockDFS(v);
   predt[v] = numt[v] = t++;
   for all vertices u adjacent to v
      if edge (v, u) has not been traversed
         push edge (v, u);
      if numt[u] == 0
         BlockDFS(u);
         if predt[u] >= numt[v]                    // If there is no edge from u to a
            x = pop();                             // vertex above v, output a block
            while x != edge(v, u)                  // by popping all edges up to and
               output x;                           // including edge (v, u) from
               x = pop();                          // the stack;
            output x;                              // x == edge(v, u);
         else predt[v] = min(predt[v], predt[u]);  // Take a predecessor to go up an
      else if u is not the parent of v             // tree;
         predt[v] = min(predt[v], numt[u]);        // update when back edge (v, u) is
                                                   // traversed;
BlockSearch()
   for all vertices v
      numt[v] = 0;
   t = 1;
   while there is a vertex v with numt[v] == 0
      BlockDFS(v);
```

An example of the execution of this algorithm is shown in Figure 8.16. It is applied to the graph in Figure 8.16(a). The table lists all changes in predt for vertices processed by the algorithm, and the arrows show the evaluation of the new values of predt. For each vertex v, BlockDFS(v) first assigns two numbers: numt[v], shown in italics, and predt, which may change during the execution of BlockDFS(v). For example, 1 is processed first with numt[1] and predt[1] set to 1. The edge (1, 2) is pushed onto the stack, and since numt[2] = 0, the algorithm is invoked for 2. At this point, numt[2] and predt[2]

are set to 1. Next the algorithm is invoked for j's descendant of 2. Since dfnum[j] and predj[j] are set to 3, and then it is invoked for 4, a descendant of j. Since dfnum[j] is never and it is not j's parent, predj[j] is set to 1 = min(predj[j], dfnum[i]) = min(3, 1).

The algorithm also outputs the edges of biconnected blocks, and these edges are shown in Figure 8.6d at the moment they were output after popping them off the stack.

8.6.2 Connectivity in Directed Graphs

For directed graphs, connectedness can be defined in two ways depending on whether or not the direction of the edges is taken into account. A directed graph is weakly connected if the underlying undirected graph with the same vertices and the same edges is connected. A directed graph is strongly connected if for each pair of vertices there is a path between them in both directions. Every directed graph is not always strongly connected but it may be composed of strongly connected components (SCC), which are defined as subsets of vertices of the graph such that each pair of these vertices induces a strongly connected digraph.

To determine SCCs we also perform a depth-first search for vertices in the first vertex of an SCC for which depth-first search is applied. Such a vertex is called the root of the SCC. Since each vertex of the SCC is reachable from a root, all vertices and arcs after all such vertices have been visited, the depth-first search backtracks to a root. This case, which is recognized by the fact that predj[v] = num[v], the SCC accessible from the root can be output.

The problem now is how to find all such roots of the graph, which is analogous to finding articulation points in undirected graphs. To that end, the parent predj is also used, where predj[v] is the lowest number reachable from v and predj[v] whereas articulation points from undirected graphs use lowpt. However, how can we determine whether two vertices belong to the same SCC when SCC has not been determined? The approach can be solved by using a stack for the vertices belonging to the SCCs under construction. Removing vertices on the stack belongs to the currently analyzed SCC. Although construction is not finished, we already know which vertices are already included in the SCC. The algorithm in C++ is as follows.

```
void StrongDFS(int v)
  predv = num[v] = ++dfn;
  push(v);
  for (adjacent vertex w of v)
    if (num[w] == 0)
      StrongDFS(w);
      predv = min(predv, predw);                  // Take a predecessor higher up in
    else if (num[w] < num[v] && !finished[w])     // tree; update if back edge found
      predv = min(predv, num[w]);                 // to vertex w in the current SCC
  if (predv == num[v])                            // If the root of a SCC is found,
    w = pop();                                    // output this SCC, i.e.,
    while (w != v)                                // pop all vertices of this SCC
      output w;                                   // until v is popped off;
      w = pop();
    output w;                                     // w == v;
```

FIGURE 8.17 Finding strongly connected components with DEPTH-FIRST SEARCH.

```
StronglyConnectedComponentSearch()
  for all vertices v
    num[v] = 0
  i = 1;
  while there exists a vertex v such that num[v] == 0
    strongDFS(v)
```

Figure 8.17 contains an example of execution of Tarjan's algorithm. The digraph in Figure 8.17a is processed by a series of calls to strongDFS(), which assigns to vertices v through K the numbers shown in parentheses in Figure 8.17b. During this process, five SCCs are detected and printed. Figure 8.17c contains the depth-first search tree created by this process. Note that two trees are created, so that the number of trees does not have to correspond to the number of SCCs, as the number of trees did not correspond to the number of blocks in the case of articulation points. Figure 8.17d indicates, for each vertex, numbers assigned to vertices and all changes of parameter pred[v] for all vertices v in the graph. It also shows the SCCs output during the processing of the graph.

8.7 Topological Sort

In many situations, there is a set of tasks to be performed. For some pairs of tasks, it matters which task is performed first, whereas for other pairs, the order of execution is unimportant. For example, students need to take into consideration which courses are prerequisites or corequisites for other courses when making a schedule for the upcoming semester, so that Computer Programming II cannot be taken before Computer Programming I, but the former can be taken alongside, or before, Ethics or Introduction to Sociology.

Precedence dependencies between tasks can be shown in the form of a digraph. A topological sort of a digraph G with n vertices assigns numbers $1, \ldots, n$ to the vertices so that $v < w$ only if there is a path from vertex v to vertex w. The digraph must not include a cycle; otherwise, a topological sort is impossible.

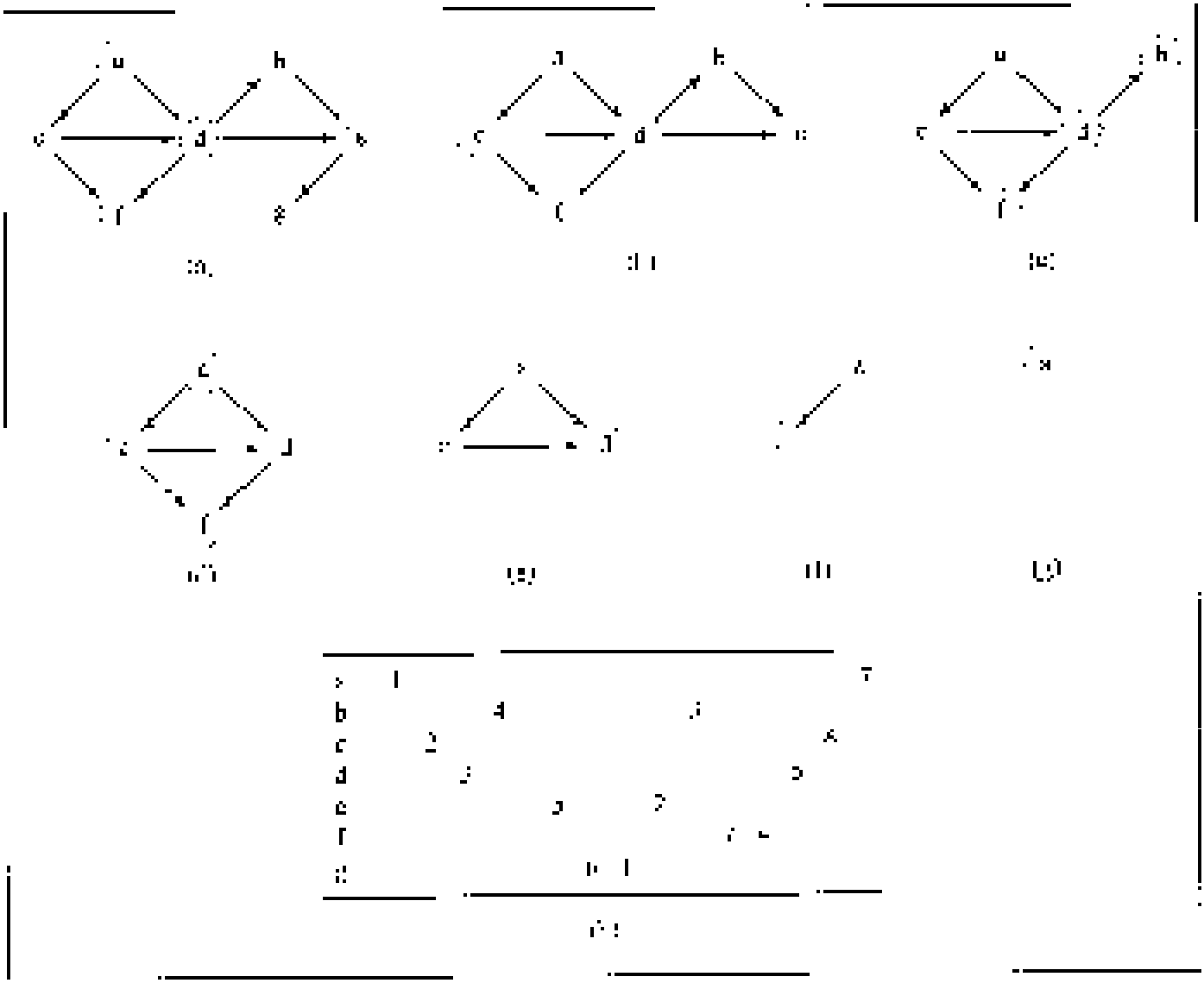
The algorithm for a topological sort is rather simple. We have to find a vertex with no outgoing edges, called a *sink* or a *terminal vertex*, and then disregard all edges leading from any vertex to it. The summary of the topological sort algorithm is as follows:

```
topologicalSort(digraph)
    for i = 1 to |v|
        find a minimal vertex v;
        order[v] = i;
        remove from digraph vertex v and all edges leading to v;
```

Figure 8.18 contains a more detailed pseudocode version of this algorithm. The graph in Figure 8.18a undergoes several modifications (see Figures 8.18b–f) and results in these numbers, as in Figure 8.18g.

Actually, we do not have to remove the vertices and edges from the digraph when it is processed; we just need to logically mark the vertices as deleted and once again, depth-first search comes to the rescue by the way of this method. The

Figure 8.18 Executing topological sort.



search on it only and no topological sort.

```
visit(v)
    num[v] = cnt++;
    for all vertices w adjacent to v
        if num[w] == 0
            visit(w);
        else if TSNum[w] == 0
            cycle            // a cycle is detected!
    TSNum[v] = tsn++;        // after processing all successors of v,
                             // assign to v a number larger than
                             // assigned to any of its successors;
```

```
Locate a vertex with a zero indegree;
   For all vertices v
      Assign v's indegree to indegree[v];
   Counter = 1;
   While there's a vertex v such that indegree[v] = 0
      Store v;
   Output vertices according to their topological numbers;
```

The table in Figure 8.41 indicates the order in which this algorithm assigns numbers to the nine vertices and the topological number for each vertex of the graph in Figure 8.40.

8.8 Networks

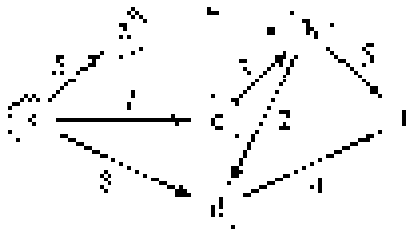
8.8.1 Maximum Flows

An important type of problem involves a network of pipes. A network is a system of pipes connected to deliver water from one point to another. However, we are not simply interested in how pipes, but through which pipes we have to pump water to maximize the amount of water that can be pumped from one point to another. For example, the network in Figure 8.42 has eight pipes and six pumping stations. The problem is to find the maximum amount of water that can be pumped from point s to point t. For example, the pipes going out of s have a capacity of 5 million gallons per hour. The problem is to find the capacity of the entire network so that we can transfer the maximum amount of water. This type of problem is known as a maximum flow problem. Notice that the pipe connecting u and v is bidirectional. It has only one pipe, but it has a capacity of 4. This means that we can pump 3 units through pipe vu since pipe uv cannot have a flow of more than 1 unit. Similarly, pipes uv and vu can be combined as well as pipes xy and yx. Determining pipes, although they are used to find a maximum flow, can be an important problem. This is far from obvious, especially for large networks, what the maximum flow is through each pipe. One of the most basic analyses of this problem for networks was initiated by Lester R. Ford and D. Ray Fulkerson. Since their work, numerous algorithms have been published to solve this problem.

To solve the problem as stated more formally, we would like to give some definitions. A network is a digraph with one vertex s called the source, with no incoming edges, and one vertex t called the sink with no outgoing edges. These definitions are chosen for their intuitiveness; however, in a more general case, both source and sink can be any two vertices. With each edge e we associate a number $c(e)$ called the capacity of the edge. A flow is a real function $f: E \rightarrow R$ that assigns a number to each edge of the network and meets these two conditions:

1. The flow through an edge e cannot be greater than its capacity or $0 \le f(e) \le c(e)$. ($f(e)$ is the capacity constraint.)

FIGURE 8.19 A pipeline with eight pipes and six pumping stations.



2. The total flow entering a vertex v is the same as the total flow emerging from v; that is, $\sum_u f(u, v) = \sum_w f(v, w)$, where we set $f(u, v) = 0$ if (u, v) is not an edge of the network.

The problem now is to maximize the flow f so that the sum $\sum_u f(u, t)$ has a maximum value for any possible function f. This is called the **maximum-flow minimum-cut problem**.

An important concept used in the Ford-Fulkerson algorithm is the concept of a cut. A **cut** separates s and t by a set of edges between vertices of s and vertices of t. Any vertex of the graph belongs to one of these two sets. A cut is denoted by $(S, \bar{S})$, where S is the set of vertices containing s and $\bar{S}$ is the set of vertices containing t. For example, in Figure 8.19, if $S = \{s, a\}$, then $\bar{S} = \{b, c, d, t\}$. The set of edges $\{(a, b), (a, c), (s, d)\}$ is a cut. This means that if all edges in this set are cut, then there is no way to get from s to t. Let us define the **capacity** of the cut to be the sum of capacities of all the edges leading from a vertex in S to a vertex in $\bar{S}$; thus, $\text{cap}(a, b) + \text{cap}(a, c) + \text{cap}(s, d) = \text{cap}(s, d) = 15$. Now f should be less than or equal to the flow through the network cannot be greater than the capacity of any cut. Intuitively, this leads to the **max-flow min-cut theorem** (Ford and Fulkerson 1956):

Theorem. In a network, the maximal flow from s to t is equal to the minimum capacity of any cut.

This theorem states what is expressed in the simile of a chain in that a system is only as strong as its weakest link. Although there may be cuts with great capacity, it is enough that one small cut limits the flow of the network. For example, a simple flow in the network in Figure 8.19 is $f(s, a) = 5$, $f(a, b) = 3$, $f(b, t) = 3$, $f(a, c) = 2$, and $f(c, t) = 2$. Now we have to find a cut which has the smallest capacity among all possible cuts. A natural candidate is the cut through each edge of this cut as many units as the capacity allows, so that a maximum flow is used.

A **flow-augmenting path** from s to t is a sequence of edges from s to t such that, for each edge in this path, the flow $f(e)$ is less than capacity on forward edges and $f(e) > 0$ on backward edges. It means that such a path is not optimally used yet, and it can transfer more units from s to t by changing it. If the flow on at least one edge of the path is equal to the capacity, then obviously the flow cannot be augmented, but for the path there can be no edges but only forward edges so that capacities of paths in flow will be augmented by a positive quantity. Backward edges are what they are, backwards. For example, in

source to sink. Depth-first processing for flows can be very inefficient. Then the overall flow in the network can be increased. Hence, the process of augmenting flows of paths is not finished until the flow for such a path is zero. The task now is to find an augmenting path if it exists. Since there may be very large numbers of paths from source to sink, finding an augmenting path is a tractable problem, and Ford and Fulkerson (1957) developed the first algorithm to accomplish it in a systematic manner.

The labeling phase of the algorithm consists of assigning to each vertex v a label which is the pair

$$\text{label}(v) = (\text{parent}(v), \text{slack}(v))$$

where parent(v) is the vertex from which v is being accessed and slack(v) is the amount of flow that can be transferred from s to v. The forward and backward edges are treated differently. If a vertex u is accessed from v along a forward edge then

$$\text{label}(u) = (v, \min(\text{slack}(v), \text{slack}(v,u)))$$

where

$$\text{slack}(v,u) = \text{capacity}(v,u) - \text{flow}(v,u)$$

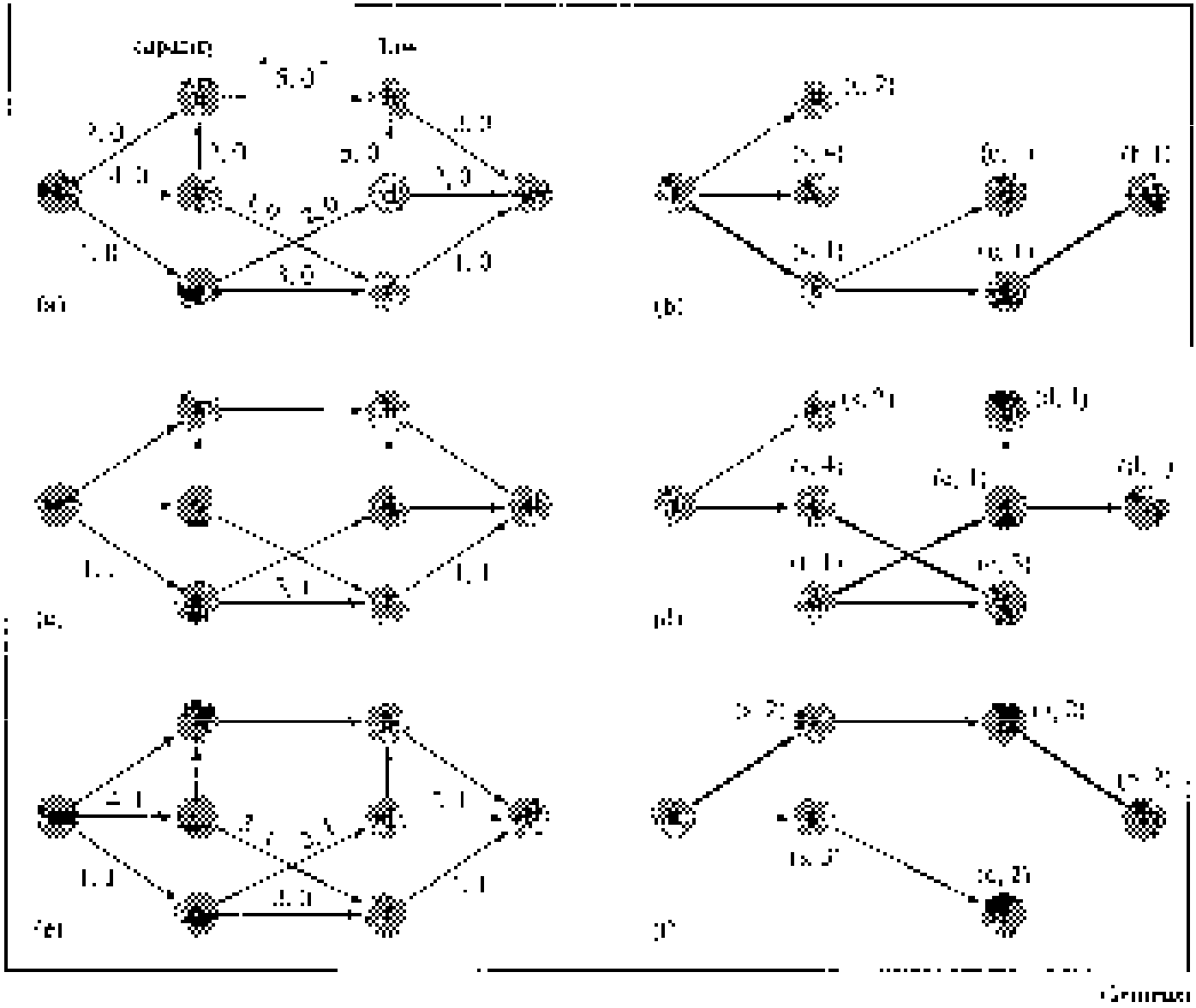
which is the difference between the capacity of edge (v,u) and the amount of flow currently caused by this edge. If the edge from v to u is a backward edge (i.e., from u to v), then

$$\text{label}(u) = (v, \min(\text{slack}(v), \text{flow}(u,v)))$$

After a vertex is labeled, it is stored for later processing. Until a process on the edge (v,u) is started, which allows for some more flow to be added. For forward edges, this is possible when slack(v,u) > 0, and for backward edges when flow(u,v) > 0. However, labeling does not guarantee that a path exists. The process is finished if we arrive at the sink in the middle of the network search to label any more edges. If we reach the sink, the flows of the edges on the augmenting path can now finally be updated by increasing flows of forward edges and decreasing flows of backward edges, and the process restarts in the quest for another augmenting path. Here are a number of key points:

```
augmentPath(network with source s and sink t)
   for each edge e in the path from s to t
      if (forward(e))
         flow(e) += slack(t);
      else flow(e) -= slack(t);

FordFulkersonAlgorithm(network with source s and sink t)
   set flow of all edges and vertices to 0;
   label(s) = (null, ∞);
   labeled = {s};
   while labeled is not empty // while not stuck;
      remove vertex v from labeled;
      for all unlabeled vertices u adjacent to v
         if forward(edge(v,u)) and slack(edge(v,u)) > 0
            label(u) = (v, min(slack(v), slack(edge(v,u))));
         else if backward(edge(v,u)) and flow(edge(u,v)) > 0
```

FIGURE 8.20 An execution of breadth-first labeling algorithm using capacity-1 edges.

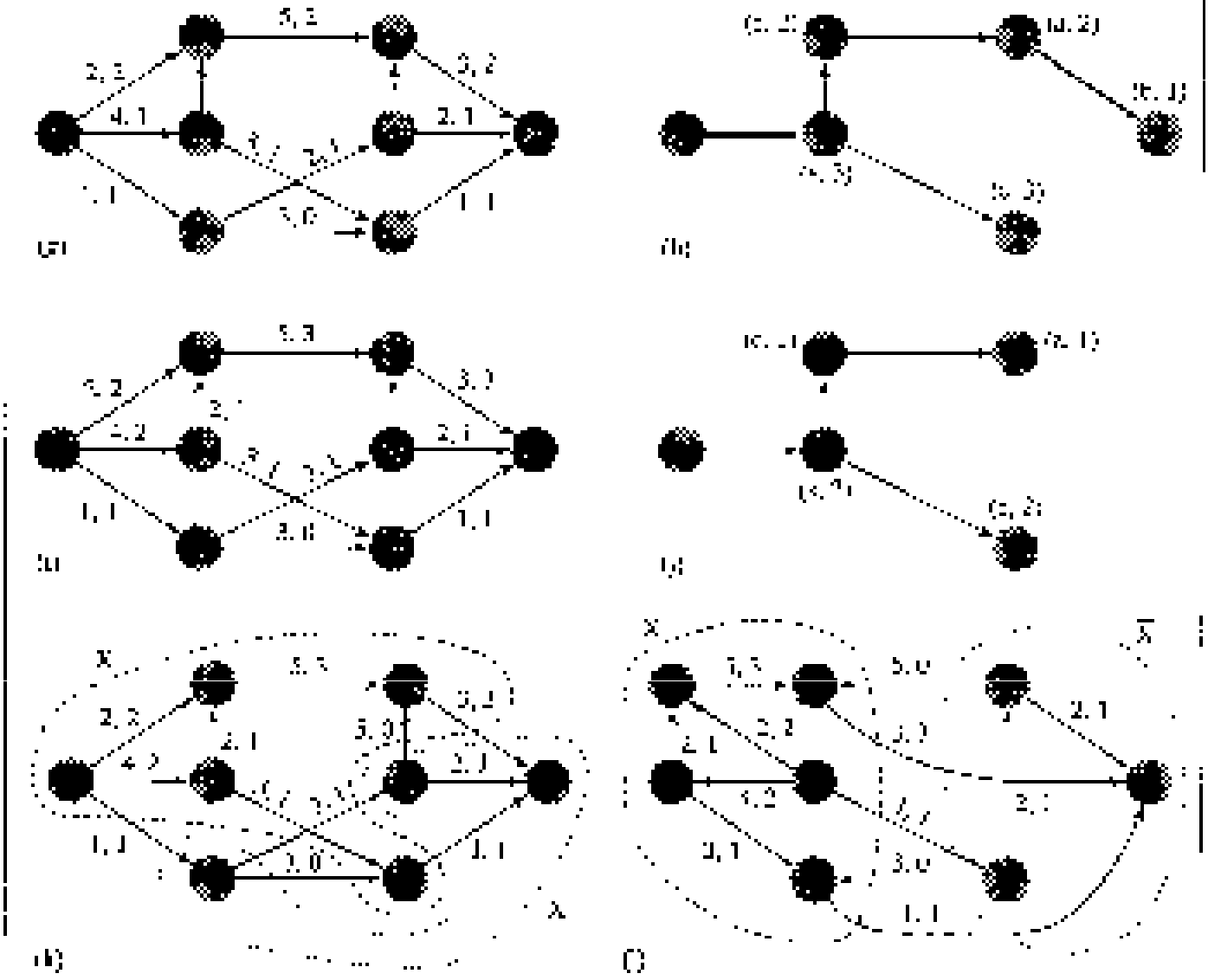



```
    labels[y] = (v, min(slack(v,y), flow(v,y)));
// y gets labeled
    if (y == t)
        augmentPath(labels, source);
        labeled = {s};    // Clear for another pass;
    else mark y in labeled;
```

Notice that this algorithm is nondeterministic with respect to the order in which the network should be searched. In exactly what order should vertices be included in *labeled* and dequeued from it? This question is left unanswered as we choose to avoid getting involved in the details of these operations that are processing the network in a depth-first fashion.

Figure 8.20 illustrates an example. Each edge is represented by two numbers associated with it, the capacity and the current flow, and initially the flow is set to zero for each edge (Figure 8.20(a)). We begin by putting the vertex *s* in *labeled*. In the first iteration of the

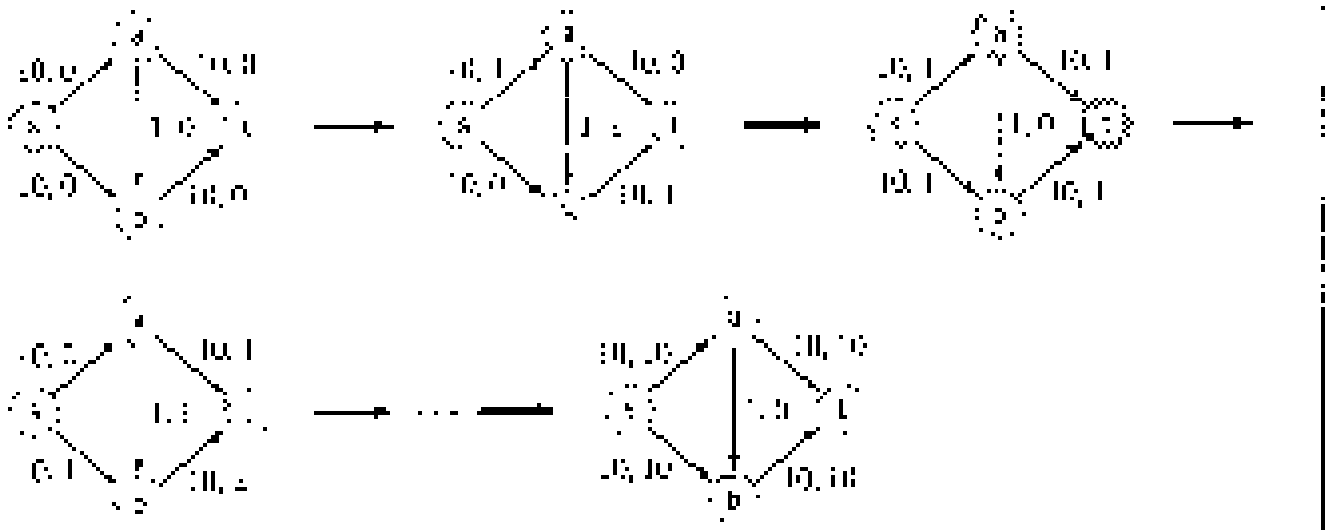
FIGURE 8.20 (continued)



which the loop, v is dequeued from labeled, and in the next loop below (p.3) it is compared to the first adjacent vertex, w, labeled f(w) to vertex v and label p(w) to vertex v (Figure 8.20b), and all three vertices are pushed onto labeled. The for loop is exited, and since labeled is not empty, the while loop begins its second iteration. In this iteration, a vertex is popped off from labeled, which is w, and both adjacent vertices of vertex w—vertices y and t—are labeled and pushed onto labeled. Now, the third iteration of the while loop begins by popping t from labeled and labeling its only unlabeled neighbor, vertex x. Because t is the sink, the flows of all edges on the augmenting path s-w-t-x are updated in the inner for loop (Figure 8.20c), labeled is reinitialized to {s}, and the next round begins to find another augmenting path.

The next round starts with the fourth iteration of the while loop. In its eighth iteration, the sink is reached (Figure 8.20d) and flows of edges on the new augment-

FIGURE 8.21 An example of an implementation of breadth-first search



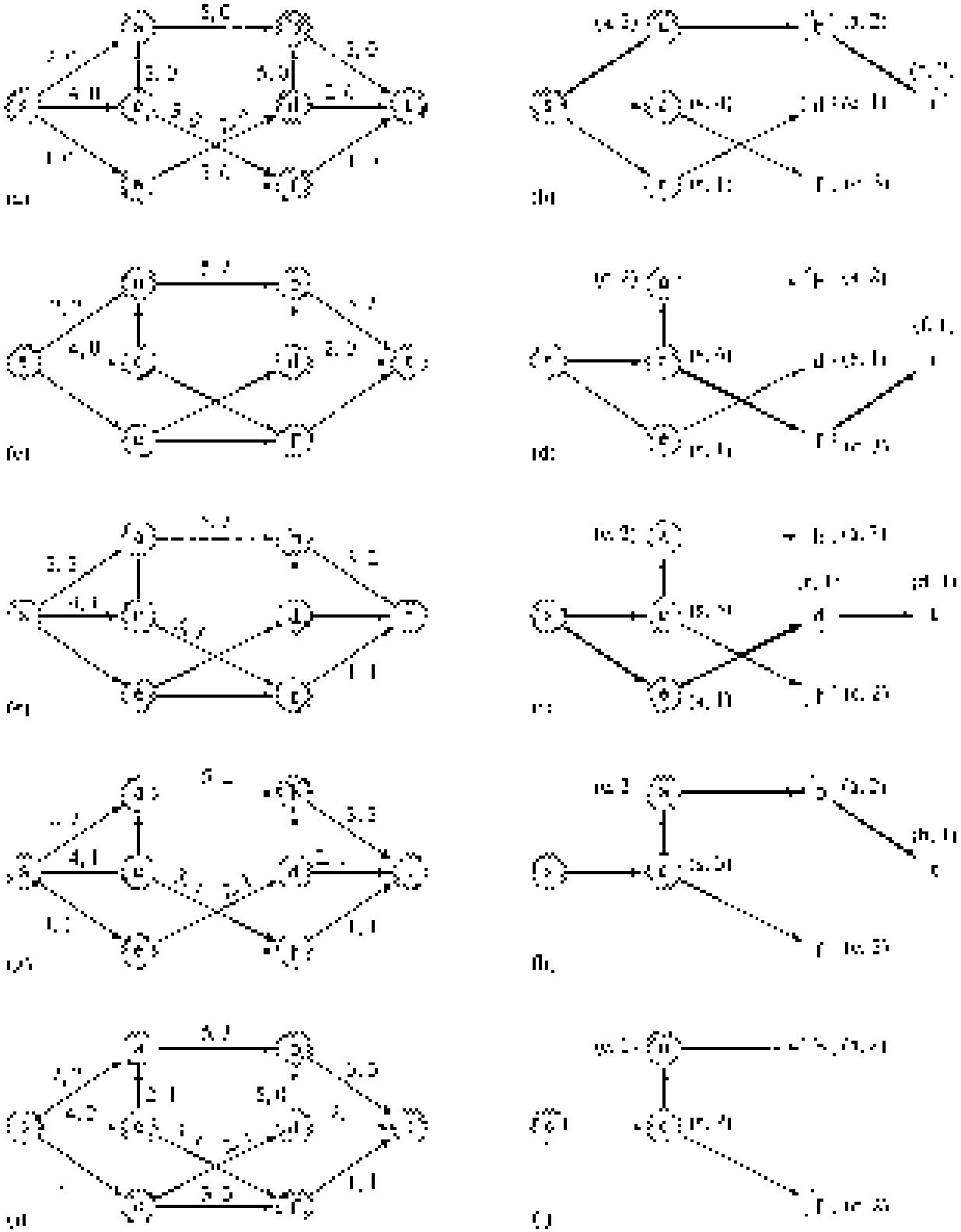
top, which are labeled (Figure 8.20(e)). Note that this time one edge, (C, t), is a backward edge. Therefore, its flow is decremented, not incremented, as is the case for forward edges. Afterward, two more augmenting paths are found, and the corresponding edges are updated. In the last round, we are unable to reach the sink (Figure 8.21(j)), which means that all augmenting edges have been found and the maximum flow has been computed.

After finishing execution of the algorithm all vertices labeled in the last round, including t, are in the set X and the unlabeled vertices in the set $\overline{X}$, that we have already seen in Figure 8.20(d). For clarity, both sets are shown in Figure 8.21(k). Note that all the edges from X to $\overline{X}$ are used in full capacity and all the edges from $\overline{X}$ to X carry no flow at all.

The complexity of this implementation of the algorithm is not necessarily a function of the number of vertices and edges in the network. Consider the network in Figure 8.21(a). During the first implementation we could choose the augmenting path s, b, t with flows of all three edges set to 1. The next augmenting path could be s, a, t with flows of two forward edges set to 1 and the flow of the backward edge (b, a) reset to 0. Next time the augmenting path could be the same as the first, with flows of two edges set to 2 and with the vertical edge set to 1. It is clear that an augmenting path would be chosen $2 \cdot 10^6$ times, although there are only four vertices in the network.

The problem with the breadth-first search algorithm is that it uses the depth-first approach when searching for an augmenting path. But as already mentioned, this choice does not stem from the nature of this algorithm. The depth-first approach attempts to reach the sink as soon as possible. However, trying to find the shortest augmenting path gives better results. This leads to a breadth-first approach (Edmonds and Karp 1972). The breadth first processing uses the same procedure as Breadth-first search (Figure 8.13), except that this time label is a queue. Figure 8.22 illustrates such an example.

FIGURE 8.22 An execution of FordFulkersonAlgorithm() using breadth-first search



To determine one single augmenting path, the algorithm requires at most $2|E|$ or $O(|E|)$ steps to check both sides of each edge. The shortest augmenting path in the network can have only one edge and the longest path can have at most $|V| - 1$ edges. Therefore, there can be augmenting paths of lengths $1, 2, \ldots, |V| - 1$. The number of augmenting paths of a certain length is at most $|E|$. Therefore, to find all augmenting paths of all possible lengths, the algorithm needs to perform $O(|V||E|)$ steps. And since finding one such path is of order $O(|E|)$, the algorithm is of order $O(|V||E|^2)$.

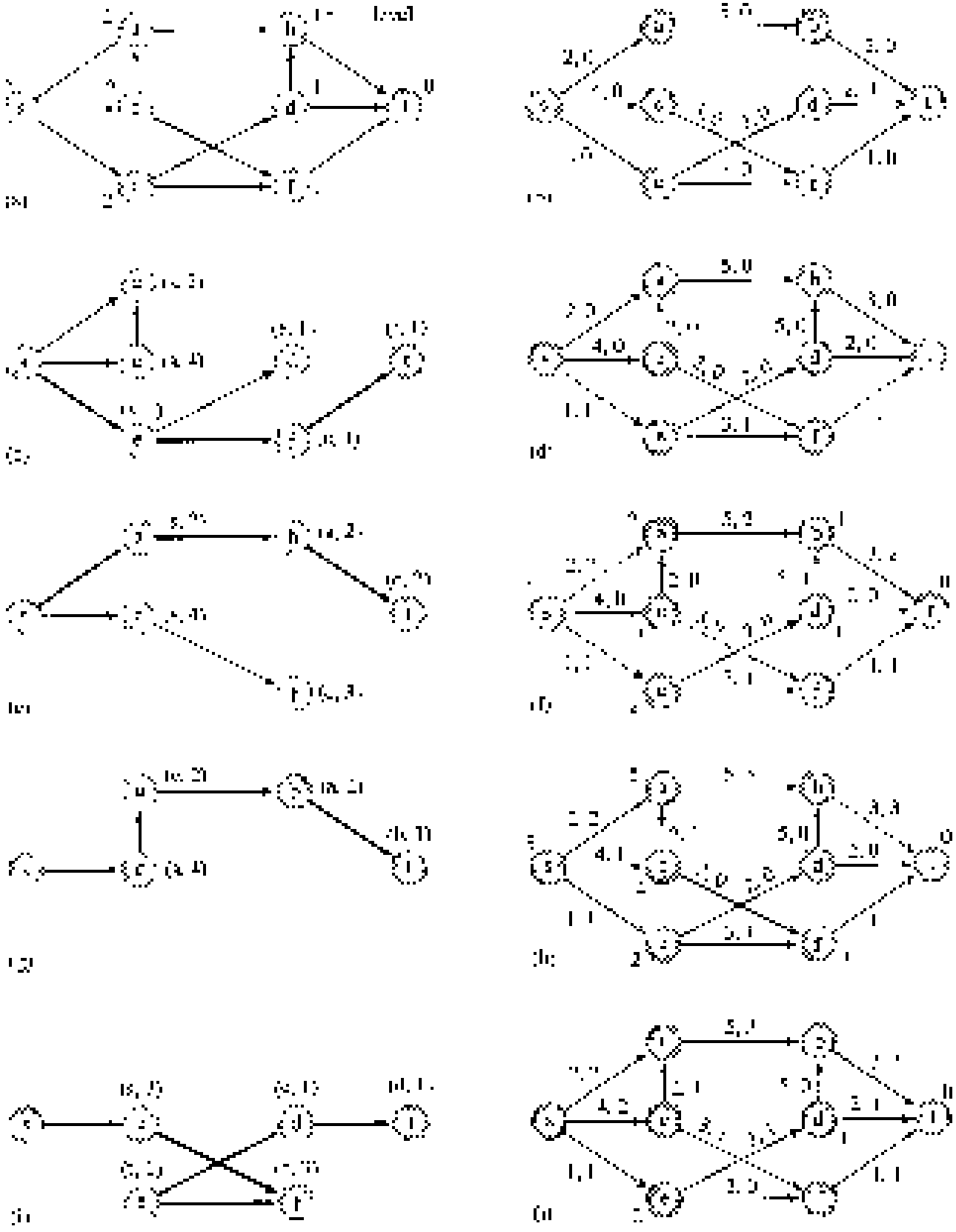
Although the pure breadth-first search approach is better than the pure depth-first search implementation, it still is far from ideal. We will not find many occurrences of augmenting steps anymore, but there still seems to be a great deal of wasted effort. In breadth-first search, a large number of vertices are labeled to find the shortest path (shortest in a given iteration). Then all these labels are discarded to be recreated when looking for another augmenting path (edges ad, ab, and ae in Figure 8.22b–c). Therefore, it is desirable to reduce this redundancy. Also, there is some merit to using the depth first approach in that it attempts to aim at the goal, the sink, without expanding a number of paths at the same time and finally choosing only one and discarding the rest. Hence, the Solomonic solution appears to use both approaches, depth first and breadth first. Breadth first search prepares the ground to prevent loops of small increments from happening (as in Figure 8.21) and to guarantee that depth first search takes the shortest route. Only afterward, the depth-first search is launched to find the sink by aiming right at it. An algorithm based upon this principle was devised first by Efim A. Dinic (pronounced: dee neetz).

In Dinic's algorithm, up to $|V| - 1$ passes (or phases) through the network are performed, and in each pass, all augmenting paths of the same length from the source to the sink are determined. Then, only some or all of these paths are augmented.

All augmenting paths form a *layered network* (also called a *level network*). Constructing layered networks from the underlying network starts from the lowest values. First, a layered network of a path of length one is found, if such a network exists. After the network is processed, a layered network of paths of length two is determined, if it exists, and so on. For example, the layered network with the shortest paths corresponding with the network in Figure 8.22a is shown in Figure 8.22b. In this network, all augmenting paths are of length three. A layered network with a single path of length one and layered networks with paths of length two do not exist. The layered network is created using breadth first processing, and only forward edges that can carry more flow and backward edges that already carry some flow are included. Otherwise, even if an edge may lay on a short path from the source to the sink, it is not included. Note that the layered network is determined by breadth first search that begins in the sink and ends in the source.

Now, since all the paths in a layered network are of the same length, it is possible to avoid redundant tests of edges that are part of augmenting paths. If in a current layered network there is no way to go from a vertex v to any of its neighbors, then in later tests in the same layered network there will be the same situation; hence, checking again all neighbors of v is not needed. Therefore, if such a dead end vertex v is detected, all edges incident with v are marked as blocked so that there is no possibility to get to v from any direction. Also, all saturated edges are considered blocked. All blocked edges are shown in dashed lines in Figure 8.23.

FIGURE 8.23 An execution of Dinic's algorithm



After a layered network is determined, the depth-first process finds as many augmenting paths as possible. Since all paths are of the same length, depth-first search does not go to the sink through some longer sequence of edges. After one such path is found, it is augmented and another augmenting path of the same length is looked for. For each such path, at least one edge becomes saturated so that eventually no augmenting path can be found. For example, in the layered network in Figure 8.33 (first redrawn in Figure 8.35a), two augmenting paths (using edges labeled *a* and *b* in Figure 8.35b) are found, and all edges are augmented (Figure 8.35c). Then only one more three-edge path is found, the path *e*, *d*, *h* (Figure 8.35d), since, for example, previous augmentation saturated edge *f* so that the partial path *a*, *f* leads with a dead end. In addition, because no other vertex can be reached from *f*, all edges incident with *f* are blocked (Figure 8.35f) so that an attempt to find the third four-edge augmenting path only tests vertex *a*, but never vertex *f* since edge *f* is blocked.

If no more augmenting paths can be found, a higher level layered network is found, and augmenting paths for this network are searched for. The process stops when no layered network can be formed. For example, one of the networks in Figure 8.33f, the layered network in Figure 8.35g is formed, which has only one four-edge path. In fact, this is the only augmenting path for this network. After augmenting this path, the situation in the network is as in Figure 8.35h, and the final layered network is formed, which also contains only one path, that is, one path of five edges. The path is augmented (Figure 8.35i), and then no other layered networks can be found. This algorithm can be summarized in the following pseudocode.

```
LayeredNetwork(network with source s and sink t)
    for all vertices v
        level[v] = -1;
    level[s] = 0;
    enqueue(s);
    while (queue is not empty)
        v = dequeue();
        for all vertices u adjacent to v such that level[u] == -1
            if (forward edge (v, u) and stack(edge(v, u)) > 0 or
                backward edge (u, v) and flow(edge(u, v)) > 0)
                level[u] = level[v] + 1;
                enqueue(u);
            if u == t
                return success;
    return failure;

ProcessAugmentingPaths(network with source s and sink t)
    mark all edges;
    labeledges = [s];
    while (labeledges is not empty // while more vertices
        pull v from labeledges;
        for all unmarked edges (v, w) adjacent to v such that
            edge(v, w) is not blocked and level[w] == level[v] + 1
            if (forward edge(v, w) and flow(edge(v, w)) < 0
```

```
            Label(w) = (v', max(flow(v), slack(edge(v, w))));
        else if (backward(edge(v, w)) && flow(edge(w, v)) > 0)
            Label(w) = (v', max(flow(v), flow(edge(w, v))));
        if w got labeled
            if w == t
                augmentPath();
                Mark all nodes unlabeled;
                Label t with (p);   // Look for another path;
            else push w onto the queue;
    if no neighbors of v was newly labeled
        Mark all edges leaving v as dead;

void FordFulkerson(network, source s, sink t)
    set flows of all edges and reverse edges to 0;
    Label(s) = (null, ∞);
    while (LayerNetwork(network) returns true)
        processAugmentingPath(network);
```

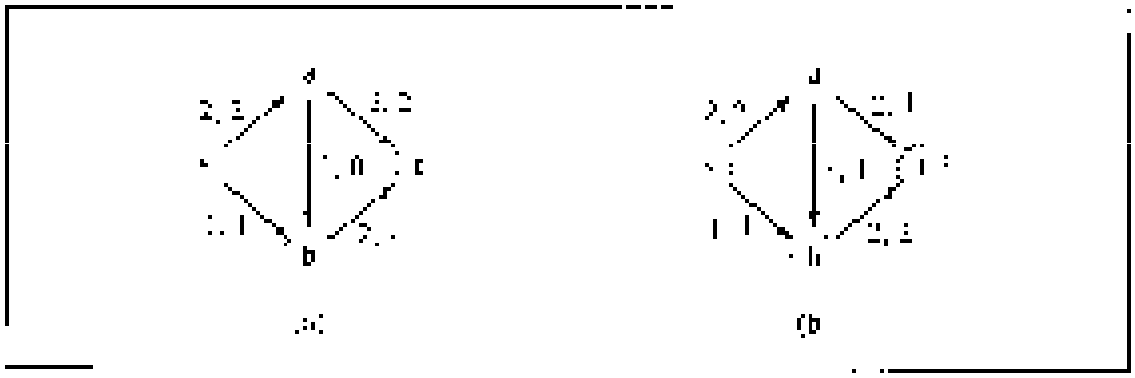
What is the complexity of this algorithm? There are a maximum of $|V|$ layerings (phases) and in a $O(|E|)$ steps to layer the network. Hence, finding all the layered networks requires $O(|V||E|)$ steps. Moreover, there are $O(|E|)$ paths per phase (per layered network) and, due to backtracking, $O(|E|)$ steps to find one path, and since there are $O(|V|)$ layered networks, in the worst case, $O(|V|^2|E|^2)$ steps are required to find the augmenting paths. The actual augmentation depends on the efficiency of the algorithm, which is better than $O(|V||E|)$ if a breadth-first search is used (as in Dinic's algorithm). The important factor is the number of steps to find an augmenting path, which is now $O(|V||E|)$. Edmonds and Karp's key insight was to prepare the network for creating layered networks quickly, as established, to add at most $O(|V|)$ steps.

The difference in performance between `FordFulkersonAlgorithm()` and `processAugmentingPath()` is very large. The most important difference is the amplification factor between the number of edges and the number of vertices. Only the edges in adjacent vertices that do not exceed augmenting paths and the length of each in the layered network are considered.

8.8.2 Maximum Flows of Minimum Cost

In the previous discussion, edges had two associated values: capacity and flow. Now, edges have a third value, cost. Each unit of flow through an edge costs a certain amount. Although many different augmenting flows through the network are possible, we are interested only in those augmenting flows that are cheapest. For example, Figure 8.24 illustrates two possible maximum flows for the same network. Note that, for the first case, the edge labeled x is used at all; only in the second case are all the edges used to carry flow. The breadth-first algorithm leads to the first circumstance. Flow can be increased after establishing it. However, in many situations, this is not a good decision. If there are many possible maximum flows, it does not mean that any one of them is equally good.

FIGURE 8.24 Two possible maximum flows for the same network



Consider the following example. If edges are roads between towns (as in Example 8.23), then it is not enough to know that a road has one or two lanes to choose a shortest path. For example, if the alternative is very long and distance(dollars) and distance(days) are relatively important, then it is better to consider the second maximum flow (Figure 8.24b) as a viable option rather than the first (Figure 8.24a). However, this may not be enough. The shorter way can have no pavement or can be muddy in wet weather, sometimes blocked by boulders, among other disadvantages. Hence, using the distance as the sole criterion for choosing a road is insufficient. Taking the roundabout way may bring us to the destination faster and cheaper (in terms of money and gasoline burned).

We clearly need a third parameter for an edge: the cost of sending one unit of flow through this edge. The problem now is how to find a maximum flow of minimum cost. More formally, if for each edge of the network, in addition to its capacity, a unit cost of flow is determined so that it costs a cents to transport one unit of flow over edge e, then we need to find a maximum flow f of minimum cost, or a flow such that

$$\text{cost}(f) = \min\{\text{cost}(g) \mid g \text{ is a maximum flow}\}.$$

Finding all possible maximum flows and comparing their costs is not a good solution because the amount of work to find all such flows can be prohibitive. Algorithms are needed that find not only a maximum flow but the maximum flow of minimum cost.

One strategy is based on the following theorem, proved first by W. S. Jewell, R. G. Busacker and P. J. Gowen, and independently by M. Iri (Proceedings of the Symposium on

Theorem. If f is a minimal cost flow with the flow value v, and p is a minimal cost augmenting path sending a flow of value 1 from source to sink, then the flow $f + p$ is minimal and its flow value is $v + 1$.

The theorem should be intuitively clear. If we found the cheapest way to send v units of flow through the network and afterward found a path for sending 1 unit of flow from the source to the sink, then we found the best way to send $v + 1$ units using the rule which is a combination of the two already determined and the path just found. If this augmenting path allows for sending 1 unit for minimum cost,

ther it also allows for sending 2 units of flow on edge (u, v), and 3 units on edge (v, t), up to the edge's capacity. The maximum amount of units that can be sent is still 3 units, that is,

$$\min(\text{capacity}(u, v) - \text{flow}(u, v), \text{capacity}(v, t) - \text{flow}(v, t))$$

This also suggests how we can proceed systematically to find the optimal maximum flow. We can start with flow 0 for the first edge we find (or any edge for that matter) through the path as possible. After the second iteration we find another path $(s, 2), (2, t)$ with capacity 1, and we send 1 unit through this path as well. In this path the edge $(2, t)$ is full, and so we can't push any more flow through it. Now we can't find any more paths from s to t, so we must accept that we're done.

Now that we've developed a feel for finding maximum flow, let's see how we can translate this into code. To find the shortest path, we use the shortest path algorithm we developed earlier. Recall that we can find the shortest path in an unweighted graph using breadth-first search. We can also find the shortest path in a weighted graph using Dijkstra's algorithm. Since we're looking for the shortest path in the residual graph, we want to find a path from s to t that has as much room as possible to send flow through it. Therefore, it makes sense to use an algorithm which solves the shortest path problem, so it shouldn't be surprising that we modify Dijkstra's algorithm to solve the maximum flow problem. We explore Dijkstra's algorithm and how to modify it for solving the maximum flow problem in more detail at the end of this chapter. Here is the algorithm:

```
modifiedDijkstrasAlgorithm(network, s, t)
    for all vertices v
        flow[v] = 0;
        cost[v] = ∞;
    setToVisit = emptyPriorityQueue();
    add(s, 0) to setToVisit;
    labeled = new Set();
    while (true)
        v = a vertex in labeled with minimal cost(v);
        if v == t
            if cost[t] == ∞ // no path from s to t can be found
                return failure;
            else return cost[t];
        add v to labeled;
        for all vertices u not in labeled and adjacent to v
            if forward edge(v, u) and cost[u] > cost[v] + cost(v, u) - cost[v]
                label[u] = (v, min(flow[v], forward edge(v, u)), cost[v] + cost(v, u));
            else if backward edge(u, v) and flow(u, v) > 0 and cost[u] > cost[v] - cost(u, v)
                label[u] = (v, min(flow[v], flow(u, v)), cost[v] - cost(u, v));

augmentFlowAlongPath(network, minCostAndPath, s, t)
    while modifiedDijkstrasAlgorithm(network, s, t) is successful
        augmentPath(network, t);
```

modifiedDijkstrasAlgorithm keeps track of three things at a time so that the label for each vertex is the triple

$$\text{label}[v] = (\text{parent}[v], \text{flow}[v], \text{cost}[v])$$

First, for each vertex v, it records the predecessor u of the vertex through which v is accessible from the source s. Second, it records the maximum amount of flow that can be

be pushed through the path from s to t and eventually to t? If not, is there a cost of passing t the edges from the source to v for a forward edge (v, w), $cost[v]$ is the sum of the costs already accumulated along the path to v plus the cost of sending one unit of flow through edge (v, w). For backward edge (w, v), the marginal cost of passing through this edge is the negative of the cost associated with edge (w, v). Also, flows of edges included in augmenting paths are maximized; it is easy to prove and by augmenting paths (see problems).

Figure 8.23(b) illustrates an example of the first iteration of the while loop. Vertex s is labeled (s, ∞), and the three vertices adjacent to s are labeled, $label[a] = (s, 3, 6)$, $label[b] = (s, 4, 7)$, and $label[c] = (s, 1, 1)$. Clearly the vertex with the smallest cost is vertex c, namely vertex c. Now, c is labeled $= [c]$ and two arrows acquire new labels: $label[d] = (c, 1, 3)$ and $label[b] = (c, 1, 2)$. At the bottom, vertex t is now labeled; its cost is 6. Vertex c receives a new label, $(c, 2, 3)$, because the cost of passing through it is smaller than accessing it directly from s. Vertex f, which is adjacent to c, does not get a new label since the cost of sending one unit of flow from s to f through c, 5, exceeds the cost of sending that unit through s, which is 2. Figure 8.24(a) illustrates f as it is known. Labeled vertices $[source]$ and $index[f] = [s, c]$. After the second iteration, the augmenting path is as pictured in Figure 8.24(b). The right-to-left traversal of the path s, c, f, t is apparent from Figure 8.24(a). The execution sequence $mcdi2.2.cpp1$ $kecture4$ $tgc=1.dmy$ is invoked four times and in the last invocation no other path can be found from s to t. Note that the same paths were found here as in Figure 8.20, although in a different order, which reveals that the cost of these paths is the cost of the first detected path (Figure 8.25a) is 6, the cost of the second path (Figure 8.25b) is 8, the cost of the third (Figure 8.25c) is 9, and the cost of the fourth (Figure 8.25d) is 10. The distribution of flows for particular edges allowing for the maximum flow is slightly different. In Figure 8.20, $edge[s, a]$ transmits 2 units of flow, $edge[s, b]$ transmits 2 units, and $edge[s, c]$ transmits 1 unit. In Figure 8.25, the same three edges transmit 3, 5, and 2 units, respectively.

8.9 Matching

Suppose that there are five job openings a, b, c, d, and e, and five applicants p, q, r, s, and t. The qualifications shown in the table

Applicants:	p	q	r	s	t
Jobs:	a, c	b, d	a, e	c	c, d, e

The problem is how to find a worker for each job; that is, how to match jobs and workers. There are many problems of this type. The job matching problem can be modeled with a bipartite graph. A bipartite graph is one in which the set of vertices V can be divided into two classes V_1 and V_2 such that, for each edge (v, w), if vertex v is in one of the two sets V or V_2, then w is in the other set. In this example, one set of vertices, V_1, represents applicants, the other set, V_2, represents jobs, and edges represent jobs for which the applicants are qualified (Figure 8.26). The task is to find a match between jobs and applicants so that one applicant is matched with one job. In a general

FIGURE 8.25 Finding a maximum flow of minimum cost.

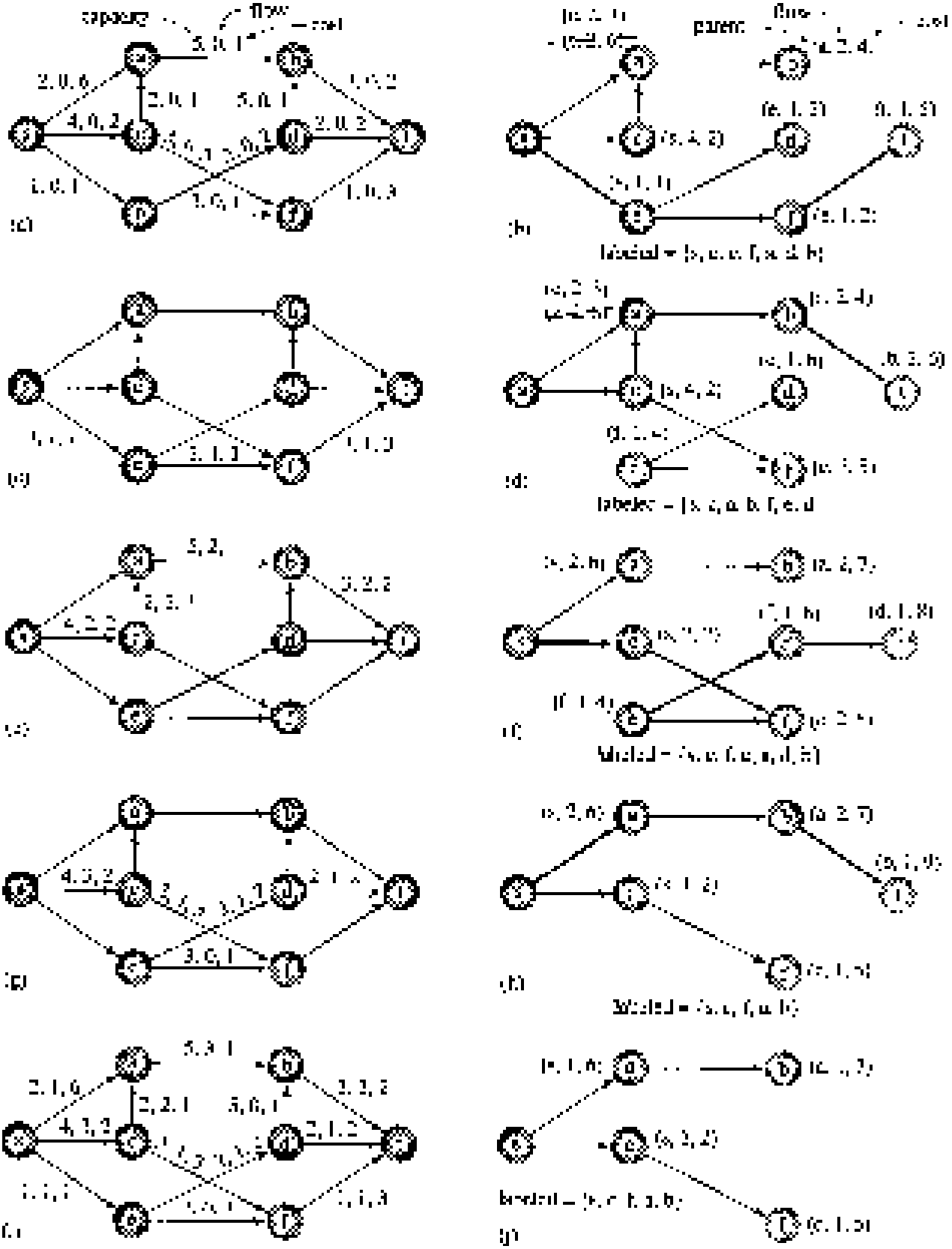


FIGURE 8.26 Matching for applicants with four jobs.

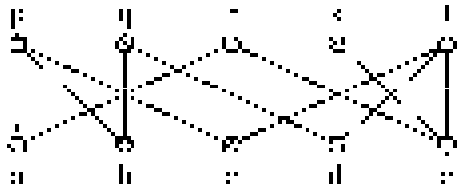
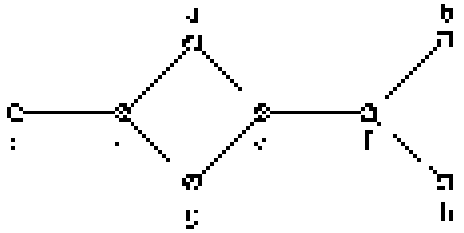


FIGURE 8.27 A graph with matchings $M_1 = \{edge(a, edge(f)\}$ and $M_2 = \{edge(a), edge(c), edge(f)\}$.

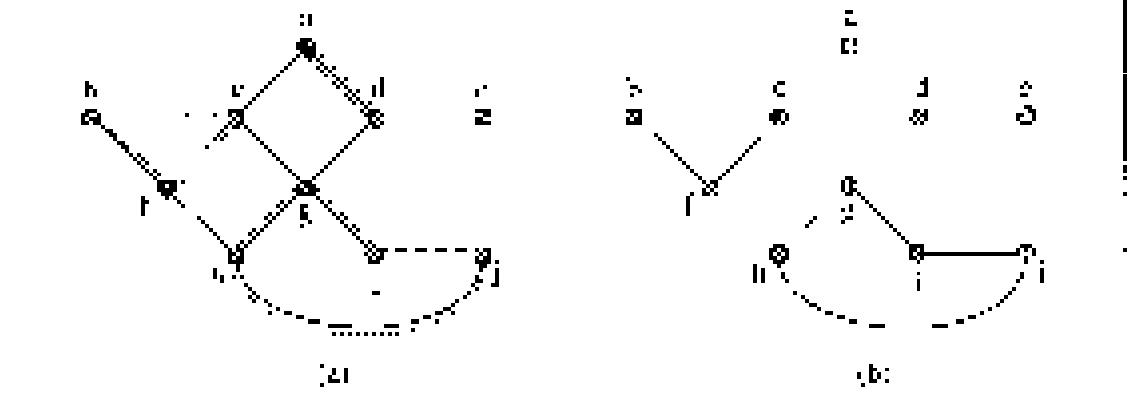


case, there may not be enough applicants, or there may be no way to assign an applicant for each opening, even if the number of applicants exceeds the number of openings. Hence, the task now is to assign applicants to as many jobs as possible.

A *matching* M in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges share the same vertex; that is, no two edges are adjacent. A *maximum matching* is a matching that contains a maximum number of edges so that the number of unmatched vertices (that is, vertices not incident with edges in M) is minimal. For example, in the graph in Figure 8.27, the sets $M_1 = \{edge(a), edge(f)\}$ and $M_2 = \{edge(a), edge(c), edge(f)\}$ are matchings, but M_2 is a maximum matching, whereas M_1 is not. A *perfect matching* is a matching that pairs all the vertices of graph G. A matching $M = \{edge(p), edge(q), edge(r), edge(s), edge(t), edge(u)\}$ in Figure 8.26 is a perfect matching, but there is no perfect matching for the graph in Figure 8.27. A *matching problem* consists in finding a maximum matching for a certain graph G. The problem of finding a perfect matching is also called the *assignment problem*.

An *alternating path* for M is a sequence of edges $\{edge(v_1, v_2), edge(v_2, v_3), \ldots, edge(v_{k-1}, v_k)\}$ that alternately belongs to M and to $E - M$ (set of edges that are not in M). An *augmenting path* for M is an alternating path whose beginning and end vertices are not incident with any edges in matching M. That is, an augmenting path has a odd number of edges, $k - 1$, of them belonging to M and k not in M. If edges in M are replaced by edges not in M, then there is a one more edge in M than before the interchange. Thus, the cardinality of the matching is augmented by one.

FIGURE 8.28 (a) Two matchings M and N in a graph $G = (V, E)$ and (b) the graph $G' = (V, M \oplus N)$



(a) (b)

A symmetric difference between two sets, $X \oplus Y$, is the set

$$X \oplus Y = (X - Y) \cup (Y - X) = (X \cup Y) - (X \cap Y)$$

In other words, a symmetric difference $X \oplus Y$ includes all elements from X and Y but excludes those elements that belong to the intersection of X and Y.

Lemma 1. If two matchings M and N in a graph $G = (V, E)$ are of unequal size, $|M| < |N|$, then $M \oplus N$ contains a connected component that is the subgraph $G' = (V, M \oplus N)$ and is either (a) a single vertex, (b) a cycle with an even number of edges alternately in M and N, or (c) a path whose edges are alternately in M and N and such that each end vertex of the path is matched by one of the two edges that are in M but not N. (Note that the whole path should be considered, not just part of it, to count the edges in the alternating sequence.)

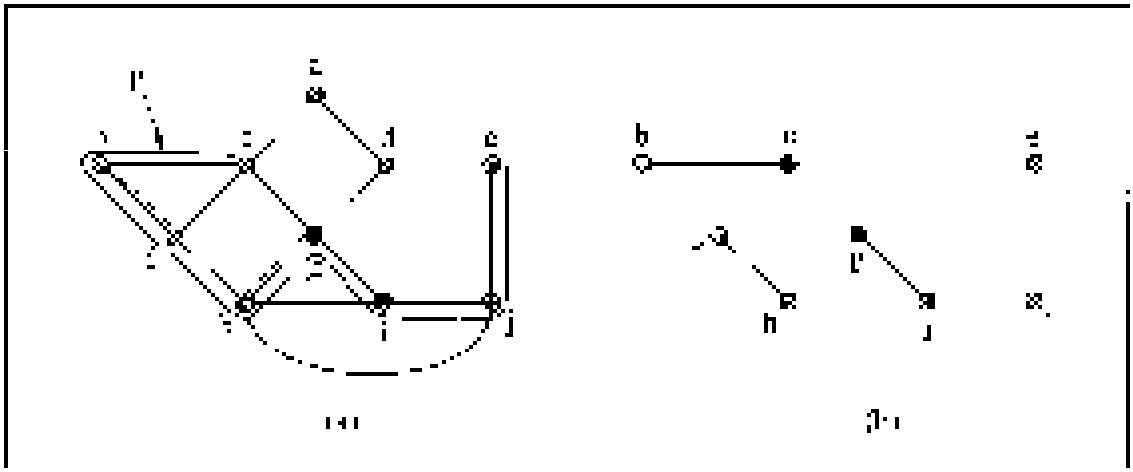
Proof. Each vertex in G', by virtue of $M \oplus N$, is met by edges from each matching and can be incident with at most two, each in one matching. Thus, each connected component of G' is either a single vertex, a path, or a cycle. If it is a cycle or a path, the edges in it alternate between those in M and those in N, by definition of matching and $M \oplus N$. Thus, in a cycle the number of edges must be even. If it is a path, then the degrees of both end vertices is 1 so that they can be met, respectively, only by one of the matching edges in each.

Figure 8.28 contains an example. A symmetric difference between matching $M = \{\text{edge}(a,b), \text{edge}(c,d), \text{edge}(e,f), \text{edge}(g,h)\}$ marked with thick solid lines and matching $N = \{\text{edge}(a,f), \text{edge}(b,c), \text{edge}(d,e), \text{edge}(g,h), \text{edge}(i,j)\}$ shown in dotted lines in the set $M \oplus N = \{\text{edge}(a,b), \text{edge}(c,d), \text{edge}(e,f), \text{edge}(a,f), \text{edge}(b,c), \text{edge}(d,e), \text{edge}(i,j)\}$, which consists of one path and one cycle (Figure 8.28(b)). The vertices of graph G that are not incident with any of the edges in $M \oplus N$ are isolated in the graph $G' = (V, M \oplus N)$.

Lemma 2. If M is a matching and P is an augmenting path for M then $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$.

Proof. By definition of symmetric difference, $M \oplus P = (M - P) \cup (P - M)$. Let v be a vertex that is not an end vertex of P. All other vertices incident with edges in P are matched by edges in M. Hence, an edge in $M - P$ contains any vertex in P. Thus, edges in $M - P$ share no vertex

FIGURE 8.29 (a) Augmenting path P and matching M, and (b) the matching $M \oplus P$.



in M with edges in $P - M$. Moreover, since P is a path, it has exactly one more edge in $P - M$ than it has in $P \cap M$. Therefore, $|M \oplus P| = |M| + 1$. Finally, $M \oplus P$ is a union of vertex-disjoint paths and cycles, because each vertex has degree 0, 1, or 2 in $M \oplus P$. (Vertices not incident to P have the same degree in $M \oplus P$ as in M, vertices incident to P but not to an endpoint of P have degree 2 in $M \oplus P$ because exactly one incident edge is in P and exactly one is in M, and endpoints of P have degree 1 in $M \oplus P$ because exactly one incident edge is in P.) ■

Figure 8.29 illustrates this lemma. For example, in M, vertex b is matched to vertex c, and vertex d is unmatched. The augmenting path P for M is the path a, b, c, d, e, f. Vertex a is unmatched, and the resulting matching $M \oplus P = \{(a, b), (c, d), (e, f), (g, h), (i, j)\}$ includes all the edges from the path P that were originally unmatched edges in M. So is vertex f, but the new matching includes a longer matching path. Its edges alternate between matched and unmatched edges and are reversed.

Theorem (Berge): A matching M in a graph G is a maximum matching if and only if there is no augmenting path connecting two unmatched vertices in G.

Proof: ⇒ By Lemma 2, if there were an augmenting path in M, then a larger matching could be produced, because M would not be a maximum matching.

⇐ Suppose M is not a maximum matching and a matching M' is a maximum. Let $G' = (V, M \oplus M')$. By Lemma 1, connected components of G' are either cycles of even length or paths (isolated vertices are not included here). If it is a cycle, then half of its edges are in M' and are in M because the edges are alternating between M and M'. If it is an even path, then it also has the same number of edges from M and M'. However, if it is an odd path, it has more edges from M' than from M, since $|M'| > |M|$, and both end vertices are incident with edges from M'. Hence, it is an augmenting path, which leads to a contradiction with the assumption that there is an augmenting path. □

This theorem suggests that a maximum matching can be found by beginning with an initial matching, possibly empty, and then by repeatedly finding new augmenting paths and increasing the cardinality of matching until no such path can be found. This requires an algorithm to determine alternate paths. It is much easier to develop

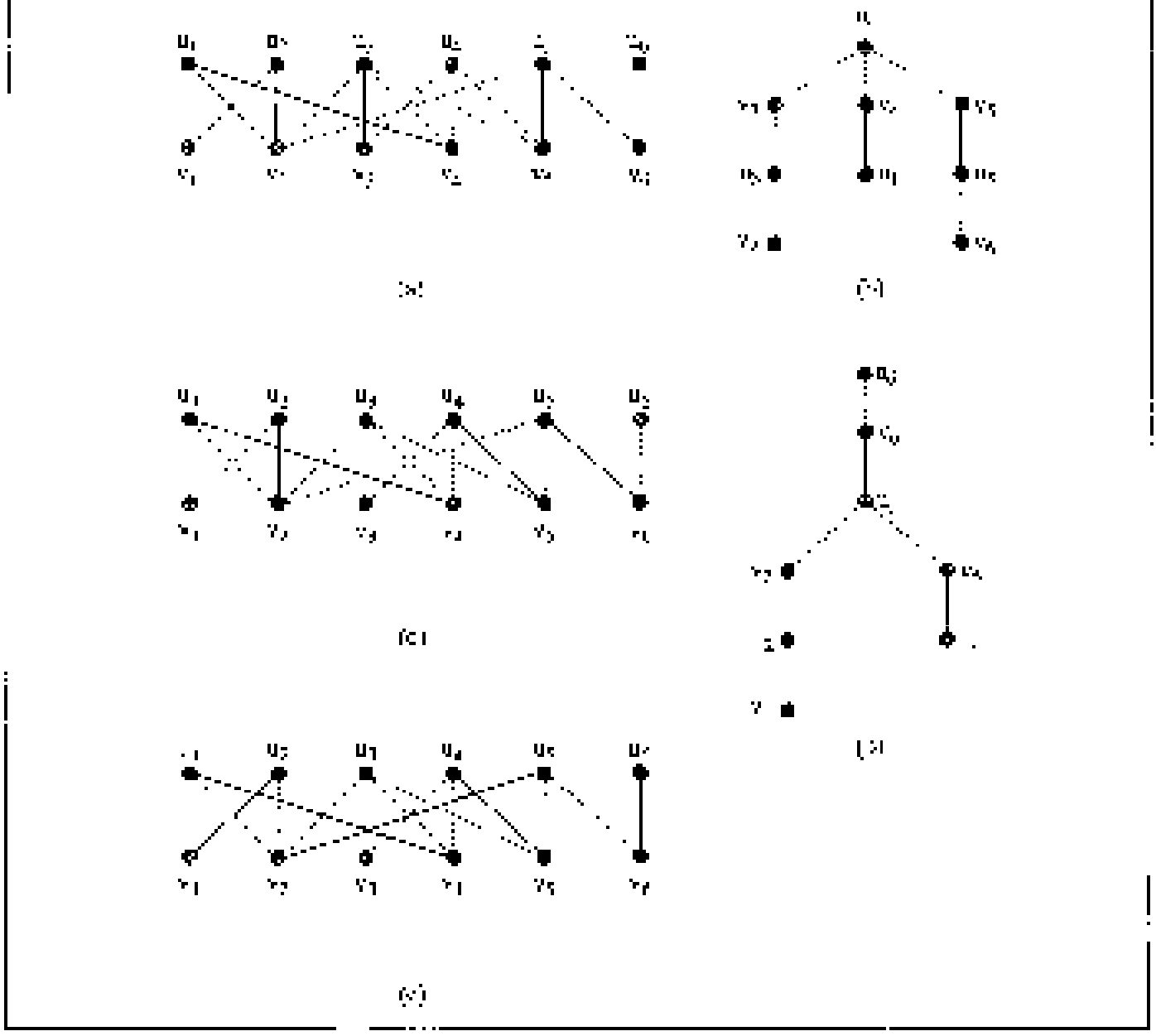
such as algorithms for bipartite graphs that for any graphs; therefore, we start with a discussion of this simpler case.

To find an augmenting path, breadth-first search is modified to allow for always finding the shortest path. The procedure builds a tree, called a *Hungarian tree*, with an unmatched vertex in the root consisting of alternating paths, and a success is pronounced as soon as it finds another unmatched vertex than the one in the root (that is, as soon as it finds an augmenting path). The augmenting path allows for increasing the size of matching. After no such path can be found, the procedure is finished. The algorithm is as follows:

```
findMaximumMatching(bipartite graph)
    for all unmatched vertices v
        set level of all vertices to 0;
        set parent of all vertices to null;
        level[v] = 1;
        last = null;
        clear queue;
        enqueue(v);
        while queue is not empty and last is null
            v = dequeue();
            if level[v] is an odd number
                for all vertices u adjacent to v such that level[u] = 0
                    if u is unmatched                    // the end of the augmenting
                        parent[u] = v;                   // path is found;
                        last = u;                        // the unmatched vertex in the alternating
                        break;                           // exit the for loop;
                    else if u is matched but not visited
                        parent[u] = v;
                        level[u] = level[v] + 1;
                        enqueue(u);
            else // if level[v] is an even number
                enqueue(vertex matched with v);
                parent[u] = v;
                level[u] = level[v] + 1;
        if last is not null // an augmenting path is found
            for (u = last; u is not null; u = parent[parent[u]])
                matched[u] = parent[u];
                matched[parent[u]] = u;
```

An example is shown in Figure 5.30. For the current matching $M = \{x_1y_1, x_2y_2, x_3y_3, x_4y_4\}$ (Figure 5.30a), vertex x_5 is unmatched and is selected as the root. Vertices y_1, y_2, y_3, and y_4 are the vertices adjacent to x_5. They are assigned level 1 in the Hungarian tree (Figure 5.30b). Vertex y_1 is considered first. Since y_1 is matched with x_1, x_1 is added to the tree. Vertex x_1 becomes adjacent to y_1 and y_2 is enqueued. The two successors of y_1 and y_2 are found, that is, x_1 and x_2, respectively, at which the vertex y_1 is considered. This vertex is an unmatched vertex y_2 is considered. This vertex is an odd-level vertex, so all vertices directly accessible from it through edges not in M are included. These are those

FIGURE 8.30 Application of the BreadthFirstSearchMatching algorithm. Matched vertices are connected with solid lines.



such vertices, v_2, v_4, and v_6, but only the first is not yet in the tree, so it is included now. Next, successors of v_2 are tested, but the only candidate, v_1, does not qualify since it is already in the tree. Finally, v_3 is checked, from which we arrive at an unmatched vertex v_6. This marks the end of an augmenting path; hence, the search loop is exited and then matching M is modified by including in M the edges in the newly found path that are not in M and excluding from M the edges of the path that are there. Since the path has one more edge in M' than not in M, after such modification the number of edges in M is increased by one. The new matching is shown in Figure 8.30e.

After finding an augmenting path, search for another augmenting path begins. But now there are still two unmatched vertices, so it still remains possible that a larger matching can be found. In the second iteration of the main loop, we begin with the vertex u_2 which eventually leads to the path in Figure 8.30(b) that includes an augmenting path which in turn gives a matching as in Figure 8.30(c). There are no unmatched vertices left; thus, the maximum matching is found to be a perfect matching.

Complexity of the algorithm is found as follows. Each alternating path increases the cardinality of matching by one, and since the maximum number of edges in matching M is $|V|/2$, the number of iterations of the outer `for` loop is at most $|V|/2$. Moreover, finding one augmenting path requires $O(|E|)$ steps so that the total cost of finding a maximum matching is $O(|V||E|)$.

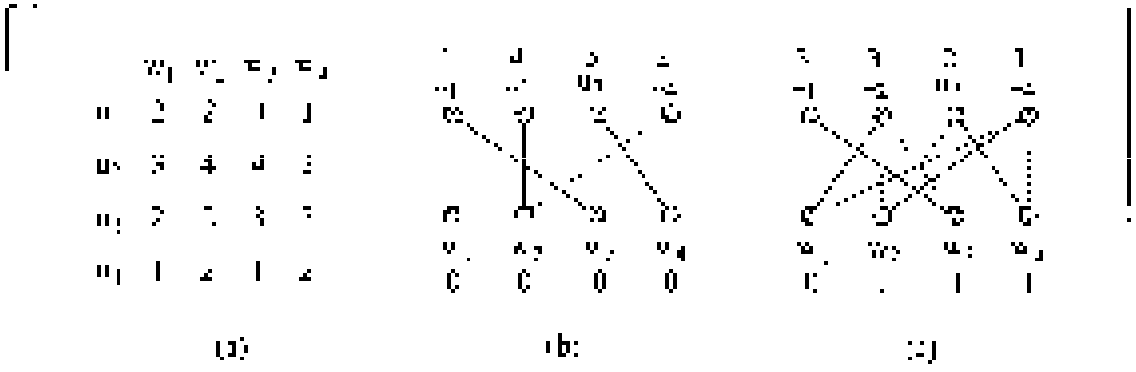
8.9.1 Assignment Problem

The problem of finding a suitable matching becomes more complicated in a weighted graph. In such a graph, we are interested in finding a matching with the maximum total weight. The problem is called an assignment problem. The assignment problem for complete bipartite graphs with two sets of vertices of the same size is called an optimal assignment problem.

An $O(|V|^4)$ algorithm is due to Kuhn and Munkres (Bondy and Murty 1976; Thulasiraman and Swamy 1992). For a bipartite graph $G = (V_1, V_2, E)$, $V = V_1 \cup V_2$, we define a labeling function $f: V_1 \cup V_2 \rightarrow R$ such that a label $f(v)$ is a number assigned to each vertex v such that, for all vertices $u \in V_1$ and $v \in V_2$, $f(u) + f(v) \geq$ weight of edge (u, v). Create a set $E_f = \{(u, v) \in E \mid f(u) + f(v) =$ weight of edge $(u, v)\}$ and then an equality subgraph $G_f = (V, E_f)$. The Kuhn-Munkres algorithm is based on the theorem stating that if for a labeling function f and an equality subgraph G_f, graph G contains a perfect matching, then this matching is optimal. This is the algorithm:

```
optimalAssignment()
   G_f = an equality subgraph for some vertex labeling f
   M = matching in G_f
   S = {some unmatched vertex in V1}; // Beginning of an augmenting path S
   T = null;
   while M is not a perfect matching
      J(S) = {w: there is adjacent(u, w) in G_f and u is in S}; // vertices adjacent in G_f to the vertices in S
      if J(S) == T
         a = min{f(u) + f(w) - weight of edge(u, w) | u in S, w not in T};
         for each vertex v
            if v in S
               f(v) = f(v) - a;
            else if v in T
               f(v) = f(v) + a;
         construct new equality subgraph G_f and new matching M;
      else // J(S) != T
         w = a vertex in J(S) - T;
         if w is matched // the end of this augmenting path is
            S = augmenting path just found;
```

FIGURE 8.37 An example of execution of the optimal bipartite assignment algorithm.



(a) (b) (c)

```
S = {some unmatched vertex u1};
T = ∅;
while S ∩ A ≠ ∅ do
    S = S ∪ {neighbors of vertices in S};
    T = T ∪ S;
```

For an example, see Figure 8.37. A complete bipartite graph $G = (\{u_1, \ldots, u_4\} \cup \{v_1, \ldots, v_4\}, E)$ has weights defined as in the matrix in Figure 8.37(a).

0. To get an initial labeling, we choose $l(u_i)$ to be $\max\{w(u_i, v_j) : v_j \in V\}$, that is, the maximum weight in the ith row of the matrix for vertex u_i, and $l(v_j) = 0$, so that for the graph G, the initial labelings are as in Figure 8.37(b). We choose the equality subgraph as in Figure 8.37(b) and, arbitrarily, $S = \{u_1\}$ and $T = \emptyset$.

1. In the first iteration of the while loop, $\Gamma(S) = \{v_1, v_4\}$, because both v_1 and v_4 are neighbors of u_1, which is the only element of S. Because $\Gamma(S) \neq T$, that is, $\{v_1, v_4\} \neq \emptyset$, the `else` clause is executed, whereby $x = v_1$. We simply choose the first element of $\Gamma(S)$ not in T, and because v_1 is not matched, the inner `else` case is executed, in which we extend S to $\{u_1, u_3\}$, because u_3 is both matched and adjacent to v_1, and extend T to $\{v_1\}$.

All the iterations are summarized in the following table.

Iteration	$\Gamma(S)$	S	x	T
0	$\emptyset$	$\{u_1\}$		$\emptyset$
1	$\{v_1, v_4\}$	$\{u_1, u_3\}$	v_1	$\{v_1\}$
2	$\{v_1, v_3, v_4\}$	$\{u_1, u_3, u_2\}$	v_3	$\{v_1, v_3\}$
3	$\{v_1, v_2, v_3, v_4\}$	$\{u_1, u_3, u_2, u_4\}$	v_4	$\{v_1, v_3, v_4\}$
4	$\{v_1, v_2, v_4\}$			

In the fourth iteration, the second time the outer `if` statement becomes true, $\Gamma(S) = T$ and $\Gamma(S)$ and T are now equal, so the algorithm computes $\alpha_l = \min\{l(u_i) + l(v_j) - w(u_i, v_j) : u_i \in S, v_j \notin T\}$ as $\min\{l(u_1) + l(v_2) - w(u_1, v_2), l(u_1) + l(v_4) - w(u_1, v_4), l(u_3) + l(v_2) - w(u_3, v_2), l(u_3) + l(v_4) - w(u_3, v_4), l(u_2) + l(v_2) - w(u_2, v_2), l(u_2) + l(v_4) - w(u_2, v_4), l(u_4) + l(v_2) - w(u_4, v_2), l(u_4) + l(v_4) - w(u_4, v_4)\} =$

min[4 + 0 − 3, 4 + 0 − 3, 3 + 0 − 4, 2 + 1 − 1] = 1. With this distance, the labels of vertices in graph G are updated to become the labels in Figure 8.31c. The labels of all four vertices in S are decremented by 1, and all three vertices in T are incremented by the same value. Next, an equality subgraph is created that includes all the edges, as in Figure 8.31d, and then the matching found that includes edges drawn with solid lines. This is a perfect matching, and hence, an optimal assignment, which concludes the execution of the algorithm.

8.9.2 Matching in Nonbipartite Graphs

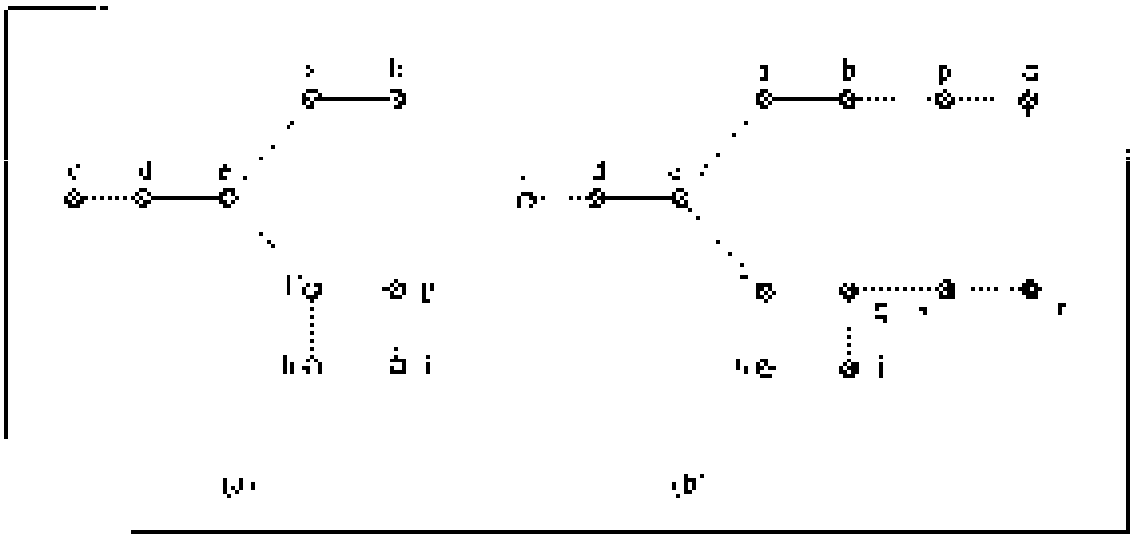
The algorithm `findMaxCardinalityMatching` is not powerful enough to properly process nonbipartite graphs. Consider the graph in Figure 8.32a. If we start building a tree using breadth-first search and demonstrate a augmenting path from vertex a, then vertex d is at an odd level, vertex e is at an even level, and vertex f is at an even level. If after processing vertex d vertex f is processed, then only edge (f, g) is included in the tree (using dashed lines for the edges of the tree), but in this way the augmenting path (a, b, d, e, f, g) cannot be detected since vertex f has been labeled and hence blocks access to it and consequently vertex g. The path (a, b, d, e, f, g) could be found if we used a depth-first search and expanded the search by finding augmenting paths leading through f, since the search would first process the path (a, b, d, e, f) and then would discover g. However, if f was not in the graph, then the very same depth-first search would miss the path (a, b, d, e, f, g). Because first the path (a, b, d, e, f) would be expanded, and all the vertices of path (a, b, d, e, f, g) would be marked.

A solution to the problem is the presence of cycles with an odd number of edges. But it is not just the odd number of edges that causes the problem. Consider the graph in Figure 8.32b. The `findMaximumCardinalityMatching` algorithm is successful because the search can only go between the depth-first search and breadth-first search lists. But (a, b, d, e, c) is an odd cycle. The problem arises in a special type of cycle with an odd number of edges, which are called blossoms. The real difficulty of determining augmenting paths for graphs with blossoms is due to jacks in boxes. But first some definitions.

A *blossom* is an alternating cycle $v_1, v_2, \ldots, v_{2p+1}, v_1$ such that edge (v_1, v_2) and edge (v_{2p}, v_{2p+1}) are not in the matching, but each cycle edge (v_{2i}, v_{2i+1}) is in the matching, for $1 \leq i \leq p$. The vertex v_1 is called the *base* of the blossom. An even-length alternating path is called a *stem*. A path of length zero that has only one vertex is also a stem. A blossom with a stem whose origin is incident with the base of the blossom is called a *flower*. For example, in Figure 8.32c, path (a, b), path (a), and path (a, b, d, e) are stems, and cycle (c, d, e, f, g, c) is a blossom with a base c.

The problems with blossoms arise if a prospective augmenting path leads into a blossom through the base. Depending on which edge is chosen to continue the path, we may or may not obtain an augmenting path. However, if the blossom is entered through any other vertex v than the base, the problem does not arise because we can choose only one of the two edges of v. Hence, an idea is to prevent a blossom from possibly harmful effects by detecting the fact that a blossom is being entered through its base. The next step is to temporarily remove the blossom from the graph

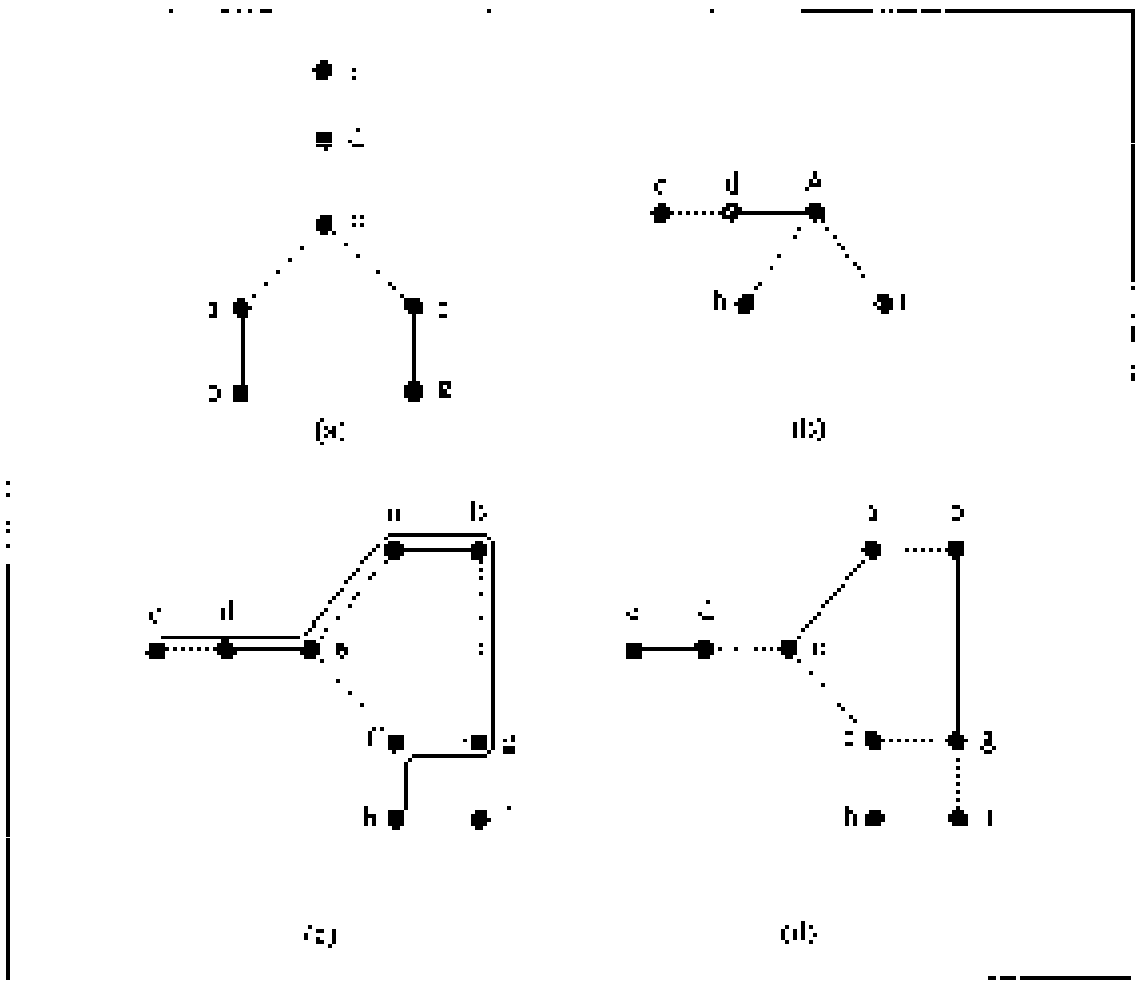
FIGURE 8.32 Application of the find-maximum-cardinality-matching algorithm to a non-bipartite graph.



by matching the pairs of edges incident to the vertices of the blossom and to attach each such pair to the vertex in the blossom. The search for an augmenting path continues until an augmenting path which leads to a vertex representing a blossom is found. The blossom is expanded and the path through it is determined by going backward from the edge that leads to the blossom to one of the edges incident to the base.

The first problem is how to recognize that a blossom has been created through the base. Consider the Hungarian tree in Figure 8.33a, which is generated using breadth-first search in the graph in Figure 8.31a. Now, if we try to find augmenting paths, the only possibilities since edge (d, h) is in the matching are edges (e, h) and (e, g). Such edges would lead to vertices at an even level in the tree. But g has already been labeled and h is located at an odd level. This means a blossom detected. If a labeled vertex is found to have two different paths, one of even length and another of odd length, between it and the root, then we know that we are in the middle of a blossom centered through its base. Now we trace the paths from g and h back in the tree until a common vertex is found. This common vertex, vertex a in our example, is the base of the detected blossom. The blossom is now represented by a vertex A, which leads to a transformed graph as in Figure 8.33b. The search for an augmenting path restarts from vertex A and continues until such a path is found, namely, path a, d, A, h. Now we expand the blossom represented by A and have the augmenting path through the blossom. We do that by starting from edge (g, h), which is now edge (g, h). Since it is an edge not in matching, then from g only edge (g, f) can be considered for the augmenting path. The augmenting path is determined by going through vertices g, f, b, a, and c. We determine the part of the augmenting path c, a, A, h, which corresponds to A (Figure 8.33c) so that the full augmenting path is c, d, e, h, g, f, b, a. After this path is processed, we obtain a new matching as in Figure 8.33d.

FIGURE 8.33 Processing a graph with a blossom.



8.10 Eulerian and Hamiltonian Graphs

8.10.1 Eulerian Graphs

An *Eulerian trail* in a graph is a path that includes all edges of the graph only once. An Eulerian cycle is a circuit that is also an Eulerian trail. A graph that has an Eulerian cycle is called an *Eulerian graph*. A theorem proved by Euler says that a graph is Eulerian if every vertex of the graph is incident to an even number of edges. Also a graph contains an Eulerian trail if it has exactly two vertices incident with a odd number of edges.

Fleury is associated with the oldest algorithm that shows us to find an Eulerian cycle if this is possible. The algorithm takes great care in not traversing a bridge, that is, an edge whose removal would disconnect the graphs G_1 and G_2, since if traversal of G_1 is not completed before traversing such an edge to pass to G_2, it would not be possible to return to G_1. Only after the entire subgraph G_1 has been traversed can the path lead through such an edge. Fleury's algorithm is as follows:

Figure 8.34 Finding an Eulerian cycle.

```
Traverse(graph theGraph)
   v = a vertex in theGraph;
   path = v
   untraversed = graph
   while v has an untraversed edge
      if edge(v,x) is the only remaining untraversed edge
         e = edge(v,x)
         remove e from untraversed
      else e = edge(v,x) such that e is not a bridge in untraversed
      path = path + x
      remove e from untraversed
      v = x
   if untraversed has no edges
      success
   else failure
```

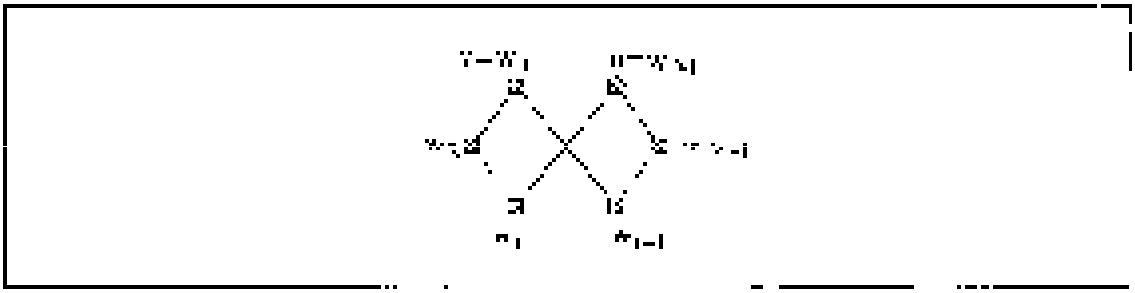
Note that for cases where a vertex has more than one untraversed edge, a connectivity checking algorithm should be applied.

An example of finding an Eulerian cycle is shown in Figure 8.34. The algorithm starts when an edge is chosen, and we need to determine whether the edge is a bridge in the untraversed subgraph or not. For example, in the graph in Figure 8.34a, the traversal begins at vertex *b* by moving across edges *a*, *f*, *e*, and *c*, thereby using the path *b*, *a*, *f*, *e*, *c*, and we need to be careful which untraversed edge is chosen in *c*: *edge*(*c*, *b*), *edge*(*c*, *d*), or *edge*(*c*, *f*) (Figure 8.34b). If we choose *edge*(*c*, *b*), then the remaining three untraversed edges are inaccessible, because in the yet untraversed subgraph *untraversed* = {*vertices*, *edge*(*c*, *b*), *edge*(*c*, *d*), *edge*(*d*, *e*), *edge*(*e*, *f*), *edge*(*f*, *c*)} *edge*(*c*, *b*) is a bridge because it disconnects two subgraphs of untraversed, {*vertex*, *edge*(*d*, *e*), *edge*(*e*, *f*), *edge*(*f*, *d*)} and {*vertex*}.

8.10.2 Hamiltonian Graphs

A *Hamiltonian cycle* in a graph is a cycle that passes through all the vertices of the graph. A graph is called a *Hamiltonian graph* if it includes at least one Hamiltonian

FIGURE 8.35 Crossover edges.



cycle. There is no formula characterizing a Hamiltonian graph. However, it is clear that all complete graphs are Hamiltonian.

Theorem (Ore 1960; Bondy and Chvatal 1976). If edge-set E of graph $G = (V, E)$ is Hamiltonian, and $deg(v) + deg(w) \geq |V|$ when graph (V, E) is also Hamiltonian.

Proof. Consider a Hamiltonian cycle in G that includes edge-set E. This implies that G has a Hamiltonian path $v = v_1, v_2, \ldots, v_{|V|-1}, v_{|V|} = w$. Now we want to find two crossover edges, $edge(v_i, v_{i+1})$ and $edge(v_j, v_{j+1})$, such that $v_1, v_2, \ldots, v_i, v_j, v_{j-1}, \ldots, v_{i+1}, v_{j+1}, \ldots, v_{|V|}, v_1$ is a Hamiltonian cycle in G (see Figure 8.35). To see that this is possible, consider a set S of subscripts of neighbors of v, $S = \{i \mid edge(v, v_{i+1})\}$, and a set T of subscripts of neighbors of w, $T = \{j \mid edge(v_j, w)\}$. Because $S \cup T \subseteq \{1, 2, \ldots, |V| - 1\}$, $|S| = deg(v)$, $|T| = deg(w)$, and $deg(v) + deg(w) \geq |V|$, then S and T must have a common subscript so that the two crossover edges, $edge(v_i, v_{i+1})$ and $edge(v_j, w)$, exist. ■

The theorem, in essence, says that some Hamiltonian graphs allow us to create Hamiltonian graphs by eliminating some of their edges. This leads to an algorithm that first expands a graph to a graph with more edges in which finding a Hamiltonian cycle is easy and then manipulates this cycle by adding some edges and removing other edges so that eventually a Hamiltonian cycle is formed that includes the edges that belong to the original graph. An algorithm for finding Hamiltonian cycles based on the preceding theorem is as follows (Chvatal 1973):

```
HamiltonianCycle(graph G = (V, E))
   arrLabel(all edges) = 0;
   k = 1;
   H = E;
   G_k = G;
   while G_k contains nonadjacent vertices v, w such that deg_Gk(v) + deg_Gk(w) >= |V|
      E = H ∪ {edge(v, w)};
      G_k = (V, E);
      label(edge(v, w)) = k++;
   // Now contains a Hamiltonian cycle C
   while (k = max(label(edge(p, q)) : edge(p, q) in C)) > 0
      C = Augment(delete from C the one edge labeled k, replace with edges labeled 0 so that C is a cycle);
```

FIGURE 8.36 Finding a Hamiltonian cycle

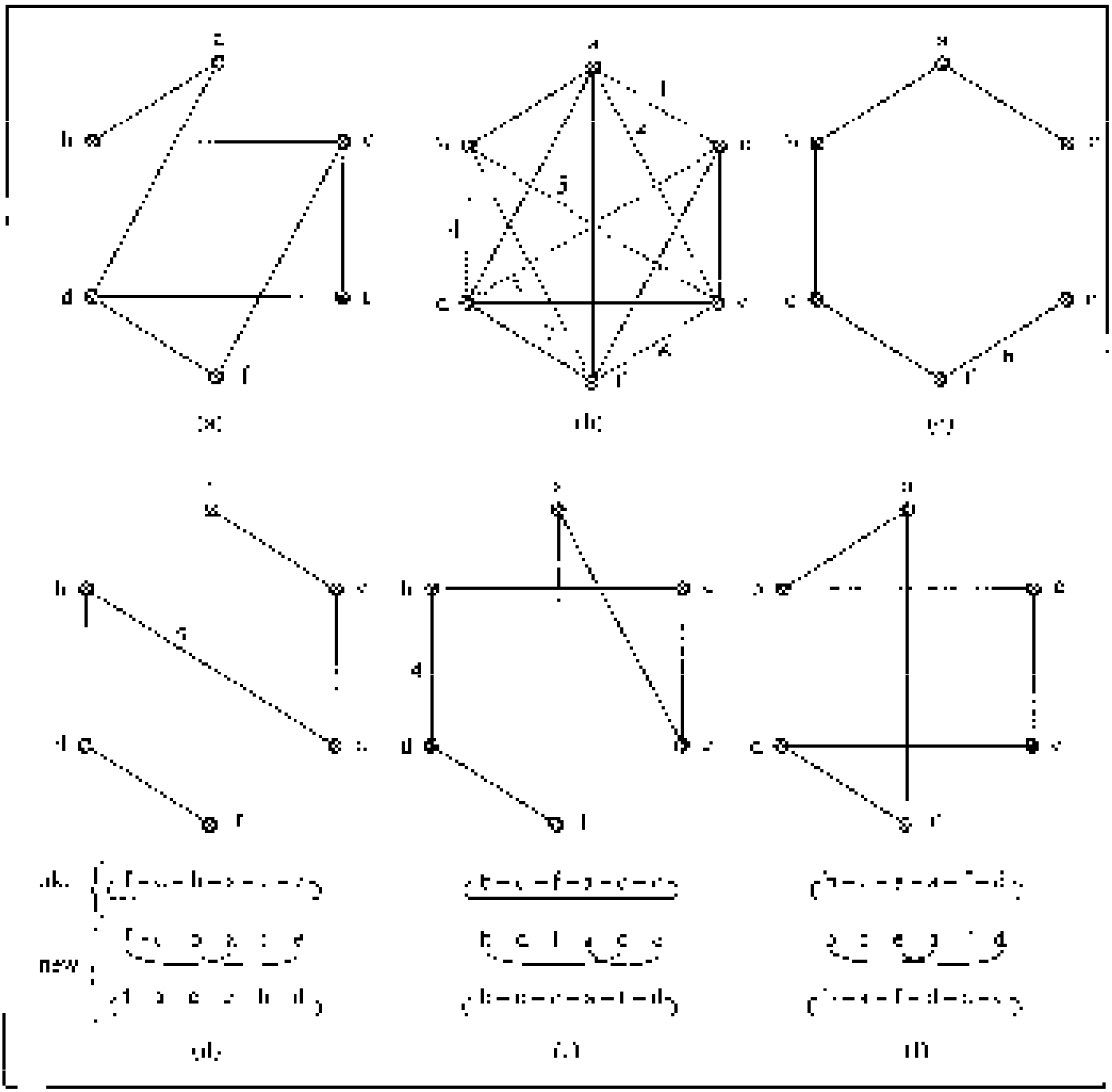


Figure 8.35 contains a complete solution. In the first phase, the 8 vertices are sorted in nondecreasing order of their degrees. In each iteration, two nonadjacent vertices are connected with an edge if the total number of their neighbors is not less than the number of all vertices in the graph. We first look at all the nonadjacent pairs of vertices $deg(u) + deg(v) \geq 8$. All edges are labeled with the number 1 to indicate that they are added in Step 1. Next, vertex a is considered, and because the degree of a just increased by acquiring a new neighbor, b, $deg(a) + deg(e) = 6$, so the edge $\{a, e\}$ is

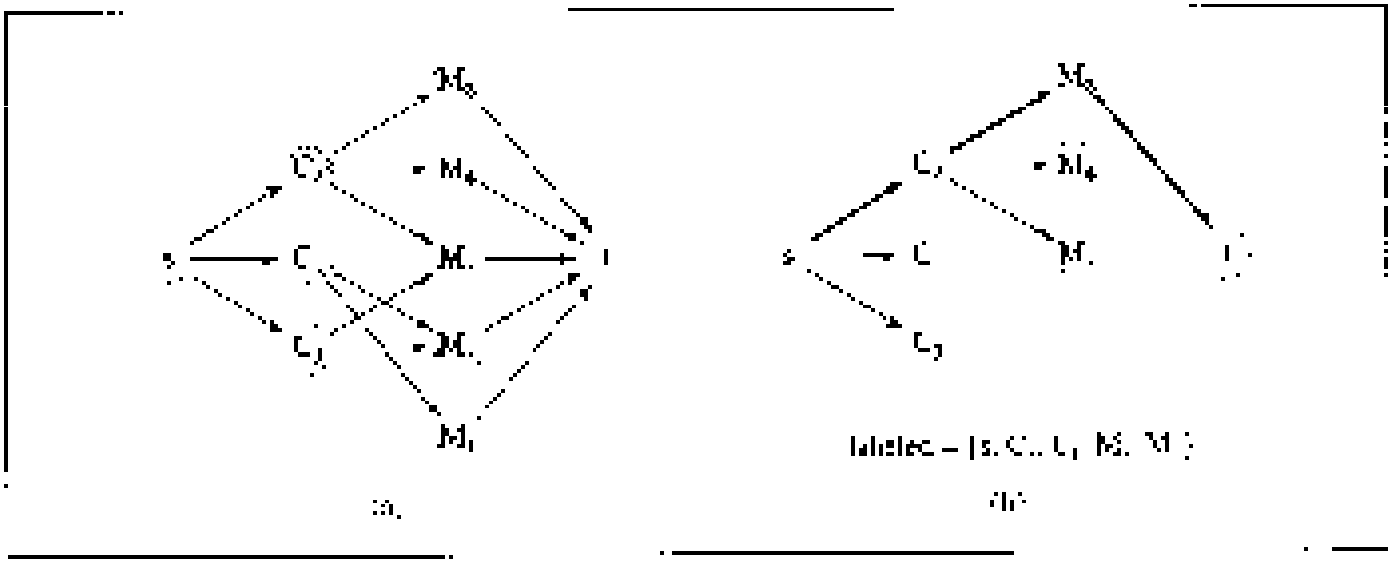
included in T. The next vertex, v, which we try to add to S, is the one for which $d[v]$ is smallest. In this case, v is x with $d[x] = 2$, and S becomes $\{s, x\}$. The values of d and p are then adjusted to reflect the addition of x to S. In this case, $d[y]$ is changed to 3 and $p[y]$ to x. The edge (x, y) is now included in T. The next iteration adds y to S because y has the smallest d value of the vertices not in S. The values of d and p are adjusted to reflect the addition of y to S, which results in $d[z]$ becoming 5 and $p[z]$ becoming y. The edges shown as dashed lines with circles are those in T.

In the second phase of `HamiltonianCycle`, a Hamiltonian cycle is found, if one exists. In this example, an edge with the highest label is found, $edge(x, y)$ (Figure 8.36a). The vertices of the cycle are ordered, beginning with x and ending with y, thereby moving left to right. In this sequence of vertices, we try to find crossover edges by checking edges from two neighbor vertices to the vertices at the ends of the sequence so that the edges cross each other. The first possibility is vertices v and w with $edge(v, x)$ and $edge(y, w)$, but this pair is rejected because the label of $edge(y, w)$ is greater than the largest label of the current Hamiltonian cycle, 5, so the pair is discarded. Next, we try the pair $edge(v, x)$ and $edge(y, t)$, but because the magnitude of label of $edge(v, x)$, 6, the test is unacceptable. The next possibility is the pair $edge(w, x)$ and $edge(y, t)$, which is acceptable, so a new cycle is formed by x, w, t, y (Figure 8.36c–e). In this cycle, a pair of crossover edges is found: $edge(w, t)$ and $edge(s, u)$, and a new cycle is formed, s, t, w, x, y (Figure 8.36e–f), which includes edges only with labels equal to 5. This is only edges from $graph(G)$, which marks the end of execution of the algorithm, with the last cycle being Hamiltonian and including edges in G.

8.11 Case Study: Distinct Representatives

Let there be a set of committees, $C = C_1, \ldots, C_n$, such committees having at least one person. The problem is to determine if possible representatives from each committee so that the committee is represented by one person and each person can represent only one committee. For example, if there are three committees $C_1 = \{M_1, M_2\}$, $C_2 = \{M_1, M_2, M_3\}$, and $C_3 = \{M_1, M_3\}$, then one possible representation is member M_1 representing committee C_1, M_2 representing C_2, and M_3 representing C_3. However, if we have these three committees, $C_1 = C_2 = \{M_1, M_2\}$, and $C_3 = \{M_1\}$, then no distinct representation can be created, since there are only two members in all three committees combined. The latter observation has been proven by P. Hall in the *system of distinct representatives* theorem which can be phrased in the following way:

FIGURE 8.37 (a) A network representing membership of three committees C_1, C_2, and C_3, and (b) the first augmenting path found in this network.



Theorem A nonempty collection of finite nonempty sets $S_1, \ldots, S_n$ has a system of distinct representatives if the union $S_1 \cup \cdots \cup S_n$ has at least n elements.

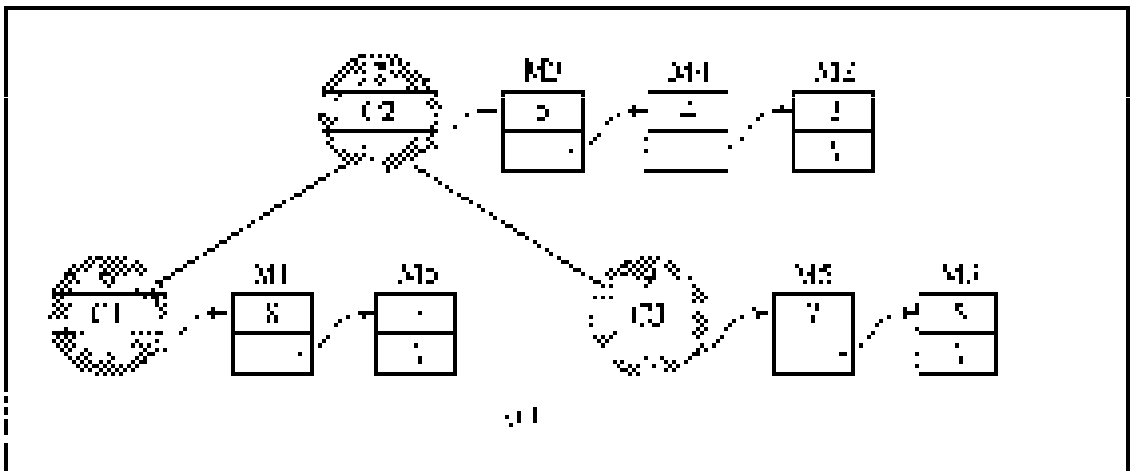
This problem can be solved by creating a network and using it to find a maximum flow in the network. For example, the network in Figure 8.37a can represent the membership of the three committees C_1, C_2, and C_3. There is a dummy source vertex connected to nodes representing committees. These committee vertices are connected to vertices representing their members, and the member vertices are all connected to a dummy sink vertex. We assume that each edge has capacity one. A system of distinct representatives is found if the maximum flow in the network equals the number of committees. The pairs determined by a particular maximum flow algorithm determine the representatives. For example, member M_1 represents the committee C_3 in a solution if (C_3, M_1) is determined.

The algorithm mentioned has two main stages. First, a network is created using a set of committees and members stored in a table. Then the network is traversed to find augmenting paths corresponding to maximum flow in the network. The first stage is specific to the problem of distinct representatives. The second stage can be used for finding the maximum flow of any network since it assumes that the network has been created before it begins.

We store the committees and members from a file. We assume that the name of a committee is always followed by a colon and then by a list of members separated by commas and ended with a semicolon. An example of the following file format is, which contains information corresponding to Figure 8.37:

```
C1: M2, M4, M3;
C2: M1, M4;
C3: M2, M1;
```

FIGURE 8.38 (a) The committee trees created by reading committees.dat; (b) snapshot of the file committees.



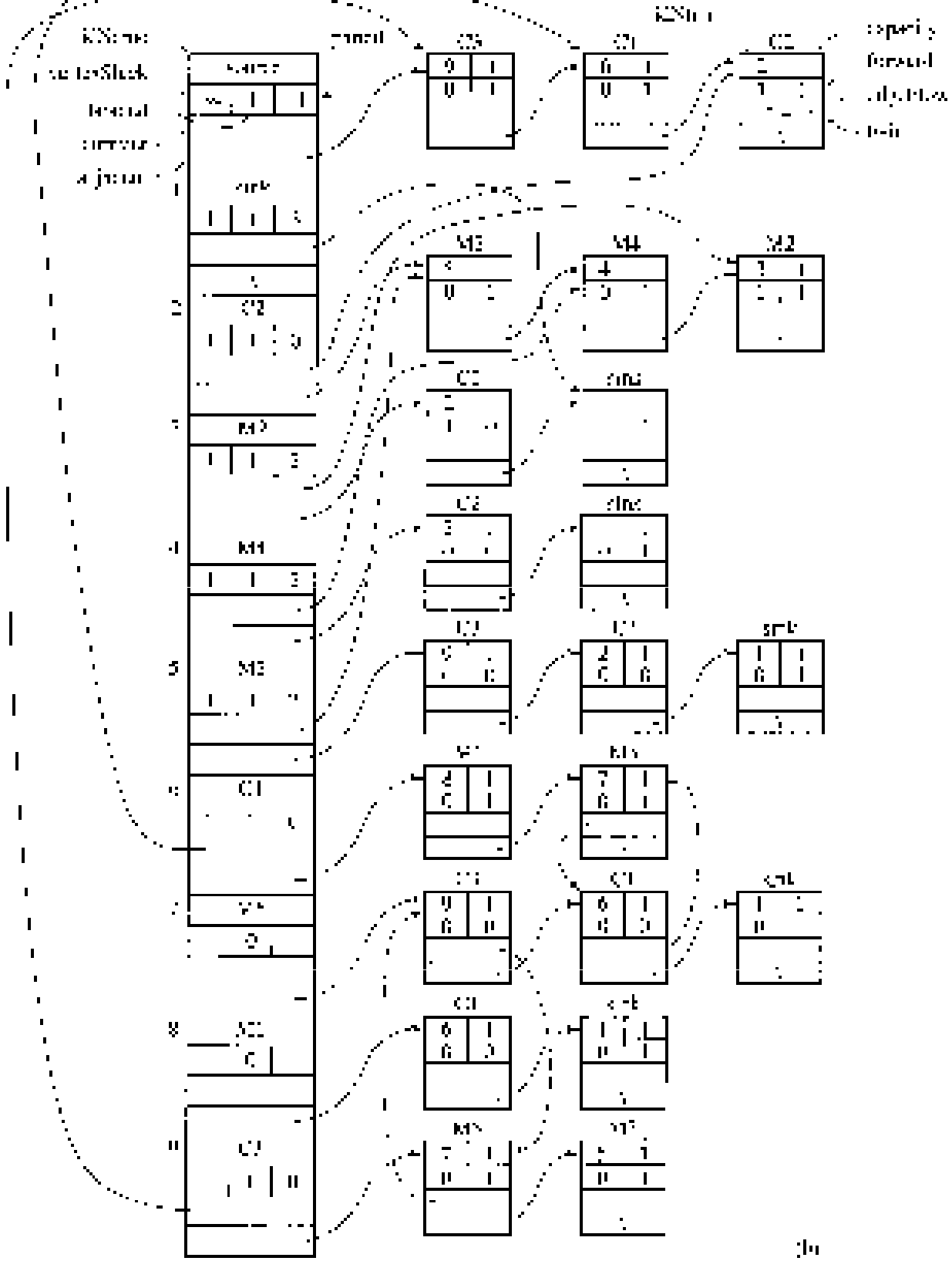
(a)

In preparation for creating a network, two trees are generated: a committee tree and a member tree. The information stored in each node includes the name of the committee or member, an index assigned by the program using a running counter, a pointer to the next node, and an adjacency list to be included later in the network. Figure 8.38a shows committee trees corresponding to the committees of the example file. The adjacency lists are implemented using the STL class `list`. The adjacency lists are shown in simplified form, with member indexes only and with forward links only. The STL lists are implemented as doubly linked lists; see Section 3.7.1. A better form of the lists is shown in Figure 8.38b, but only with the forward links. The names of the members are shown above the nodes of these adjacency lists. A separate adjacency list, `adjacentMembers`, is built for the source vertex. Note that the member adjacency list `adjacentMembers` is one of type `list<vertexNode*>::iterator` but a pointer type `list<vertexNode*>::iterator*` is used as these lists are later transferred to the array `vertices`, a non-pointer type would use the assignment to clear out the lists for the tree to the array, which is inefficient. Moreover, in the program, users' pointers to vertices stored in lists in the trees (such as `vertex` and `adjacentMembers`) are used to set up the trees, and the copies of the pointers, if separate, would have different addresses than the originals in the trees, which would inevitably lead to a program crash.

After the file is processed and all committees and members are included in the trees, the presentation of the network begins. The network is represented by the array `vertices`. The index of each cell in `vertices` coincides with the index assigned to each node of the two trees. Each cell in `vertices` contains information necessary for proper processing of the vertex in the form of the `vertex` type class, identification-labeled flag, pointer to an adjacency list, pointer to the committee or member node, and information on the participation-of-vertex list.

An adjacency list of a vertex in position `v` represents edges incident with this vertex. Each node in the list is identified by its position, which is the position in `vertices` of the other vertex. Also, information in each node of such a list includes capacity of the edge, its flow, forward/backward flag, and a pointer to the next node.

Figure 8.38 (b) the network representation produced by TopSortDistinctRepresentatives( ).



is an edge from vertex i to j, then i's adjacency list includes a node containing j, and a forward edge from vertex j to i, and j's adjacency list has a node corresponding to a backward edge pointing to i. However, each edge is represented twice in the network. If a path is found, then the augmenting algorithm operates on two nodes on two adjacency lists. To make it possible, each node on such a list points to the list containing the node representing the reverse edge, if any, taken in the opposite direction.

In the first phase of the process, the function `readCommittees()` builds both the source vertex and the adjacency list for each vertex in the network represented by the data from the file `committees`. Both the vertex and the lists include unit capacities. The function also builds separate adjacency lists for the source vertex.

Note the assignments in which the address operator `&` and dereference operator `*` are used together as in

```
memberVertex = &committeeVector[node->edge[i].begin];
```

For pointers, one operator cancels another so that `p == &*p`. But the function `begin()` returns an iterator indicating the position of the first element on the list, not a pointer. To obtain the address of this element, the dereferencing operator `*` is applied to the iterator to extract the referenced element, and then its address can be determined with the operator `&`.

In the second phase, the program looks for augmenting paths. In the algorithm used here, the source node is always processed first, since it is always pushed first onto `LabeledList`. Because the algorithm requires processing only unlabeled vertices, there is no need to include the source vertex in any adjacency list, since the edge from any vertex to the source has no chance to be included in any augmenting path. Also, after the sink is reached, the process of finding an augmenting path is discontinued, whereby no edge incident with the sink is processed, so there is no need to keep an adjacency list for the sink.

The structure created by `readCommittees()` using the file `committees` is shown in Figure 8.36a. This structure represents the network shown in Figure 8.37a. The numbers in the nodes and array cells are put by `FordFulkersonMaxFlow()` right after finding the last augmenting path 0, 2, 1, 4, that is the path shown in Figure 8.37b. Nodes in the adjacency list of a vertex i do not include the names of vertices accessible from i, only their indices; therefore, these names are shown above each node. The dashed lines show twin edges. In order not to clutter Figure 8.36 with too many links, only the links for two pairs of twin nodes are shown.

The output generated by the program

```
Augmenting paths:
   source -> C2 -> M2 -> sink (sequenced by 1);
   source -> C1 -> M5 -> sink (sequenced by 1);
   source -> C3 -> M3 -> sink (sequenced by 1);
```

determines the following representation. Member M_2 represents committee C_2, M_5 represents C_1, and M_3 represents C_3.

Figure 8.34 contains the code for this program.

Program 8.39 An implementation of the distinct representatives problem.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <limits>
#include <list>
#include <stack>
#include "queue.h"

using namespace std;

class VertexArray;
class LocalVertex;
class Network;

class Vertex {
public:
    Vertex() {
    }
    Vertex(int id, int c, int wf, bool f, Vertex *v = 0) {
        idNum = id; capacity = c; excess = wf; forward = f; twin = v;
    }
    bool operator== (const Vertex &v) const {
        return idNum == v.idNum;
    }
    bool operator!= (const Vertex &v) const { // required
        return idNum != v.idNum;
    }
    bool operator< (const Vertex &v) const { // by the compiler
        return idNum < v.idNum;
    }
    bool operator> (const Vertex &v) const {
        return idNum > v.idNum;
    }
private:
    int idNum, capacity, weight;
    bool forward;    // direction
    Vertex *twin;    // edge in the opposite direction
    friend Network;
```

Continued

Program 8.39 (continued)

```
    friend ostream& operator<< (ostream&, const Vertex&);
};

class NetTreeNode {
public:
    NetTreeNode(ForestDirectory** d) {
        vertexPtr = v;
        adjacent = new List<Vertex>;
    }
    bool operator< (const NetTreeNode& r) const {
        return strcmp(idName,r.idName) < 0;
    }
    bool operator== (const NetTreeNode& r) const {
        return strcmp(idName,r.idName) == 0;
    }
private:
    int idNum;
    char *idName;
    VertexArrayNode **vertexArr;
    List<Vertex> *adjacent;
    friend Network;
    friend LocalTree;
    friend ostream& operator<< (ostream&, const NetTreeNode&);
};

class VertexArrayNode {
public:
    VertexArrayNode() {
        adjacent = 0;
    }
private:
    char *idName;
    int vertexBlock;
    bool labeled;
    int parent;
    Vertex *vertex;              // correspondig vertex: vertex in parent's
    List<Vertex> *adjacent;      // list of adjacent vertices with the same
    friend Network;              // idNum as this node's idNum;
    friend LocalTree;
    friend ostream& operator<< (ostream&, const Network&);
};
```

Listing 8.39 (continued)

```
// Define new visit() to be used by findMin() from generic.h
class LocalItem : public VertexNodeBase {
    void visit(GVWeightedPrimNode* p) {
        ((p->key.vertex)->dist) p->key.idNum].fldNum  = p->key.idNum;
        ((p->key.vertex)->dist) p->key.idNum].wt[fwNum]
                                                  = p->key.wt[fwNum];
    }
};

class Network {
public:
    Network() : sink(0), source(0), size(0), numCompositions(2) {
        vertices = new VertexArray(max);
    }
    void readCompositions(char *compositionFile);
    void findMinNetworkFlow();
private:
    const int sink, source, max;
    int numCompositions;
    VertexArray *vertices;
    VertexArray *verticesPtr; // used by visit() in LocalItem to
                              // update vertices
    int edgeInSink(Vertex *v) const {
        return v->capacity - v->edgeFlow;
    }
    int id(int i, int n) const {
        return n * i + j;
    }
    bool labeled(Vertex *v) const {
        return vertices[v->idNum].labeled;
    }
    void label(Vertex*,int);
    void augmentPath();
    GLboolean terminal;
    GLboolean readNetwork(char*,const GLMetrics&);
};

GLboolean Network::readNetwork(char *fileName, const GLMetrics &m) {
    ifstream in(fileName, ios::in);
    return in;
}
```

Listing 8.39 (continued)

```
ostream& operator<<(ostream& out, const Vertex& v) {
    out << v.label << " " << v.capacity << " " << v.predFlow << " "
        << v.isSource << "| ";
    return out;
}

ostream& operator<<(ostream& out, const Network& net) {
    cout << "Vertices:" << endl;
    for (int i = 0; i < net.numOfVertices; i++) {
        out << i << ": "
            << net.vertices[i].idNum << "|"
            << net.vertices[i].visitedNum << "|"
            << net.vertices[i].label << " "
            << net.vertices[i].pred << "  |"
            << /* net.vertices[i].lastVer << */ "\n";
        if (net.vertices[i].adjacent != 0)
            copy (net.vertices[i].adjacent->begin(),
                  net.vertices[i].adjacent->end()).output();
        out << endl;
    }
    return out;
}

void Network::readCapacities(char *fileName) {
    char i, name[80], ch;
    bool lastNumber;
    ifstream currentVertex, nextVertex;
    Vertex emptyVer(0,1,0,false), *current=&(1,1,0,true);
    Vertex *currVerAdr, *nextVerAdr;
    VertexNode currentVertexNode(&currentVer),
               nextVertexNode(&nextVer), *nodeAdr;
    LinkedList *adjacentList = new LinkedList;
    ifstream f in(fileName);
    if (!f.is_open()) {
        cout << "Cannot open " << fileName << endl;
        exit(-1);
    }
    while (!f.eof()) {
        f >> noskipws; // skip leading spaces;
        if (f.peek()) // spaces at the end of file;
```

Figure 8.39 (continued)

```
        Locals:
    for (i = 0; name[i] != ':'; i++)
        Locals[i] = Things[i];
    for (i++; isspace(name[i]); i++); // discard trailing spaces
    Locals[i] = '\0';
    x = new char[strlen(name)+1];
    strcpy(x,name);
    newNode->nameNode.idNum = x;
    newNode.forward = false;
    if (prevLen = membersTree.search(newNode->nameNode) == 0) {
        newNode->id.idNum = newNode->nameNode.idNum =
                            numOfVertices++;
        membersTreeNode.adjacent->push_front(Vertex(x,0,0,name));
        membersTree.insert(newNode);
        prevPerAdr = &membersTreeNode.adjacent->begin();
        newNode->adjacent.newLen(prevPerAdr);
        newNode->adjacent = new IntOneWay;
    }
    else {
        newNode->id.idNum = newNode->id.idNum;
        member->adjacent->push_front(prevPer);
        prevPerAdr = &member->adjacent->begin();
    }
    newNode->id.forward = true;
    committeeTreeNode.adjacent->push_front(newNode);
    newNodeAdr = &committeeTreeNode.adjacent->begin();
    newNodeAdr->next = prevPerAdr;
```

FIGURE 8.39 (continued)

```
            commVecAndr->twin = memberVecAndr;
        }
        commVec.forward = true;
        sourceList->push_front(commVec);
        commTreeTree.insert(commTreeTreeNode);
        commTreeTreeNode.adjacent = new List<Vertex>;
    }
    fin.close();
    cout << "\nCommittee tree:\n"; commTreeTree.inorderView();
    cout << "\nMember trees\n";       memberTree.inorderView();
    vertices = verticesPtr = new VertexArrayRec[numOfVertices];
    if (vertices == 0) {
        cerr << "Not enough memory\n";
        exit(-1);
    }
    vertices[source].inName = "source";
    vertices[sink].inName   = "sink";
    vertices[source].adjacent = sourceList;
    vertices[source].parent = NONE;
    commTreeTree.inorder(); // transfer data from both trees
    memberTree.inorder();   // to array vertices[];
}

void Network::label(Vertex *u, int v) {
    vertices[u->idNum].labeled = true;
    if (u->forward)
        vertices[u->idNum].vertexStack =
            min(vertices[v].vertexStack,edgeStack(u));
    else vertices[u->idNum].vertexStack =
            min(vertices[v].vertexStack,u->edgeFlow);
    vertices[u->idNum].parent  = v;
    vertices[u->idNum].corrVer = u;
}

void Network::augmentPath() {
    register int i, sinkStack = vertices[sink].vertexStack;
    Stack<int> path;
    for (i = sink; i != source; i = vertices[i].parent) {
        path.push(vertices[i].idName);
        if (vertices[i].corrVer->forward)
```

FIGURE 8.39 (continued)

```
            vertices[i].currVar->edgeFlow -= minSlack;
        else vertices[i].currVar->edgeFlow += minSlack;
        if (vertices[i].parent != source && i != sink)
            vertices[i].currVar->residualEdgeFlow =
               vertices[i].currVar->edgeFlow;
    }
    for (i = 0; i < numOfVertices; i++)
        vertices[i].labeled = false;
    cout << "  augment";
    while (!path.empty())
        cout << " => " << path.pop();
    cout << " !augmented by " << minSlack << "!\n";
}

void Network::findMaxFlow() {
    register int i, v;
    Stack<int> labeled;
    Vertex *u;
    list<Vertex>::iterator it;
    for (i = 0; i < numOfVertices; i++) {
        vertices[i].labeled = false;
        vertices[i].vertexSlack = 0;
        vertices[i].parent = none;
    }
    vertices[source].vertexSlack = INT_MAX;
    labeled.push(source);
    cout << "Augmenting paths:\n";
    while (!labeled.empty()) {   // while not empty
        v = labeled.pop();
        for (it = vertices[v].adjacent->begin(), u = &*it;
             it != vertices[v].adjacent->end(); it++, u = &*it)
            if (!labeled[u]) {
                if (u->forward && edgeSlack(u) > 0)
                    labeled.push(u);
                else if (!u->forward && u->edgeFlow > 0)
                    labeled.push(u);
                if (labeled[u])
                    if (u->index == sink) {
                        augmentPath();
                        while (!labeled.empty())
```

FIGURE 8.39 (continued)

```
                    labeled.pop();    // clear the stack;
                    labeled.push(source);// look for another path;
                    break;
                }
                else {
                    labeled.push(w->vertex);
                    vertices[w->vertex].labeled = true;
                }
            }
        }
    }

void main(int argc, char* argv[]) {
    char filename[30] = "g.in";
    ifstream ant;
    if (argc != 2) {
        cout << "Enter a file name: ";
        cin.getline(filename,'0');
    }
    else strcpy(filename,argv[1]);
    ant.open(filename);
    cout << name;
    ant.getline(name,MAX);
    cout << name;
}
```

8.12 EXERCISES

1. Look carefully at the definition of a graph. In one respect, graphs are more specific than trees. What is it?
2. What is the relationship between the sum of the degrees of all vertices and the number of edges of graph $G = (V, E)$?
3. What is the complexity of the `DELETE` operation?
4. Show that a simple graph is connected if it has a spanning tree.
5. Show that a tree with n vertices has $n - 1$ edges.

6. How can `DijkstraShortestPath` be applied to undirected graphs?

7. How can `DijkstraShortestPath` be modified to become an algorithm for finding the shortest path from vertex one to *N*?

8. The last statement from `computeShortestAllPairs`:

   ```
   addEdge(vertexOfInterest[j], k, w);
   ```

   is not included in `DijkstraShortestPath`. Can this omission cause any problem?

9. Modify `floydAlgorithm` so that it does not fall into an infinite loop if applied to a graph with negative cycles.

10. For what digraph does the `while` loop of `floydAlgorithm` iterate only one time? Two times?

11. Can `floydAlgorithm` be applied to undirected graphs?

12. Make necessary changes in `floydAlgorithm` to adapt it to solving the all-to-one shortest path problem and apply the new algorithm to vertex five for the graph in Figure 8.55. Using the same order of edges, produce a table similar to the table shown in this figure.

13. The Dantzig-Pape algorithm is erroneous in the worst case. Consider the following method to construct pathological graphs of *n* vertices (Kershenbaum 1981), each vertex identified by a number $1, \ldots, n$:

    ```
    KershenbaumAnalyze(int n)
        allocate a two-dimensional graph with vertices 1 and 3, and edge(1, 3) = 1;
        for (k = 4; k <= n; k++)
            add vertex k;
            for (i = 2; i < k; i++)
                add edge(k, i) with weight: edge(k, i-1) = weight: edge(1, i-1);
                edge(edge(1, i-1) = weight: edge(1, i-1) - 2^(k-i-1) + 1;
            add edge(1, k) with weight: edge(1, k) = 1;
    ```

    The vertices adjacent to vertex 1 are put in ascending order and the remaining adjacency lists are in descending order. Using this algorithm, construct a five-vertex graph and execute the Dantzig-Pape algorithm showing all changes in the distance and *P* edge updates. What generalization can you make about applying Pape's method to such graphs?

14. What do you need to change in your solution to `floydAlgorithm` in order to convert it to Dijkstra's one-to-all algorithm?

15. Enhance `floydAlgorithm` to indicate the shortest paths, in addition to their lengths.

16. `floydAlgorithm` fails to execute gracefully even in the presence of a single cycle. How do we know that the graph contains such a cycle?

17. The original implementation of `floydAlgorithm` given by Floyd was as follows:

```
Weight[][] shortestWeight;
   for (i = 1 to |V|
      for (j = 1 to |V|
         if (weight[j,i] < ∞
            for k = 1 to |V|
               if (weight[i,k] < ∞
                  if (weight[j][k] > weight[j][i] + weight[i][k])
                     weight[j][k] = weight[j][i] + weight[i][k];
```

Is there any advantage to the longer implementation?

18. Dijkstra's method of finding shortest paths from a vertex to all other vertices can be used to transform the graph so that it does not contain negative weights. We simply let k be the smallest negative weight and add $-k$ to the weights of all edges. Why is this method inapplicable?

19. For which edges does Euler's inequality

$$\text{dist}(v) \leq \text{dist}(w) + \text{weight}(\text{edge}(w,v)) \text{ for any vertex } w$$

become =?

20. Modify `cycleDetectionDFS()` so that it could determine whether a particular edge is part of a cycle in an undirected graph.

21. When would `BrokenDijkstraAlgorithm()` require $|V|$ iterations?

22. Our implementation of `union()` recycles three arrays. Is it possible to use only two of them and still have the same information concerning pools, test vertices, and lengths? Consider using negative numbers.

23. How can the second minimum spanning tree be found?

24. Is the minimum spanning tree unique?

25. How can the algorithm for finding the minimum spanning tree be used to find the maximum spanning tree?

26. The algorithm `blockSearch()`, when used for undirected graphs, relies on the following observation: In a depth-first search tree created for an undirected graph, each back edge connects a successor to a predecessor (and not, for instance, a sibling to a sibling). Show the validity of this observation.

27. What is the complexity of `blockSearch()`?

28. Edges in undirected graphs are defined in terms of edges, and the algorithm `blockDFS()` stores edges on the stack to output blocks. On the other hand, SCCs in digraphs are defined in terms of vertices, and the algorithm `strongDFS()` stores vertices on the stack to output SCCs. Why?

29. Consider a possible implementation of `copyGraphBad()` by using in it the following routine

```
findAnyVertexOfDigraph(
    v = a vertex of digraph;
    while (v has a successor)
        v = successor(v);
    return v;
```

What is the disadvantage of using this implementation?

30. A tournament is a digraph in which there is exactly one edge between every two vertices.

a. How many edges does a tournament have?

b. How many different tournaments of n edges can be created?

c. Is each tournament topologically sortable?

d. How many minimal vertices can a tournament have?

e. A transitive tournament is a tournament which has edge (v, x) if it has edges (v, w) and (w, x). Can such a tournament have a cycle?

31. Does considering loops and parallel edges complicate the analysis of networks? How about multiple sources and sinks?

32. `findMaxPathAugmentation()` assumes that it terminates. Do you think such an assumption is safe?

33. `findMaxPathAugmentation()` is coded in a depth-first fashion that has some redundancy. First all outgoing edges are pushed onto the stack, and then the last is popped off to be followed in the algorithm. For example, in the network in Figure 8.29a, first all three edges coming out of vertex `s` are pushed, and then `sA` is followed. Modify `findMaxPathAugmentation()` so that the first edge coming out of a certain vertex is immediately followed, and the second is followed only if the first does not lead to the sink. Consider using recursion.

34. Find the capacity of the cut determined by the set $X = \{s, A\}$ in the graph in Figure 8.29a.

35. What is the complexity of `DinicsAlgorithm()` on a network where all edges have a capacity of one?

36. Why does `DinicsAlgorithm()` start over from `s` to determine a layered network?

37. The member function `readCommitteeSet()` in the case study uses two lists, `committeeTree` and `memberTree`, to generate adjacency lists and then initialize the array vertices. However, one tree would be sufficient. What do you think is the reason for using two trees, not one?

38. What would be the output of the program in the case study if linked lists in Figure 8.38b were in the reverse order?

8.13 Programming Assignments

1. All algorithms discussed in this chapter for determining the minimum spanning tree have one thing in common: They start building the tree from the beginning and they add new edges to the structure which eventually becomes a tree. However, we can go in the opposite direction and build this tree by successively removing edges to break cycles in the graph until no circuit is left. In this way, the graph turns into a tree. The edges chosen for removal should be the edges of maximum weight, among those which can break any cycle in the tree (for example, Dijkstra's method). This algorithm somewhat resembles the Kruskal method, but, since it works in the opposite direction, it can be called a Kruskal method *à rebours*. Use this approach to find the minimum spanning tree for the graph of distances between at least 8 European cities.

2. Write a graphics demonstrative program to show the difference between Kruskal's method and Jarník-Prim's algorithm. Randomly generate 50 vertices and display them in the left half of the screen. Then, randomly generate 200 edges and display them. Make sure that the graph is connected. After the graph is ready, create the minimum spanning tree using Kruskal's method and display each edge included in the tree. Use a different color than the one used during graph generation. Then display the same graph in the right half of the screen, create the minimum spanning tree using Jarník-Prim's algorithm, and display all edges being included in the tree.

3. An important problem in database management is preventing deadlocks between transactions. A transaction is a sequence of operations performed in the database. In large databases, many transactions can be executed at the same time. This can lead to inconsistencies if the order of executing operations is not monitored. However, this monitoring may cause transactions to block each other, thereby causing a deadlock. To detect a deadlock, a wait for graph is constructed to show which transaction waits for which. Use a binary locking mechanism to implement a wait for graph. In this mechanism, if a record R is accessed by a transaction T_i, then T_i puts a lock on R and this record cannot be processed by any other transaction before T_i is finished. Release all locks put on by a transaction T_i when T_i finishes. The input is composed of the following commands: read(T_i, A), write(T_i, A), end(T_i). For example, if input is

$$\text{read}(T_2, A_1), \text{read}(T_3, A_2), \text{read}(T_1, A_2), \text{write}(T_1, A_2), \text{end}(T_1) \ldots$$

then T_1 is suspended when attempting to execute the step read(T_1, A_2), and edge $T_1 T_3$ is created, since T_1 waits for T_3 to finish. If T_3 does not have to wait, resume its execution. After each graph update, check for a cycle in the graph. If a cycle is detected, interrupt execution of the youngest transaction T and put its steps at the end of the input.

Note that some records might have been modified by such a transaction, so they should be restored to their state before T started. But such a modification could have been used by another transaction which should also be interrupted. In this program, do not address the problem of restoring the values of records (this problem of rolling back transactions and of cascading this roll-up back), Concentrate on operating the

monitoring the wait-for graph. Note that if a transaction is finished, its vertex should be removed from the graph, which may be what other transactions are waiting for.

4. Write a rudimentary spreadsheet program. Display a grid of cells with columns A through H and rows 1 through 20. Accept input in the first row of the screen. The commands are of the form *cell=num* or *cell=exp*, where *exp* is a number, a cell address preceded by a plus (e.g., +A3), a string, or a function preceded by an at sign (@). The functions are *sum*, *min*, *avg*, and *sum*. During execution of your program, build and modify the graph reflecting the situation in the spreadsheet. Show the proper values in proper cells. If a value in a cell is updated, then the values in all cells dependent on it should also be modified. For example, after the following sequence of entries,

```
A1: 10
B1: 20
A2: 30
C1: +A1
C2: @sum(A1..B2)
D1: +C1
```

both cells C1 and D1 should display the number 60.

Consider using a modification of the interpreter from Chapter 5 as an enhancement of this spreadsheet so that simple arithmetic expressions could also be used to enter values, such as

```
D3: 2*A1
D4: @max(A1..B2) - (A2 + 7)
```

Bibliography

Ahuja, Ravindra K., Magnanti, Thomas L., and Orlin, James B., *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Bellmore, Dennis P., "A Simple and Fast Label Correcting Algorithm for Shortest Paths," *Networks* 23 (1993), 703–709.

Bondy, J. A., and Chvátal, V., "A Method in Graph Theory," *Discrete Mathematics* 15 (1976), 111–135.

Bondy, John A., and Murty, U. S. R., *Graph Theory with Applications*. New York: Elsevier, 1976.

Chvátal, V., "Hamiltonian Cycles," in Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., and Shmoys, D. B. (eds.), *The Traveling Salesman Problem*. New York: Wiley, 1985, 403–429.

Dan, Ravindra, and Fang, Chieyin, "Shortest Path Algorithms: Taxonomy and Annotation," *Networks* 14 (1984), 275–323.

Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik* 1 (1959), 269–271.

Dijkstra, E. W., "Some Theorems on Spanning Subtrees of a Graph," *Indagationes Mathematicae* 28 (1960), 196–199.

Dinic, E. A., "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation" [Mistranslated work with Polynomial Bound], *Soviet Mathematics Doklady* 11 (1970), 1277–1280.

Edmonds, Jack. "Edge-Trees and Flowers," *Canadian Journal of Mathematics* 17 (1965), 449–467.

Edmonds, Jack and Johnson, Ellis L., "Matching, Euler Tours and the Chinese Postman," *Mathematical Programming* 5 (1973), 88–124.

Edmonds, Jack and Karp, Richard M., "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the ACM* 19 (1972), 248–264.

Floyd, Robert W., "Algorithm 97: Shortest Path," *Communications of the ACM* 5 (1962), 345.

Ford, L. R. and Fulkerson, D. R., "Maximal Flow Through a Network," *Canadian Journal of Mathematics* 8 (1956), 399–404.

Ford, L. R. and Fulkerson, D. R., "A Suggested Computation for Maximal Multi-Commodity Network Flows," *Management Science* 5 (1958), 97–101.

Ford, L. R. and Fulkerson, D. R., *Flows in Networks*, Princeton, NJ: Princeton University Press, 1962.

Gallo, Giorgio and Pallottino, Stefano, "Shortest Path Methods: A Unified Approach," *Mathematical Programming Study* 26 (1986), 38–64.

Gallo, Giorgio and Pallottino, Stefano, "Shortest Path Methods," *Annals of Operations Research* 7 (1988), 3–79.

Gibbons, Alan, *Algorithmic Graph Theory*, New York: Cambridge University Press, 1985.

Glover, Fred, Glover, Randy, and Klingman, Darwin, "Computational Study of an Improved Shortest Path Algorithm," *Networks* 14 (1984), 25–36.

Gondran, Michel and Minoux, Michel, *Graphs and Algorithms*, Chichester, UK: Wiley-Interscience, 1984.

Graham, R. L. and Hell, Pavol, "On the History of the Minimum Spanning Tree Problem," *Annals of the History of Computing* 7 (1985), 43–57.

Hall, Philip, "On Representatives of Subsets," *Journal of the London Mathematical Society* 10 (1935), 26–30.

Ingerman, P. Z., "Algorithm 141: Path Matrix," *Communications of the ACM* 5 (1962), 556.

Johnson, Donald B., "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM* 24 (1977), 1–13.

Kalaba, Robert, "On Some Communication Network Problems," *Combinatorial Analysis*, Providence, RI: American Mathematical Society 1960, 261–280.

Karzanov, Alexander, "A Note on Finding Shortest Path Trees," *Networks* 11 (1981), 399–400.

Kruskal, Joseph B., "On the Shortest Spanning Tree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society* 7 (1956), 48–50.

Kuhn, H. W., "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly* 2 (1955), 83–97.

Munkres, James, "Algorithms for the Assignment and Transportation Problems," Journal of the Society of Industrial and Applied Mathematics 5 (1957), 32–38.

Ore, Oystein, "Note on Hamilton Circuits," American Mathematical Monthly 67 (1960), 55.

Papadimitriou, Christos L., and Steiglitz, Kenneth, Combinatorial Optimization: Algorithms and Complexity, Englewood Cliffs, N.J.: Prentice-Hall, 1982.

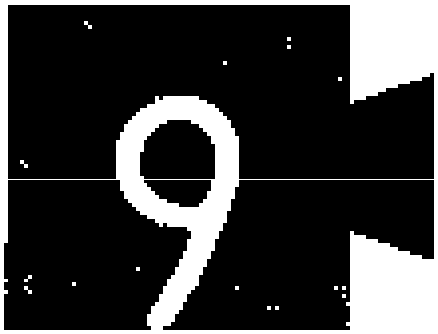
Rego, C., "Improved matching methods for the Traveling Salesman," Mathematical Programming 71 (1995), 223–322.

Tolbert, Maurice, and Widdoes, Warren, "Solutions for Several Route Problems—A Review," Operations Research 8 (1960), 224–230.

Tarjan, Robert E., Data Structures and Network Algorithms, Philadelphia: Society for Industrial and Applied Mathematics, 1983.

Trefethen, L., and Bau, D., Numerical Linear Algebra, New York: Wiley, 1997.

Warshall, Stephen, "A Theorem on Boolean Matrices," Journal of the ACM 9 (1962), 11–12.



Sorting

The efficiency of data handling can often be substantially increased if the data are sorted according to some criterion or order. For example, it would be practically impossible to find a name in the telephone directory if the names were not alphabetically ordered. The sorting can be on several criteria, for example, payrolls, bank accounts, students, and other alphabetical or numerical entries. The convenience of using sorted data is frequently realized in computer science as well. Although a computer can compare people with an unordered telephone book more easily and quickly than a human, it is sometimes inefficient to have the computer process such an unordered data set. It is often necessary to sort data before processing.

The first step is to choose the criteria which will be used to order the data. The choices will vary from application to application and must be defined by the user. Very often, the sorting criteria are numerical, as in the case of numbers. A set of numbers can be sorted in ascending or descending order. The set of five positive integers {5, 8, 1, 2, 9} can be sorted in ascending order resulting in the set {1, 2, 5, 8, 9} or in descending order, order resulting in the set {9, 8, 5, 2, 1}. Names in the phone book are ordered alphabetically by last name, which is the natural order for alphabetical and nonalphabetic characters. The ASCII code is commonly used, although other codes such as EBCDIC are possible. Once a criterion is selected, the second step is how to put a set of data in order using that criterion.

The final ordering of data can be obtained in a variety of ways, and only a few of them can be considered meaningful and efficient. To decide which method is best, certain criteria of efficiency have to be established and a method for quantitatively comparing different algorithms must be known.

To make the computer machine-independent, certain critical properties of sorting algorithms should be defined when comparing qualitative methods. Two such properties are the number of comparisons and the number of data movements. The choice of these two properties should not be surprising, for any set of data to be sor-

have to be compared and moved as necessary. For this family of these two operations depends on the size of the data set.

Since determining the precise number of comparisons is not always necessary or possible, an approximate value can be computed. For this reason, the number of comparisons and movements is approximated with big-O notation by giving the order of magnitude of these numbers. But the order of magnitude can vary depending on the initial ordering of data. How much time, for example, does the machine spend on data ordering if the data are already ordered? Does it recognize this initial ordering immediately or is it completely unaware of that fact? Hence, the efficiency measure also indicates the "intelligence" of the algorithm. For this reason, the number of comparisons and movements is computed (if possible) for the following three cases: best case (often, data already in order), worst case (usually, data in reverse order), and average case (data in random order). Some sorting methods perform the same operations regardless of the initial ordering of data. It is easy to measure the performance of such algorithms, but the performance level is usually not very good. Many other methods are more flexible and their performance measures for all three cases differ.

The number of comparisons and the number of movements do not always coincide. An algorithm can be very efficient in the former and perform poorly in the latter, or vice versa. Therefore, practical reasons must aid in the choice of which algorithm to use. For example, if only simple keys are compared, such as integers or characters, then the comparisons are relatively fast and inexpensive. If strings or arrays of numbers are compared, then the cost of comparisons goes up substantially, and the weight of the comparison measure becomes more important. If, on the other hand, the data items moved are large, such as structures, then the movement measure may stand out as the determining factor in efficiency considerations. All theoretically established measures have to be used with discretion, and theoretical considerations should be balanced with practical applications. After all, the practical applications serve as a rubber stamp for theory decisions.

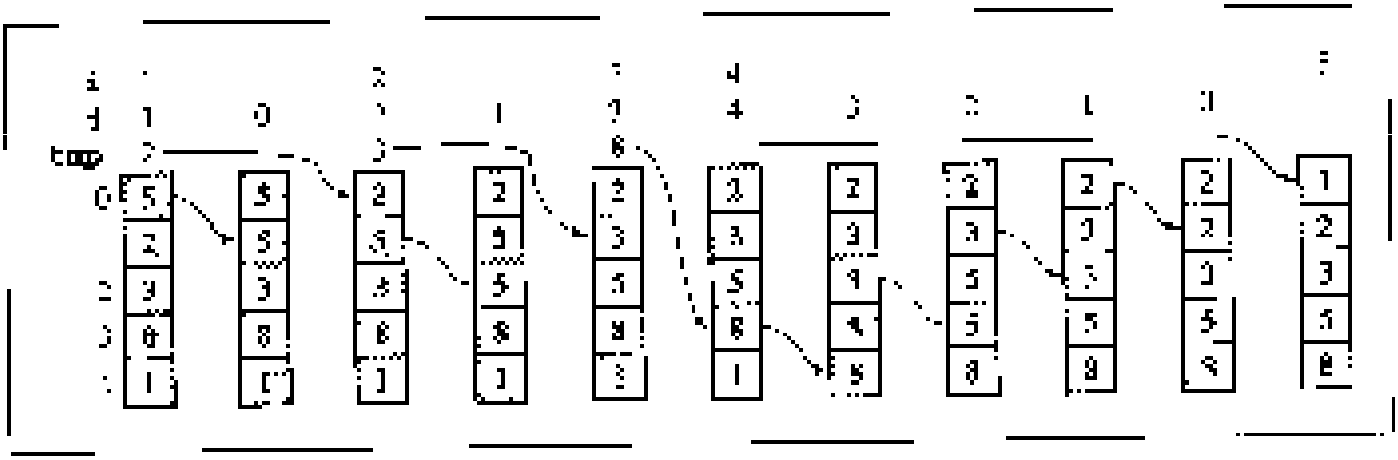
Sorting algorithms, whose number can be counted in the dozens, are of different levels of complexity. A simple method can be only 2–3 times less efficient than a more elaborate one. If sorting is used in the program once in a while and only for small sets of data, then using a sophisticated and slightly more efficient algorithm may not be desirable; the same operation can be performed using a simpler method and simpler code. But if thousands of items are to be sorted, then a gain of 20% must not be neglected. Simple algorithms often perform better with a small amount of data than their more complex counterparts whose effectiveness may only become obvious when data samples become very large.

9.1 Elementary Sorting Algorithms

9.1.1 Insertion Sort

An insertion sort starts by considering the two first elements of the array data, which are data[0] and data[1]. If they are out of order, an interchange takes place. Then,

FIGURE 9.1 The array [5 2 3 6 1] sorted by insertion sort.



the third element, data[2], is considered and inserted into its proper place. If data[2] is less than data[0] and data[1], then two elements are shifted by one position; data[0] is placed in position 1, data[1] in position 2, and data[2] in position 0. If data[2] is less than data[1] and not less than data[0], then only data[1] is moved to position 2 and its place is taken by data[2]. If finally data[2] is not less than both its predecessors, it stays in its current position. Each element data[i] is inserted into its proper location j such that $0 \le j \le i$ and all elements greater than data[i] are moved by one position.

An outline of the insertion sort algorithm is as follows:

```
insertionSort(data[],n)
   for(i = 1; i < n; i++)
      move all elements data[j] greater than data[i] by one position;
      place data[i] in its proper position;
```

Note that sorting is restricted only to a fraction of the array in each iteration, and only in the last pass is the whole array considered. Figure 9.1 shows what changes are made to the array [5 2 3 6 1] when insertionSort() executes.

Since an array having only one element is already ordered, the algorithm starts sorting from the second position, position 1. Then for each element temp = data[i], all elements greater than temp are copied to the next position, and temp is put in its proper place.

An implementation of insertion sort is:

```
template<class T>
void insertionSort(T data[], int n) {
    for (int i = 1; i < n; i++) {
        T temp = data[i];
        for (j = i; j > 0 && temp < data[j-1]; j--)
            data[j] = data[j-1];
        data[j] = temp;
    }
}
```

An advantage of insertion sort is that it stops the array only when it is really necessary. If the array is already in order, no additional moves are performed; only the variable temp is initialized, and the value stored in it is moved back to the same position. The algorithm recognizes that part of the array is already sorted and stops execution accordingly. But remember only that the first few elements may already be in their proper positions as is exploited. Therefore, they cannot be moved from these positions and then later moved back. This happens to numbers 2 and 3 in the example in Figure 9.1. Another disadvantage is that if an element is being inserted, all elements greater than the one whose temp value has to be moved. Insertion is not localized and may require moving a significant number of elements. Considering that an element can be moved from its final position only to be placed there again later, the number of redundant moves can slow down execution substantially.

To find the number of comparisons and/or operations performed by `insertionSort()`, observe that the outer `for` loop always performs $n-1$ iterations. However, the number of elements (other than `data[i]`) to be moved by one position is not always the same.

The best case is when the data are already in order. Only one comparison is made for each position, so there are $n-1$ comparisons, which is $O(n)$, and $2(n-1)$ moves, all of them redundant.

The worst case is when the data are in reverse order. In this case, for each i, the item `data[i]` is less than every item `data[0]`, ..., `data[i-1]`, and each of them is moved by one position. For each iteration of the outer `for` loop, there are i comparisons and the total number of comparisons for all iterations of this loop is

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

The number of times the assignment in the inner `for` loop is executed can be computed using the same formula. The number of times `temp` is loaded and unloaded in the outer `for` loop is added to that, resulting in the total number of moves:

$$\frac{n(n-1)}{2} + 2(n-1) = \frac{n^2 - 3n}{2} + 2 = O(n^2)$$

Only extreme cases have been taken into consideration. What happens if the data are in random order? Is the sorting time closer to the time of the best case, $O(n)$, or to the worst case, $O(n^2)$? Or is it somewhere in between? The answer is not immediately evident and requires certain introductory computations. The outer `for` loop always executes $n-1$ times, but it is also necessary to determine the number of iterations for the inner loop.

For every iteration i of the outer `for` loop, the number of comparisons depends on how far away the item `data[i]` is from its proper position in the currently sorted subvector `data[0..i]`. If it is already in this position, only one test is performed that compares `data[i]` and `data[i-1]`. If it is one position away from its proper place, two comparisons are performed: `data[i]` is compared with `data[i-1]` and then with `data[i-2]`. Generally, if it is j positions away from its proper location, `data[i]` is compared with $j+1$ other elements. This means that, in iteration i of the outer `for` loop, there are either $1, 2, \ldots$ or i comparisons.

Under the assumption of equal probability of occupying any array cells, the average number of comparisons of `data[i]` with other elements during the iteration of the outer `for` loop can be computed by adding all the possible numbers of comparisons and dividing the sum by the number of such possibilities. The result is

$$\frac{1+2+\ldots+i}{i} = \frac{\frac{1}{2}i(i+1)}{i} = \frac{i+1}{2}$$

To obtain the average number of all comparisons, the computed expression has to be added for all i (for all iterations of the outer `for` loop), from 1 to $n-1$. The result is

$$\sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{\frac{1}{2}n(n-1)}{2} + \frac{1}{2}(n-1) = \frac{n^2+n-2}{4}$$

which is $O(n^2)$ and approximately one-half of the number of comparisons in the worst case.

By similar reasoning, we can establish that, in iteration i of the outer `for` loop, `data[i]` can be moved either $0, 1, \ldots$, or $i-1$ times; that is

$$\frac{0+1+\ldots+(i-1)}{i} = \frac{\frac{1}{2}i(i-1)}{i} = \frac{i-1}{2}$$

times plus two unconditional movements (to swap and from swap). Hence in all the iterations of the outer `for` loop we have, on the average,

$$\sum_{i=1}^{n-1}\left(\frac{i-1}{2}+2\right) = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1}\frac{3}{2} = \frac{\frac{1}{2}n(n-1)}{2} + \frac{3}{2}(n-1) = \frac{n^2+5n-6}{4}$$

movements, which is also $O(n^2)$.

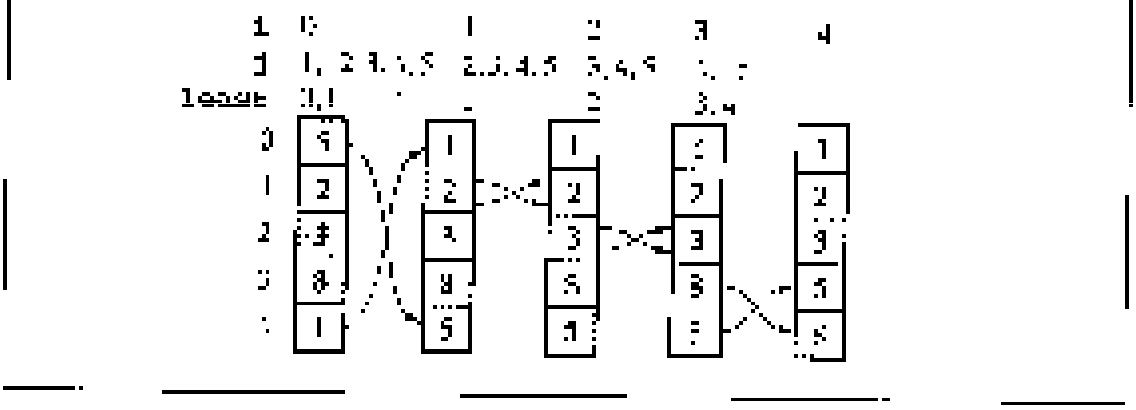
The answer to the question is the number of movements and comparisons for a randomly ordered array closer to the best or to the worst case? Unfortunately, it is closer to the latter, which becomes clear, on the average, when the size of an array is doubled; the sorting effort quadruples.

9.1.2 Selection Sort

Selection sort is an attempt to localize the exchanges of array elements by looking for misplaced elements first and putting it in its final place. The element with the lowest value is searched and exchanged with the element in the first position. Then, the next-to-lowest value among the remaining elements `data[1]`, ..., `data[n-1]` is found and put in the second position. This selection and placement by finding in each pass the lowest value among the elements `data[1]`, ..., `data[n-1]` and swapping it with `data[i]` are continued until all elements are in their proper positions. The following pseudocode reflects the simplicity of the algorithm:

```
selectionSort(data[], n)
   for(i = 0; i < n-1; i++)
      select the smallest element among data[i],...,data[n-1];
      swap it with data[i];
```

Figure 9.2. The array [5 2 3 5 1] sorted by selection sort.



It is rather obvious that $n-2$ should be the last value for i, since if all elements but the last one have already been transferred and placed in their proper position, then the nth element (occupying position $n-1$) has to be the largest. An example is shown in Figure 9.2. Here is a C++ implementation of selection sort:

```
template<class T>
void selectionSort(T data[], int n) {
    for (int i = 0, j, least; i < n-1; i++) {
        for (j = i+1, least = i; j < n; j++)
            if (data[j] < data[least])
                least = j;
        swap(data[least], data[i]);
    }
}
```

where the function `swap()` exchanges elements `data[least]` and `data[i]` (see the end of Section 1.2). Note that `least` is not the smallest element but its position.

The analysis of the performance of the function `selectionSort()` is simplified by the presence of two `for` loops with known end points. The outer loop executes $n-1$ times, and for each i between 0 and $n-2$, the inner loop iterates $j = (n-1)-i$ times. Because comparisons of keys occur in the inner loop, there are

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + \cdots + 1 = \frac{n(n-1)}{2} \quad O(n^2)$$

comparisons. This number stays the same for all cases. There can be some savings in the number of swaps. Note that if the assignment in the `if` statement is executed, only the index j is moved, not the item located currently at position j. An element is not swapped unconditionally in the outer loop as many times as this loop executes, which is $n-1$. Thus, in all cases, items are moved the same number of times, $3(n-1)$.

The best thing about this sort is the required number of assignments, which can hardly be beaten by any other algorithm. However, it might seem somewhat unsatisfactory

that the total number of exchanges, $3 \cdot (n-1)$, is the same for all cases. Obviously no exchange is needed if an item is in its final position. The algorithm disregards that and swaps such an item with itself, making three redundant moves. The problem can be alleviated by making `swap()` a conditional operation. The condition preceding the `swap()` should indicate that no item less than `data[leave]` has been found among elements `data[i+1]`, ..., `data[n-1]`. The last line of `selectionSort()` might be replaced by the lines:

```
if (data[i] != data[least])
   swap (data[local], data[i]);
```

This increases the number of array element comparisons by $n-1$, but this increase can be avoided by noting that there is no need to compare items. We proceed as we did in the case of the 10 assignment of `selectionSort()` by comparing the indexes and not the items. The last line of `selectionSort()` can be replaced by:

```
if (i != least)
   swap (data[least], data[i]);
```

Is such an improvement worth the price of introducing a new condition in the procedure and adding $n-1$ index comparisons as a consequence? It depends on what types of elements are being sorted. If the elements are numbers or characters, then interpreting a new condition to avoid execution of redundant swap statements is little inefficiency. But if the elements in `data` are large compound records such as employee structures, then one swap (which requires three assignments) may take the same amount of time as say 100 index comparisons, and nearly all the time `swap()` is executed.

9.1.3 Bubble Sort

A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column in which smaller elements are at the top and where larger elements are at the bottom. The array is scanned from the bottom up and two adjacent elements are interchanged if they are found to be out of order with respect to each other. First, `data[n-1]` and `data[n-2]` are compared and swapped if they are out of order. Next, `data[n-2]` and `data[n-3]` are compared, and their order is changed if necessary and so on up to `data[1]` and `data[0]`. In this way, the smallest element is bubbled up to the top of the array.

However, this is only the first pass through the array. The array is examined again comparing successive items and interchanging them as needed, but this time the last comparison is done for `data[2]` and `data[1]` because the smallest element is already in its proper position, namely, position 0. The second pass bubbles the second smallest element of the array up to the second position, position 1. The procedure continues until the last pass, which only has one comparison, `data[1]` with `data[0]`, and possibly one interchange are performed.

A pseudocode of the algorithm is as follows:

```
bubbleSort(data[],n)
    for (i = 0; i < n-1; i++)
        for (j = n-1; j > i; j--)
            swap elements in positions j and j-1 if they are out of order
```

...ure 9.3 The array [5 2 3 8 1] sorted by successive...

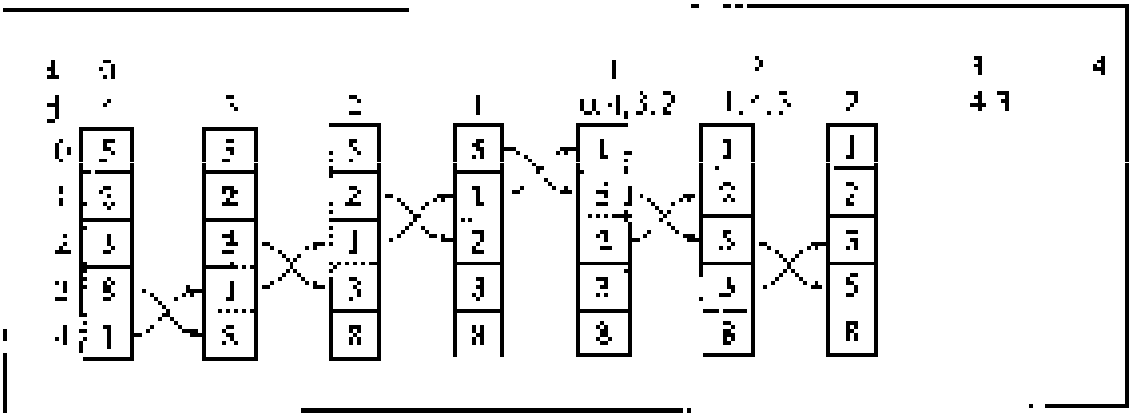


Figure 9.3 illustrates the changes performed in the integer array [5 2 3 8 1] during the execution of bubble sort(). Here is an implementation in C++:

```
template<class T>
void bubbleSort(T data[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; j--)
            if (data[j] < data[j-1])
                swap(data[j], data[j-1]);
}
```

The number of comparisons is the same in each case (best, average, and worst) and equals the total number of iterations of the inner for loop:

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This formula also computes the number of swaps in the worst case, which the array is in reverse order. In this case, $\frac{n(n-1)}{2}$ moves have to be made.

The best case, when all elements are already ordered, requires no swaps. If an input array is in random order, then the number of swaps can be any number between zero and $n-1$; that is, there can be either no swaps at all (if the array is sorted in order), one swap (two swaps, ..., or $n-1$ swaps. The array processed by the inner for loop is data[i], ..., data[n-1], and the number of swaps in this subarray—if its elements are randomly ordered—is either zero, one, two, ..., or $n-i-1$. After averaging the sum of all these possible numbers of swaps by the number of these possibilities, the average number of swaps is obtained, which is

$$\frac{0+1+2+\cdots+(n-1-i)}{n-i} = \frac{n-i-1}{2}$$

If all these averages for all the subarrays processed by bubbleSort() are added (that is, if such figures are summed over all iterations of the outer for loop), the result is

$$\sum_{i=1}^{n-1} \frac{n-i}{2} = \frac{1}{2}\sum_{i=1}^{n-1}(n-i) = \frac{1}{2}\sum_{i=1}^{n-1} i$$

$$= \frac{(n-1)}{2}\,\frac{(n-1+1)}{2} = \frac{n(n-1)}{4}$$

swaps, which is equal to $\frac{1}{4}n(n-1)$ moves.

The main disadvantage of bubble sort is that it still painstakingly bubbles items step by step up toward the top of the array. It looks at two adjacent array elements at a time and swaps them if they are not in order. If an element has to be moved from the bottom to the top, it is exchanged with every element in the array. It does not skip them as selection sort did. In addition, the algorithm concentrates only on the item that is being bubbled up. Therefore, all elements that disturb the order are moved, even those that are already in their final positions (see numbers 2 and 3 in Figure 9.5, the situation analogous to that in insertion sort).

What is bubble sort's performance in comparison to insertion and selection sort? In the average case, bubble sort makes approximately twice as many comparisons and the same number of moves as insertion sort, as many comparisons as selection sort, and n times more moves than selection sort.

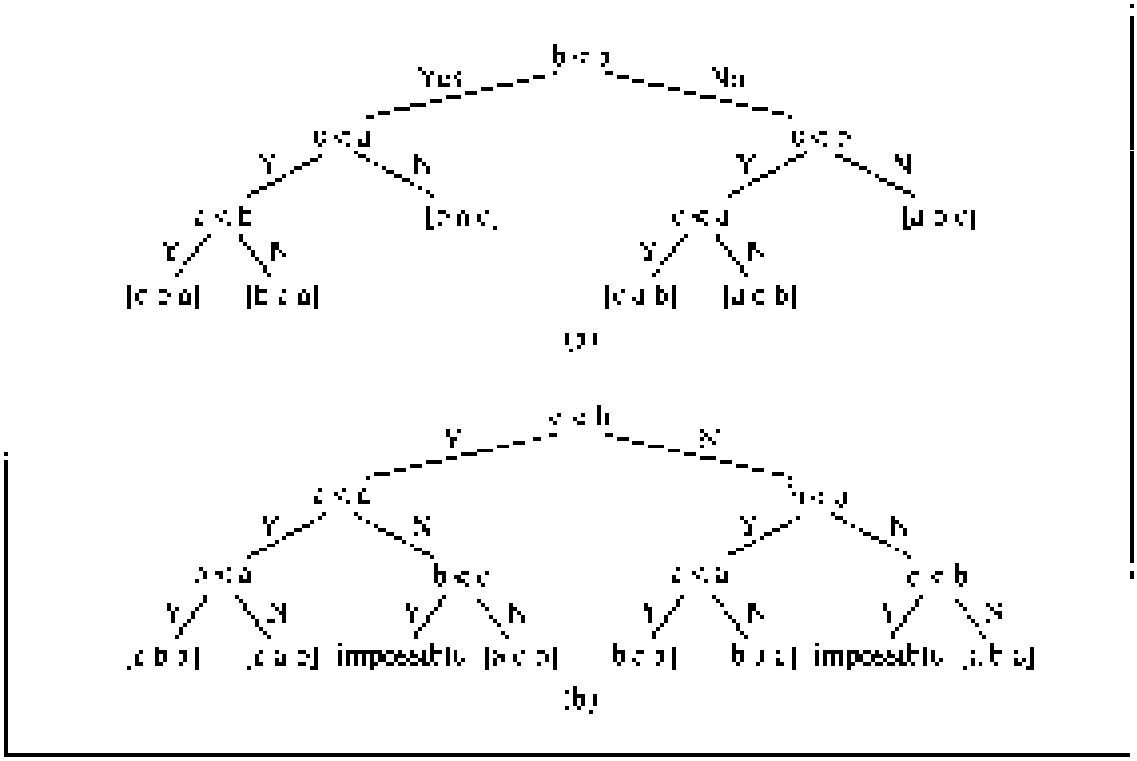
It could be said that insertion sort is twice as fast as bubble sort. In fact it is, but this fact does not immediately follow from the performance estimates. The point is that when determining a formula for the number of comparisons, only comparisons of data items have been included. The actual implementation for each algorithm involves more than just that. In `bubbleSort()`, for example, there are two loops, both of which compare indexes: `i` and `n-1` in the first loop, `j` and `i` in the second. All in all, there are $\frac{n^2-n}{2}$ such comparisons, and this number should not be treated too lightly. It becomes negligible if the data items are large structures. But if data consists of integers, then comparing the data takes a similar amount of time as comparing indexes. A more thorough treatment of the problem of efficiency should focus on more than just data comparison and exchange. It should also include the overhead necessary for implementation of the algorithm.

9.2 Decision Trees

The three sorting methods analyzed in previous sections were not very efficient. This leads to several questions: Can any better level of efficiency for a sorting algorithm be expected? Can algorithms, at least theoretically, be more efficient by executing faster? If so, when can we be satisfied with an algorithm and be sure that the sorting speed is unlikely to be increased? We need a quantitative measurement to estimate a lower bound of sorting speed.

This section focuses on the comparisons of two elements and not the element exchange. The questions are: On the average, how many comparisons have to be made to sort n elements? Or what is the best estimate of the number of item comparisons if an array is assumed to be ordered randomly?

Figure 9.4 Decision trees for (a) insertion sort and (b) bubble sort as applied to the array [a b c].



Every sorting algorithm can be expressed in terms of a binary tree in which the arcs carry the labels Yes or No, the internal nodes of the tree contain conditions or queries for labels, and the leaves have all possible orderings of the array to which the algorithm is applied. This type of tree is called a *decision tree*. Since the initial ordering cannot be predicted, all possibilities have to be listed in the tree in order for the sorting procedure to graph with any array and any possible initial order of data. This initial order determines which path is taken by the algorithm and what sequence of comparisons is actually chosen. Note that different trees have to be drawn for arrays of different length.

Figure 9.4 illustrates decision trees for insertion sort and bubble sort for an array [a b c]. The tree for insertion sort has six leaves, and the tree for bubble sort has eight leaves. How many leaves does a tree for an *n*-element array have? Such an array can be ordered in *n*! different ways, as many ways as the possible permutations of the array elements, and all of these orderings have to be stored in the leaves of the decision tree. Thus, the tree for insertion sort has six leaves because $3! = 6$, and $3! = 6$.

But as the example of the decision tree for bubble sort indicates, the number of leaves does not have to equal *n*!. In fact, it is never less than *n*!, which means that it can be greater than *n*!. This is a consequence of the fact that a decision tree can have leaves corresponding to failures, not only to possible orderings. The failure leaves are reached by an inconsistent sequence of operations. Also, the total number of leaves can be greater than *n*! because some orderings (permutations) can occur in more than one leaf, since the comparisons may be repeated.

Obviously the minimum property of any decision tree is that there must be enough leaves to handle all the possible permutations of the input. Since we have worked out the formula for the number of permutations of n elements, we know that a decision tree for sorting n elements must have at least $n!$ leaves.

As we have seen in Chapter 8, a tree of height h can have at most 2^h leaves. Therefore, we can conclude that $2^h \geq n!$, where h is the number of levels in the tree. Taking the logarithm of both sides, we get $h \geq \log(n!)$. This inequality answers the following question: How many comparisons are necessary in order to certainly sort n elements? The answer is at least $\log(n!)$ comparisons in the worst case. Or rather, at least the ceiling of $\log(n!)$ comparisons, since h must be an integer and cannot be less than the ceiling of $\log(n!)$.

Note that this analysis pertains to the worst case. We assume that it is a tree of any shape, regardless of whether or not it is complete. h always refers to the longest path leading from the root of the tree to the lowest tree level, which is also the largest number of comparisons needed to reach an ordered configuration of the array stored in the tree. Finally, the inequality $h \geq \log(n!)$ is usually rewritten as $h \geq \lceil \log(n!) \rceil$, where $\lceil x \rceil$ is the integer ceiling function that means that the actual length of a decision tree with at least $n!$ leaves must be at least $\log(n!)$, or rather, the next integer greater than or equal to $\log(n!)$, where $\log(n!)$ is not an integer, as we see in the example in Figure 9.5.

It can be proven that for a number of chosen leaf or an individual decision tree, the length of the path from the root to the leaf is not less than $\log(n!)$ and that, both in the average case and the worst case, the minimal number of comparisons $\log(n!)$ is $\Omega(n \log n)$. See Section A.2 in Appendix A). That is, $\Omega(n \log n)$ is also the best that can be expected in average cases.

It is interesting to compare this approximation in terms of the numbers of computations for sorting methods, especially for the average and worst cases. For example, selection sort requires only n^2 comparisons in the best case, but in the average and the worst cases, this sorting algorithm is $O(n^2)$, since the functions relating the number of comparisons to the number of elements are, for these cases, the n^2. Obviously, n^2 is significantly greater than $n \log n$ for large numbers of n. Consequently, insertion sort is not an ideal algorithm. The quest for better methods can be continued, with at least the expectation that the number of comparisons could be approximated by a function smaller than $n \log n$.

The difference between these two functions is best seen in Figure 9.6. It ranges from performance of the algorithms in practice as compared with the expected performance in the average case. The numbers in the table in Figure 9.6 show exactly how many times faster than selection sort and bubble sort the Quicksort algorithm is, and how many times faster than bubble sort the Insertion sort is. For 10,000, the differences in performance are 1,666 times faster and 3,333 times faster, respectively. This can only serve to encourage the search for an algorithm that will further speed up the performance of the Quicksort algorithm.

Figure 9.5 Examples of decision trees for an array of three elements.

There are some possible decision trees for an array of three elements. Here there are just two cases, $3! = 6$ cases. For this take on the example is associated with each one has one external leaf (or partition) that is [illegible]. In [illegible] cases, the number of comparisons is [illegible]. In this example, $\log(3!) = 2.585$. And, in fact, only for the best balanced tree can the external path length to the average case is less than this.

Level

1

2

3

4

5

n

7

(a) (b) (c) (d)

The sum of the lengths of the paths from the root to all leaves for each of the trees in Figure 9.5 and the average path lengths:

(a) 1 + 2 + 3 + 3 = 9; average = $\frac{9}{3} = 3.0$

(b) 1 + 2 + 3 + 3 = 9; average = $\frac{9}{3} = 3$

(c) 1 + 2 + 3 + 3 + 3 + 3 = 15; average = $\frac{15}{6} = 2.5$

(d) 1 + 2 + 3 + 4 + 5 + 5 = 20; average = $\frac{20}{6} = 3.33$

9.3 Efficient Sorting Algorithms

9.3.1 Shell Sort

The algorithm for a sorting shell is much more complex and takes more time to program. Although it is not faster than the slower sorts below. The main point is that the time required for ordering an array by the shell sorting algorithm usually grows faster than the size of the array, or less, it is customarily a quadratic function of that

TABLE 9.6 Number of comparisons performed by the simple sorting method and an algorithm whose efficiency is estimated by the function $n \log_2 n$

sort type	n	100	1,000	10,000
worst	$\frac{n(n-1)}{2}$	4,950	499,500	49,995,000
average	$\frac{n(n-1)}{4}$	2,475	249,750	24,997,500
expected	$n \log_2 n$	664	9,966	132,877

size. It may turn out to be more efficient to sort parts of the original array and then merge it, or it may at least partly be ordered. To sort the entire array, if the two arrays already sorted, we can do that in much closer to the best case of an ordered array. A simple version of such a procedure is as follows:

```
single data from subarray1
for (i = 1; i <= by; i++)
    add element to subarray;
subarray sorted;
```

In practice, each of the subarrays data of more elements could be best sorted by the appropriate sorting algorithm. A high-level algorithm will be the other hand. It is to be expected that the subarrays will be very small and almost ordered. Hence, the overall order of cases. Lastly, it only a small part of data is out of place in the execution time may be reduced. To solve the problem, several different subdivisions are used, and for every subdivision, the procedure is applied to the elements as in:

```
determine number of subarrays and allocate memory for subarrays
for each subarray
    divide data into subarrays
    for (i = 1; i <= number of subarrays; i++)
        sort subarray data;
combine sorted data;
```

This idea is the basis of the Shellsort algorithm, also known as Shell sort or diminishing increment sort, developed by Donald L. Shell who designed this technique. Note that this pseudocode does not identify a specific sorting method for ordering the subarrays; it can be any sound method. Usually, however, Shellsort uses insertion sort.

The heart of Shellsort is an ingenious division of the array data into several subarrays. The trick is that the elements spaced far apart are compared first, then the elements closer to each other are compared, and so on, until adjacent elements are compared on the last pass. The original array is logically subdivided into subarrays so

picking every pth element to produce a subarray. These are then sorted by a distinct method for every $k = 0, \ldots, p-1$.

$$data_{p,k}[i] = data[k + i \cdot p]$$

In our example $p = 3$. The three subarrays are defined and shown as $data_{3,0}$, $data_{3,1}$, and $data_{3,2}$:

$data_{3,0}[0] = data[0] = 25$, $data_{3,0}[1] = data[3] = 36$, $data_{3,0}[2] = data[6] = 14$, ...
$data_{3,1}[0] = data[1] = 57$, $data_{3,1}[1] = data[4] = 48$, $data_{3,1}[2] = data[7] = 12$, ...
$data_{3,2}[0] = data[2] = 39$, $data_{3,2}[1] = data[5] = 86$, $data_{3,2}[2] = data[8] = 33$, ...

and so on. Each subarray is sorted separately by the insertion sort algorithm with $h_1 = 1$ and the resulting sorted subarrays replace the original ones. The process is repeated with a new p value smaller than the previous one, h_2 in the previous example, and the entire process is called a *pass*.

Figure 9.5 shows the contents of the array after the first pass. Note that the elements are nearly (but not completely) in order. Typically, the physically last pass of the algorithm is a regular insertion sort with $p = 1$, but the values of the previous passes have the property that the number of steps the insertion sort algorithm takes is relatively small. The choice of the h_k values is somewhat arbitrary, with the only restriction being that $h_1 = 1$. Figure 9.6 shows the result of the second pass ($p = 3$) and the sorted array after the last pass.

There is still room for improvement in this algorithm. For example, we could choose a better set of h_k values. In fact, Figure 9.5 actually shows that the array is almost sorted after the second pass. Only three elements must be moved to put the array in order. Intuitively, it seems that if h_k is small relative to n, then the array is relatively sorted. Thus, the insertion sort algorithm should run quickly when h_k is small. Unfortunately, the converse is not true. Although the last pass always requires a full insertion sort, Shell sort is often quite efficient in practice. Its worst-case performance is $O(n^2)$, but the exact order of Shell sort is still unknown. Sedgewick has proposed several sequences of h_k values that have $O(n^{4/3})$ worst-case performance, and other sequences are known to have $O(n^{3/2})$ worst-case performance. Although these results are of considerable theoretical interest, the actual performance of Shell sort is sensitive to small changes in the sequence. Empirical studies have shown that sequences with $O(n^{3/2})$ worst-case performance usually perform worse than the $O(n^2)$ sequences for all but very large values of n. For example, the sequence $h_k = 2^k - 1$ has $O(n^{3/2})$ worst-case performance, but it rarely outperforms the sequence $h_k = 3h_{k-1} + 1$ with $h_1 = 1$, which has $O(n^2)$ worst-case performance.

Empirical studies have shown that Shell sort is competitive with more complex algorithms for sorts of moderate size. It is also a good algorithm to use if memory space is a serious consideration, because it sorts in place and requires only a small amount of code. Shell sort is also a good algorithm to use if the input is almost sorted, because it will perform only the last pass. Although Shell sort is rarely the fastest algorithm possible, it is fast enough for many applications and is much easier to implement than the more efficient algorithms we will discuss next.

The following code segment implements Shell sort, using the sequence mentioned previously ($h_1 = 1$, $h_{k+1} = 3h_k + 1$):

```
h = 1;
while (h < data.length / 3)
    h = 3 * h + 1;
while (h > 0)
{
```

Figure 9.7 The array [10 6 20 4 3 22 1 0 15 16] sorted by shell sort.

Initial before sort:	10	6	20	4	3	22	1	0	15	16	
Five subarrays before sorting:	10	6	20	4	3	22	1	0	15	16	
Five subarrays after sorting:	1	6	15	0	3	10	1	0	20	16	
Initial after 5-sort and before 3-sort:	1	6	15	0	3	10	1	0	20	16	
Three subarrays after 3-sorting:	1	6	15	0	3	10	1	0	20	16	
Initial after 3-sort and before 1-sort:	1	6	15	0	3	10	1	0	20	16	
Final after 1-sort:	0	1	3	6	10	15	16	20	22	24	

sort stop with i_1 for which P_{i_1+2} is. For $n = 10{,}000$, this gives the sequence

1, 4, 13, 40, 121, 364, 1093, ..., 9841.

Experimental runs have been performed by the experimental function, the estimate, $1.2 n^{1.25}$, and the logarithmic function $n(\ln n)^2$. The first term fits the results of the test better. For $n = 10{,}000$, $n^{1.25}$ is much greater than the expected (average) performance.

Figure 9.8 contains a function to sort the array data using shell sort. Note that before sorting starts, increment is set to zero and stored in the array incs[ ].

The core of shell sort is a double-for array that subarrays by making elements h positions apart. There are three features of this algorithm that vary from one implementation to another:

1. The sequence of increments
2. A simple sorting algorithm applied in all passes except the last
3. A simple sorting algorithm applied in the final pass, i = 1

Listing 9.8 Implementation of Shell sort.

```
template<class T>
void ShellSort(T data[], int arraySize)
{
    register int i, j, hCnt, k;
    int increments[20], k;
//  create an appropriate number of increments h
    for (k = 1, i = 0; k < arraySize; i++) {
        increments[i] = k;
        k = 3*k + 1;
    }
//  loop on the number of different increments h
    for (i--; i >= 0; i--) {
        k = increments[i];
//  sort all the subarrays created by this h
        for (hCnt = k; hCnt<2*k; hCnt++) {
//  insertion sort for subarray containing every kth element of
            for (i = hCnt; i < arraySize; ) {   // array data
                T tmp = data[i];
                k = i;
                while (j >= k && tmp < data[j-k]) {
                    data[j] = data[j-k];
                    j -= k;
                }
                data[j] = tmp;
                i += k;
            }
        }
    }
}
```

In our implementation of Shell sort, insertion sort is applied to a fixed number of increments that could be used. For example, the bubble sort for the last pass and the selection sort for each pass. Improved feedback could be used to avoid unnecessary sorting, but only for the worst case. All these versions perform better than simple insertion, although there are some differences in their performance. Analytical results concerning the complexity of these sorts are not available. All are considered to be some form of an $n \log n$ function.

9.3.2 Heap Sort

Selection sort makes $O(n^2)$ comparisons and is very inefficient, especially for large n. But it performs relatively few moves. If the comparison part of the algorithm is improved, the entire process can be greatly sped up.

Heap sort was invented by John W. J. Williams and uses the approach inherent in selection sort. Selection sort finds among the elements the one that precedes all of the $n-1$ elements, then finds the next one among those $n-2$ elements, and so forth, until the array is sorted. To have the entire unsorted list scanned to find the largest element at the end of the array, heap sort puts the largest element in front of it and then uses a heap sort starts from the end of the array by finding the largest elements, whereas selection sort starts from the beginning to find the smallest elements. The final results in both cases are indeed the same.

Heap sort uses a heap described in Section 9.2. A heap is a binary tree with the following two properties:

1. The value of each node is greater than or equal to the value of each of its children.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

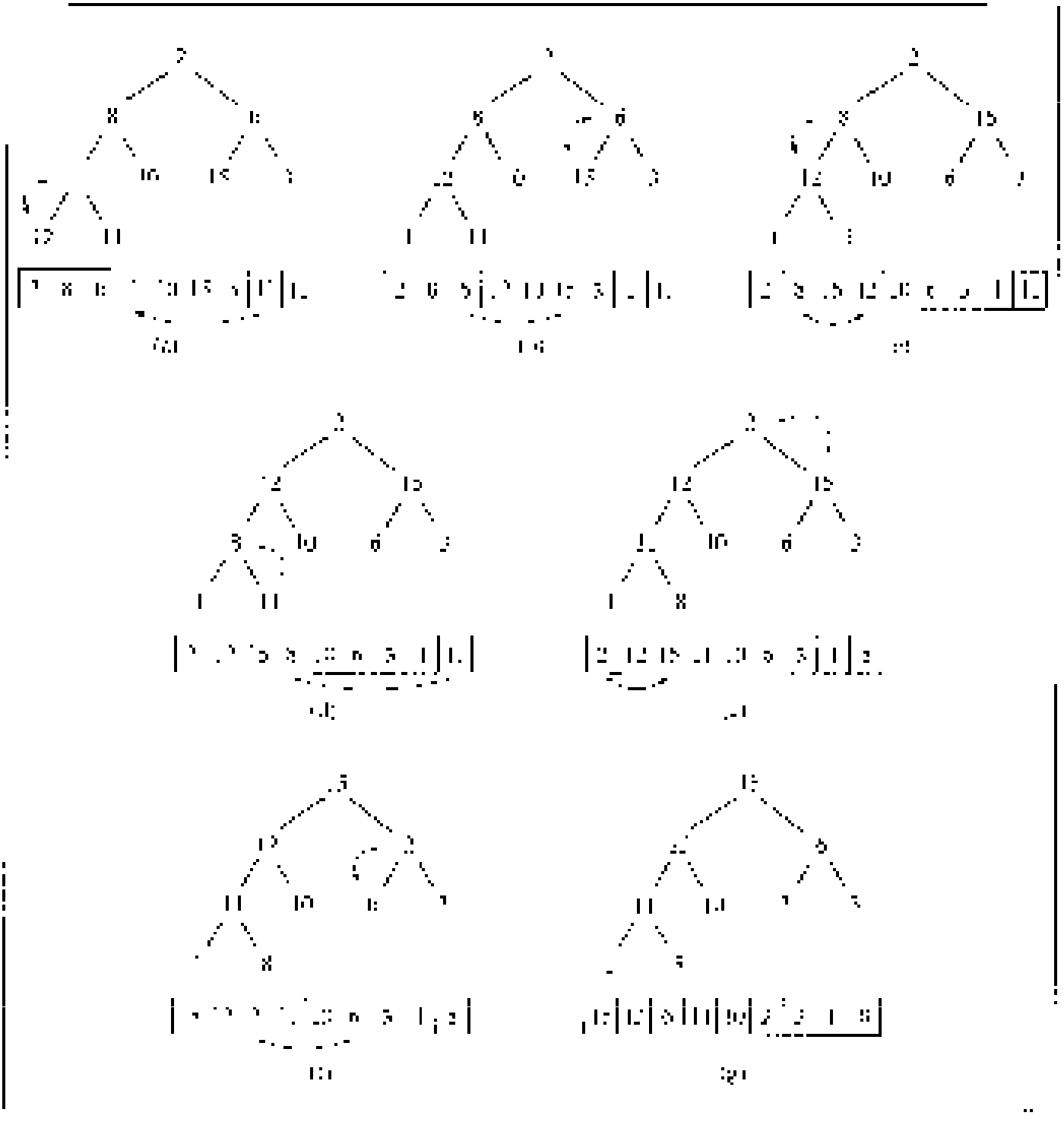
A tree has the heap property if it satisfies condition 1. Both conditions are useful for sorting, although this is not immediately apparent for the second condition. The reason is that only the array heap can be sorted without using additional storage for the array. Specifically, selection 2 puts elements into consecutive positions in the array starting from position 0, with no unused position inside the array. In other words, selection 2 forces the packing of an array with no gaps.

Elements in a heap are not perfectly ordered. It is only known that the largest element is in the root node and that for each other node, its descendants are not greater than the element in this node. Heap sort thus starts from the heap, puts the largest element at the end of the array, and restores the heap that now has one less element. From the new heap, the largest element is removed and put in its final position, and then the heap property is restored for the remaining elements. Thus, in each round, one element of the array ends up in its final position and the heap becomes smaller by one element. The process ends with evacuating all elements from the heap and is summarized in the following pseudocode:

```
heapSort(data[1..n])
    transform data into a heap
    for i = n to 2 by -1 do
        swap the root with the element in position i
        restore the heap property for data[1..i-1]
```

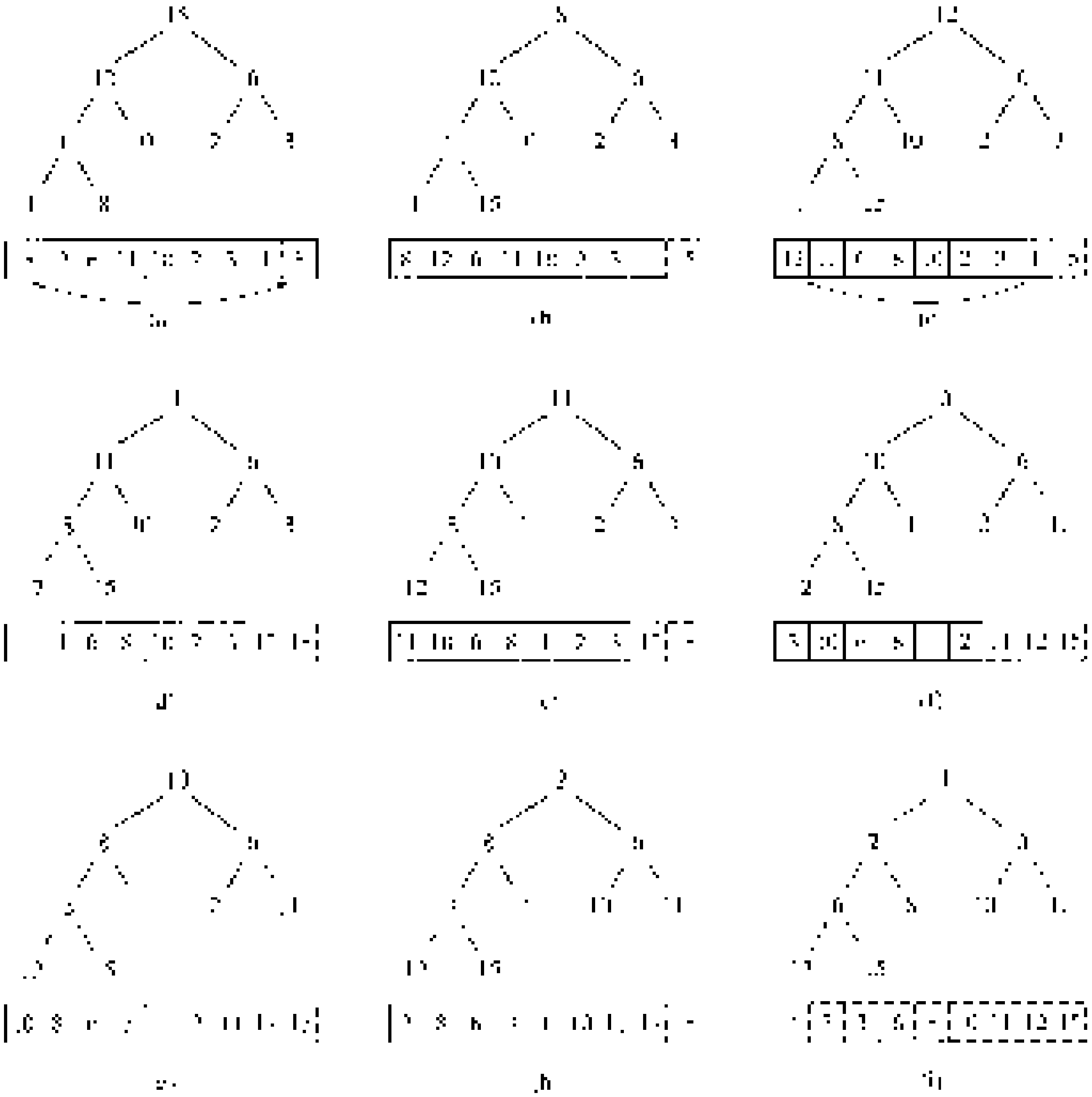
In the first phase of heap sort, an array is transformed into a heap. In this process we use a bottom-up method developed by Floyd and described in Section 9.2. All steps leading to the transformation of the array [4 6 1 15 7 14 11] into a heap are illustrated in Figure 9.5 (cf. Figure 9.4).

The second phase begins after the unsorted heap has the shape as in Figure 9.5g and is sorted. At that point, the largest element, number 15, is moved to the end of the array. In order to



taken by `fixDown()`, the loop terminates. The temporary has been restored, but the heap is not yet correct. In the next call to `fixDown()`, `k` is 1 and `N` is 10. The largest child of node 1 is node 2, and it is moved up to fill the hole left by node 1. Node 2 is then left with a hole, and `fixDown()` sets `a[2]` to the value of `a[4]`. Finally, `fixDown()` sets `a[4]` to the value of the temporary, and the heap is restored.

FIGURE 9.10 Execution of heapSort on the array [3 12 6 1 10 2 5 13], which is the heap constructed in Figure 9.9.



The method heapSort begins by calling buildHeap to construct a heap out of the elements in the array. The buildHeap method, which is called for the heap in Figure 9.10a, begins by calling percolateDown on the subtree rooted at index 3, giving the tree in Figure 9.10b. The percolateDown method is called again on the subtree rooted at index 2, and the

element, swapped with the last element of the current subarray which is 7 (Figure 9.10c). Now it is shifted down (Figure 9.10d) and swapped with 7 (Figure 9.10e). The resulting possible configuration is the result of the first pass through the loop of `heapsort()` after the first pass. The array is now partially ordered and the root is ordered correctly. A complete implementation of `heapsort()` is shown:

```
template<class T>
void heapsort(T data[], int size)
{
    for (int i = size/2 - 1; i >= 0; --i)   // create the heap:
        moveDown(data,i,size-1);
    for (int i = size-1; i >= 1; --i) {
        swap(data[0],data[i]); // move the largest item to data[i]
        moveDown(data,0,i-1);  // restore the heap property
    }
}
```

Heapsort might be considered inefficient because the naive algorithm for creating a heap requires two passes. First, `heapify` is applied to move the largest element to the left-most side of the array in order to move it to the right-most side. Further, this process is repeated for the first phase of heapsort. The `heapsort()` uses `moveDown()`, which performs $O(\log n)$ steps (see Section 9.2).

In the second phase, `heapsort()` exchanges the root with the element at the specified end and restores the heap property by sifting the new root node `moveDown()` to its rightful place. This is repeated for all elements of `moveDown()` in the second phase of `heapsort()` is $O(n \log n)$, which is the same as `heapsort()` as a whole. Thus, the second phase of `heapsort()` is the most time-consuming part of the algorithm. Since `heapify` operations to restore the heap property in the second phase can be in the worst case $O(\log n)$, the overall complexity for the second phase is in the worst case $O(n \log n)$.

For the best case, where the array contains identical elements, `heapify` is called $\frac{n}{2}$ times in the first phase, but no moves are performed. In the second phase, `heapsort()` makes one swap for each element of the root node to the end of the array, and `moveDown()` immediately returns, since the root node is the only node in the heap. Thus, the second phase is $O(n)$. Therefore, the best case for `heapsort()` is $O(n)$. However, if the array contains distinct elements, then in the best case no more than $n \log n$ comparisons are required (Wegener 1991).

9.3.3 Quicksort

Quicksort operates on the principle of recursively dividing the original array into subarrays, sorting the subarrays, and then dividing up the subarrays to sort the subarrays until the whole array is sorted. The process to select the appropriate pivot value to subdivide the array is the most crucial and time-consuming part of the algorithm. There are many approaches to selecting the pivot, but the most commonly used is the median-of-three method (Sedgewick 1988).

The original array is divided into two subarrays: the first one which contains elements less than or equal to the pivot value, and the second one which contains elements greater than or equal to the pivot. The two subarrays are sorted

separately, just before this is done the partitioning process is repeated for both subarrays. As a result, two new bounds are chosen, one for each subarray. The four subarrays are created because each subarray obtained in the first phase is now divided into two segments. This process of partitioning is carried down until there are only nine-element arrays that do not need to be sorted any further. Dividing the task of sorting a large array into two smaller tasks and then dividing those tasks into even smaller tasks is characteristic of the process of getting prepared to sort; the sort may have already been started. Since the sorting has been somewhat completed in the preparation process, this process is the core of quicksort.

Quicksort is recursive in nature because it is applied to both subarrays of an array at each level of partitioning. The technique is summarized in the following pseudocode:

```
quicksort(array[]):
   if length(array) > 1
      choose bound; // partition array into subarray1 and subarray2
      while more elements left in array
            if element1 in subarray1 < bound1:
            if element2 in subarray2 > bound2:
      quicksort(subarray1);
      quicksort(subarray2);
```

A partitioning operation has to be performed. A bound has to be found and the array has to be scanned to place the elements in the proper subarrays. However, choosing a good bound is not a trivial task. The problem is that the subarrays should be approximately the same length if an array contains the numbers 1 through 100 in any order and 2 is chosen as the bound, then in most circumstances the first subarray contains only one number after partitioning, whereas the second has 99 numbers.

A number of different strategies for selecting a bound have been developed. One of the simplest consists of choosing the first element of an array. This approach can suffice for some applications. However, since many arrays will be sorted already, too many elements in the proper positions will result. A more successful approach is to choose the element located in the middle of the array. This approach is incorporated in the implementation in Figure 9.4.

Another task is scanning the array and dividing up the elements between the two subarrays. The procedure is to position a variable at each end of the array. Each variable does not stop when it finds an element equal to the bound. It only stops when it finds elements that are placed in the first subarray if they are less than or the same as the bound and in the second if they are greater than or equal to the bound. The reason is that the difference between the length and number of zeros should be minimal. Therefore, elements equal to the bound are divided between the two subarrays to make the difference as small as possible. The decision of whether to stop at equal elements and move them to the other subarray is given in Figure 9.4. In this implementation, a variable called `pivot` is used to keep track of the bound, which is the largest element in the array and is the last element of the array. Having the largest element as the bound ensures that the index `lower` can never run off the end of the array. This can happen if the first element in the last subarray is the largest element in the array. The index `upper` would then never be decremented.

Program 9.11 Implementation of quicksort

```
template<class Item>
void quicksort(Item data[], int first, int last) {
    int lower = first+1, upper = last;
    swap(data[first],data[(first+last)/2]);
    Item bound = data[first];
    while (lower <= upper) {
        while (data[lower] < bound)
            lower++;
        while (bound < data[upper])
            upper--;
        if (lower < upper)
            swap(data[lower++],data[upper--]);
        else lower++;
    }
    swap(data[upper],data[first]);
    if (first < upper-1)
        quicksort(data,first,upper-1);
    if (upper+1 < last)
        quicksort(data,upper+1,last);
}

template<class Item>
void quicksort(Item data[], int n) {
    if (n < 2)
        return;
    for (int i = 1, max = 0; i < n; i++)// find the largest
        if (data[max] < data[i])        // element and put it
            max = i;                    // at the end of data[];
    swap(data[n-1],data[max]); // largest is now in its
    quicksort(data,0,n-2);     // final position;
}
```

inconvenience eventually causing an abnormal program termination. Without this preprocessing, the first inner while loop would have to be

```
while (lower < last && data[lower] < bound);
```

The first test, however, would be necessary only in extreme cases, but it would be executed in each iteration of this while loop.

In the current implementation, the main property of the bound is used, namely, that it is a particular item. Hence, as before, the boundary item is placed on the borderline between

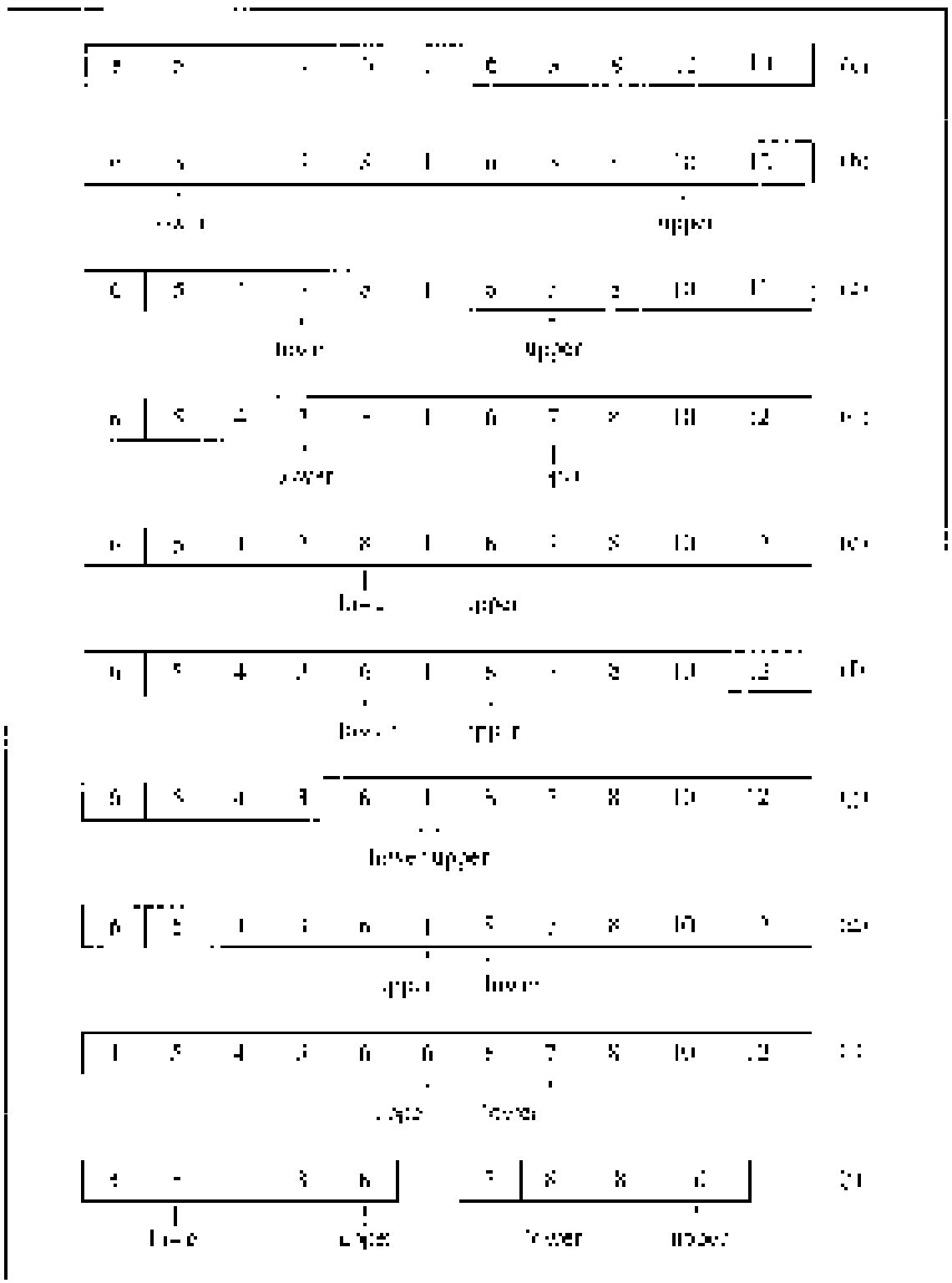
the two subarrays obtained as a result of one call to quicksort. In this way the pivot is locked in its final position and can be excluded from further processing. To ensure that the pivot is not moved around it is stashed in the first position after partitioning is done, it is moved to its proper position, which is the rightmost position of the first subarray.

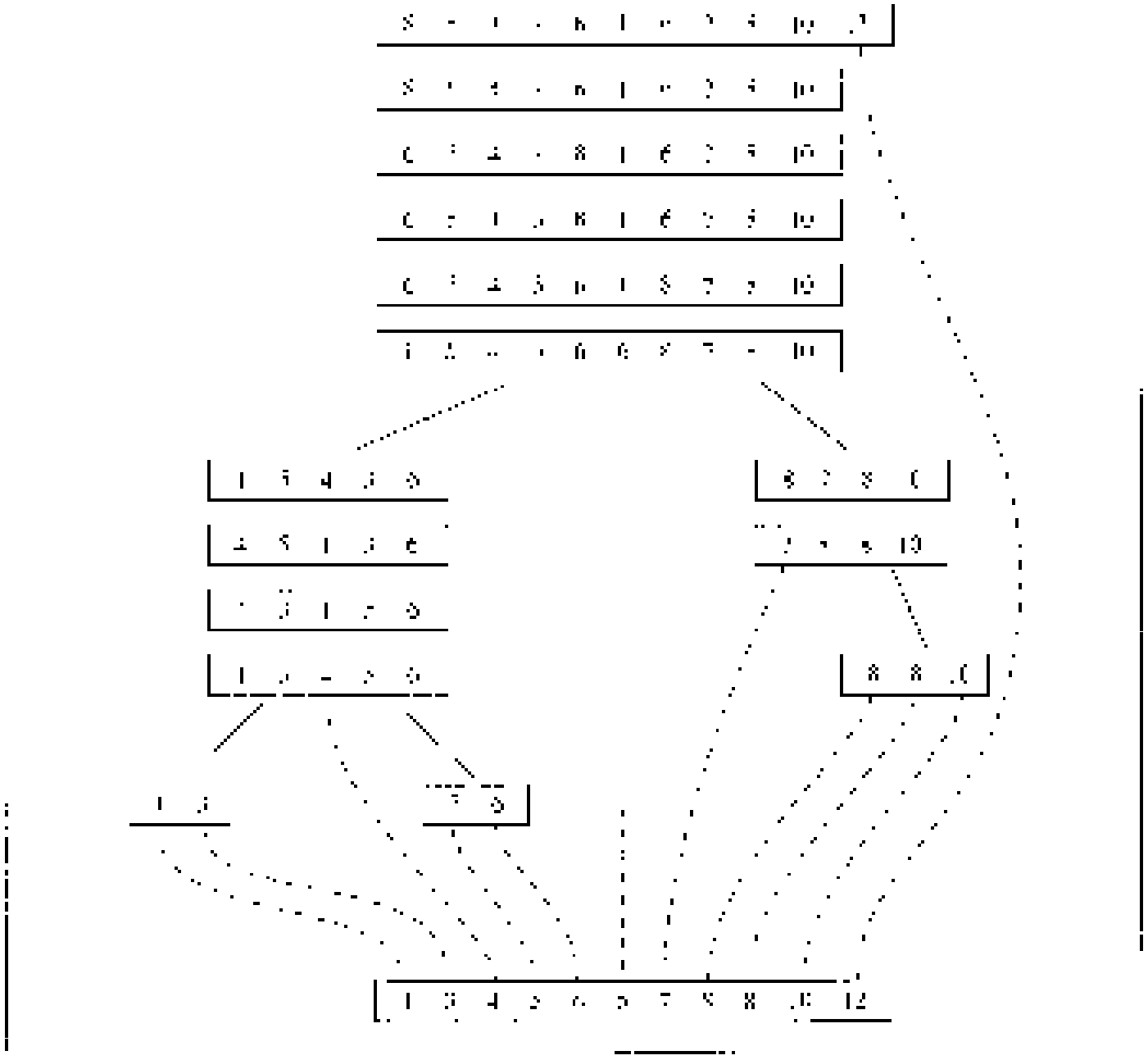
Figure 9.12 contains an example of partitioning the array [6 5 1 9 0 3 8 2]. In the first partitioning, the largest element in the array is located and exchanged with the last element, resulting in the array [6 5 1 2 0 3 8 9]. Because the last element is already in its final position, it does not have to be processed anymore. Therefore, in the first partitioning, lower = 0, upper = 6, and the first element of the array, 6, is exchanged with the second 0 in position 6, so that the array is [0 5 1 2 0 3 8 9] (Figure 9.12a). In the first iteration of the outer while loop, the inner while loop moves lower to position 3 with a 2, which is greater than the bound. The second inner while loop moves upper to position 2 with a 0, which is less than the bound (Figure 9.12b). Next, the elements in these two cells are exchanged, giving the array [0 5 1 2 0 3 8 9] (Figure 9.12c). Then lower is incremented to 4 and upper is decremented to 3 (Figure 9.12d). This concludes the first iteration of the outer while loop.

In the second iteration, neither of the two inner while loops modifies any of the two indexes because lower indicates a position occupied by 5, which is greater than the bound, and upper indicates a position occupied by 0, which is equal to the bound. The two numbers are exchanged (Figure 9.12e), and both indexes are updated to 5 (Figure 9.12g).

In the third iteration of the outer while loop, lower is moved to the next position containing 3, which is greater than the bound, and upper stays on the same position because the current position is less than the bound (Figure 9.12h). Since at this point lower and upper cross each other, no swapping takes place, and after a redundant exchange of lower with itself, the outer while loop is exited. At that point, upper is the index of the rightmost element of the first subarray, with the element not exceeding the bound, so the element in this position is exchanged with the bound (Figure 9.12i). In this way, the bound is placed in its final position and can be excluded from subsequent processing. Therefore, the two subarrays that are processed next are the left subarray, with elements to the left of the bound, and the right subarray, with elements to the right (Figure 9.12j). Then partitioning is performed for these two subarrays separately, and then for subarrays of these subarrays, until subarrays have less than two elements. The entire process is summarized in Figure 9.13, in which all the contiguous current arrays are isolated.

The worst case occurs if in each invocation of quicksort(), the smallest or largest element of the array is chosen for the bound. This is the case if we try to sort the array [5 3 1 2 4 6 8]. The first bound is 1, and the array is broken into an empty array and the array [3 5 2 4 6 8]. The largest number, 8, does not participate in partitioning. The next bound is 2, and again an empty array and [3 5 4 6] is obtained as the result of partitioning. The next bound and arrays obtained by partitioning are 3 and [5 4 6], then 4 and [5 6], and finally 5 and [6]. This algorithm thus operates on arrays of sizes n, $n-1$, $n-2$, ..., 2. The partitioning requires $n-1+n-2+\cdots+1$ comparisons, one for each position, only the bound is placed in the proper position. This results in a running time equal to $O(n^2)$, which is totally undesirable result, especially for large arrays or files.





The best case occurs when the algorithm divides the array into two subarrays of approximately equal size. If the bounds for both subarrays are well approximated by $\frac{n}{2}$, then the algorithm creates two subarrays, each of which is approximately $\frac{n}{2}$ elements. Therefore, the number of comparisons performed after k partitions is approximately equal to

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + \cdots + n\frac{n}{n} = n\log n + 1.$$

which is $O(n \log n)$. Intuitively, the $\log n$ comes from the fact that we can divide n by 2 only $\log n$ times before reaching 1, and the n comes from the fact that we have to do n comparisons to split an array of size n. Similar arguments show that the average case and the best case are also $O(n \log n)$. A quicksort is $O(n \log n)$ when the pivot is the median element of the array. This fact can be strengthened by noting that the expected number of comparisons done by quicksort on the arrays in Figure 9.13 is $1.38 n \log n$. Therefore, if we can find a way to select the median element of the array quickly, then we can implement a fast sorting algorithm. Unfortunately, finding the median element of an array is a difficult problem. It can be done in linear time, but the algorithm is complicated and slow in practice. Instead, we will look for a simple algorithm that finds a pivot element that is usually near the median.

The standard method for choosing a pivot is to choose one of the first, middle, and last elements of the array. But if the array is partially sorted, then these elements are likely to be poor choices. For example, if the array is already sorted, then this method will always choose the smallest element as the pivot. A better method is to choose three elements randomly from the array and use the median of these three elements as the pivot. Even though this method requires the generation of random numbers, it is worthwhile because it makes the worst case unlikely to occur. Sedgewick (1978) suggests choosing the pivot from the first, middle, and last elements of the array, but it is not clear that this is any better than choosing the pivot randomly.

A quicksort is a good sorting algorithm for general purpose use—usually it is not the worst case, and it can be implemented to be very fast. In the next section, we will see that we can improve the performance of quicksort even more by using insertion sort to sort small subarrays. Because insertion sort has a small constant factor, it can sort small arrays quickly. Since quicksort uses insertion sort, it should also be the method of choice for small subarrays. Therefore, a good implementation of quicksort should not call itself recursively when there are only a few elements in the array. Instead, it should call insertion sort. Because insertion sort is fast on nearly sorted arrays, quicksort can be extended to be a sort that is fast.

Second, it is important to use a quicksort that works well on small arrays. Even though quicksort has fewer instructions, insertion sort is more efficient for small arrays (Cook and Kim 1980). For this reason, the initial pseudocode can be changed to

```
quicksort2(array)
   if length(array) > 10
      partition array into subarray1 and subarray2;
      quicksort2(subarray1);
      quicksort2(subarray2);
   else insertionsort(array);
```

and the implementations changed accordingly. However, the table in Figure 9.18 indicates that the improvement is not significant.

9.3.4 Mergesort

[illegible] procedure mergesort is that it requires [illegible] of the two halves of the array. Suppose that we [illegible] sorted halves of the array. Then we could [illegible] sorted halves to produce a sorted version of the entire array. Is it possible to merge the two halves of the array in place? The answer is yes, but the algorithm is complex. We will describe a simpler algorithm that uses an auxiliary array for the merging process. The mergesort algorithm can be summarized in the following pseudocode:

```
mergesort(data):
    if data has more than one element
        mergesort(left half of data);
        mergesort(right half of data);
        merge(both halves using temporary);
```

Merging two subarrays into one is a relatively straightforward process, as indicated in the pseudocode:

```
merge(array1, array2, array3):
    i1, i2, i3 are, respectively, initialized;
    while both array2 and array3 have more elements:
        if array2[i2] < array3[i3]
            array1[i1++]  array2[i2++];
        else array1[i1++]  array3[i3++];
    Add the remaining elements from either array2 or array3;
```

For example, if array2 = [1 4 5 9 10] and array3 = [2 3 5 7], then the result is array1 = [1 2 3 4 5 5 7 9 10 22]. Note that the pseudocode for mergesort suggests that array1, array2, and array3 are physically separate entities. However, for the proper execution of mergesort, array1 is a concatenation of array2 and array3 so that array1 before the execution of merge is [1 4 5 9 10 2 3 5 7 22]. In this situation, merge() works because after the second iteration of the while loop array1 is [1 2 4 5 9 10 2 3 5 7] and array1 is [1 2 3 4 5 9 10 5 7 22]. Therefore, a temporary array is not needed for the merging process. At the end of the merging process, the contents of this temporary array are transferred to array1, because array2 and array3 are subarrays of array1. They do not need to be passed as parameters to merge(). Instead, indexes for the beginning and the end of array1 are passed, since array1 can be a part of another array. The new pseudocode is

```
merge(array1, first, last):
    mid    (first + last) / 2;
    i1    0;
```

```
    i2 = first;
    i3 = mid + 1;
    while both subarrays are nonempty, compare
       if array1[i2] < array1[i3]
            temp[i1++] = array1[i2++];
       else temp[i1++] = array1[i3++];
    Now one temp is empty, move remainder of array1
    Move to array1 remainder of temp
```

Since the entire array1 is copied to temp and then temp is copied back to array1, the number of movements in each execution of merge() is always twice the second argument, which is equal to 2 * (last – first + 1). The number of comparisons depends on the ordering in array1. If array1 is in order or if all the elements in the right half precede the elements in the left half, the number of comparisons is first – last. The worst case is when the last element of each subarray is greater than the other, such as in {1, 3, 10, 12} and {2, 9, 11, 15}. In this case, the number of comparisons is last – first. For an n-element array, the number of comparisons is $n - 1$.

The pseudocode for mergesort() is as follows:

```
mergesort (data, first, last)
   if first < last
      mid = (last - last) / 2;
      mergesort(data, first, mid);
      mergesort(data, mid+1, last);
      merge(data, first, last);
```

Figure 8.21 illustrates an example using this sorting algorithm. This procedure can be used to analyze the computing time for mergesort. For an n-element array, the number of movements is computed by the following recurrence relation:

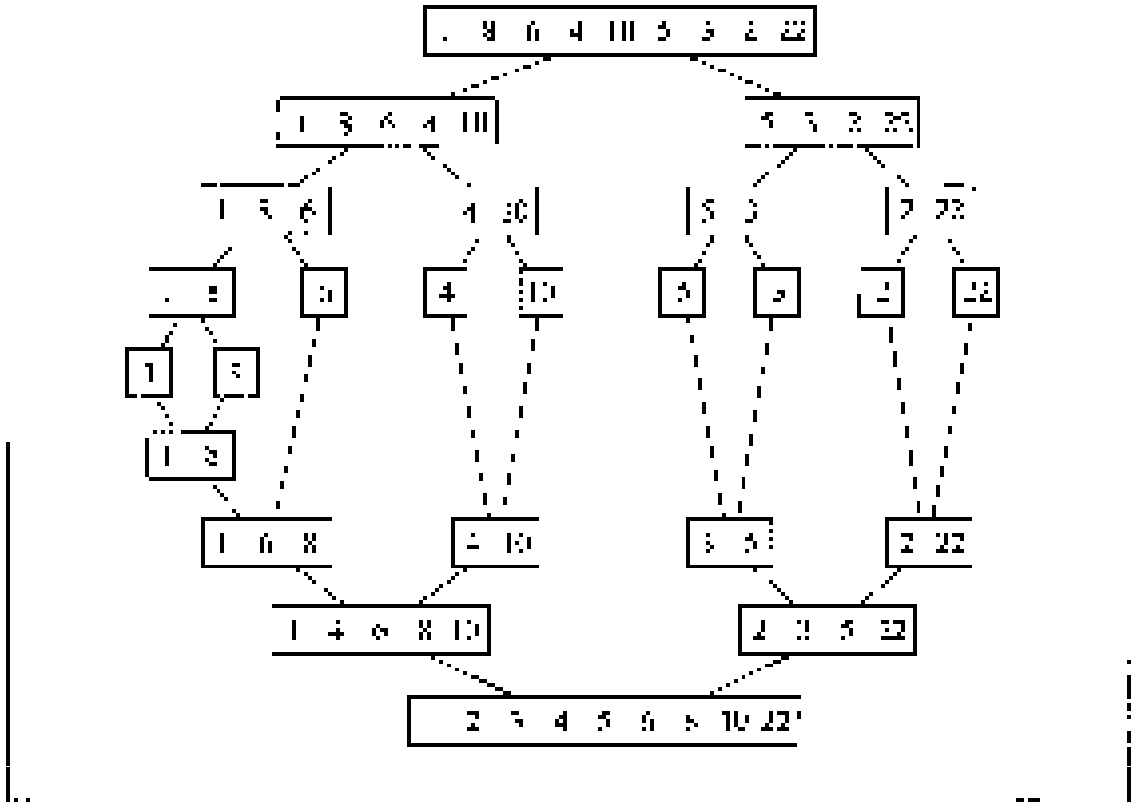
$$M(1) = 0$$
$$M(n) = 2M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2n$$

$M(n)$ can be computed in the following way:

$$M(n) = 2\left[2M\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 2\left\lfloor \frac{n}{2} \right\rfloor\right] + 2n = 4M\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 4n$$
$$= 4\left[2M\left(\left\lfloor \frac{n}{8} \right\rfloor\right) + 2\left\lfloor \frac{n}{4} \right\rfloor\right] + 4n = 8M\left(\left\lfloor \frac{n}{8} \right\rfloor\right) + 4n$$
$$\vdots$$
$$2^k M\left(\left\lfloor \frac{n}{2^k} \right\rfloor\right) + 2kn$$

Choose $k = \lg n$ so that $2^k = n$. This gives us $M(n) = nM(1) + 2n \lg n = 2n \lg n = O(n \log n)$.

Figure 9.14 The array 18 64 10 33 22 sorted by merge.



The worst-case running time of mergesort, expressed by a recurrence relation, is

$$T(1) = 1$$

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

which can be solved by various means. We saw the solution earlier in the context of top-down recursive functions—the analysis of the selection problem also applies here.

Although mergesort's running time is $O(N \log N)$, it is rarely used for main memory sorts. The main problem is merging two sorted lists together. To do this merge, we need an extra array. The merge routine is subtle if you want to do it efficiently—recursion helps here. The code we have written takes advantage that the last element in the array is larger than any element in the array, so there are no special cases. We have also provided a nonrecursive implementation of mergesort. Nonetheless, mergesort's use for main memory sorting has been largely supplanted by more modern algorithms. We will discuss one of the algorithms in this section. If external sorting is necessary, mergesort is a good candidate, and we will discuss external sorting in Section 9.5.

9.3.5 Radix Sort

Suppose you have a penchant for punching cards and have a deck of cards that you need to sort. Each card has a number between 1 and 10 on it. One simple algorithm that you might consider is to make 10 passes over the deck of cards. On the first pass, you remove all the cards that have a 1 punched in them, in order. On the second pass, you remove all the cards that have a 2 punched in them, in order. You repeat this for each of the digits 3 through 9, and the result is a sorted deck. The amount of time that this algorithm takes is proportional to $10N$, where N is the number of cards in the deck. If you have a large deck of cards, then this algorithm is reasonably fast. If you have only a few cards, then it is probably not worthwhile.

the same method, namely sales, are treated according to the second letter of the author's name. For instance, the number of books sold by Smith, John is listed in a smaller font size than the number of books sold by Smith, Mary. This method was used in the early days of the post office, and it was used to sort 80 million records after each census until the early 1960s.

When sorting libraries of books, we process from left to right. The method we have used for sorting integers and floats above will not work for strings. However, we can use the same algorithm for strings if we provide a function that compares two strings and returns true if the first string is less than the second string. For example, "apple" is less than "banana" because 'a' comes before 'b' in the alphabet. Similarly, "apple" is less than "apples" because a string is always less than a longer string that starts with the shorter string. Case is important as well. "Zebra" is less than "apple" because all uppercase letters are less than all lowercase letters. Still another way to sort integers is to process them digit by digit. This method is discussed shortly. Still another way to sort integers is to process them digit by digit. This method is discussed shortly.

When sorting integers, we pick a pivot value and then create two new lists, one for values less than the pivot and one for values greater than the pivot. For example, if we pick 5 as the pivot in the list [3, 7, 1, 9, 4, 6, 2, 8], we create two new lists: [3, 1, 4, 2] and [7, 9, 6, 8]. We then recursively sort these two lists and combine the results to get the sorted list [1, 2, 3, 4, 5, 6, 7, 8, 9]. The pivot value is usually the first value in the list, but it can be any value in the list. The algorithm is described in the following pseudocode:

```
function quicksort(list)
    if length(list) <= 1 then return list
    pivot = list[0]
    left = []
    right = []
    for i = 1 to length(list) - 1 do
        if list[i] < pivot then append list[i] to left
        else append list[i] to right
    return concatenate(quicksort(left), pivot, quicksort(right))
```

The key to choosing the pivot value is to choose a value that is close to the middle of the list. If we choose a value that is close to the beginning or end of the list, then the two new lists will be very unbalanced, and the algorithm will be very slow. For example, if we choose 1 as the pivot in the list [3, 7, 1, 9, 4, 6, 2, 8], then the two new lists will be [] and [3, 7, 9, 4, 6, 2, 8]. The first list is empty, and the second list is almost as large as the original list. This is called a worst-case scenario, and it is to be avoided if possible. The algorithm is described in the following pseudocode:

```
pivot = list[0]
pivot = list[length(list) - 1]
```

These pivots are not very good. In the first case, the pivot is the first value in the list, and in the second case, the pivot is the last value in the list. Both of these pivots will result in very unbalanced lists. A better pivot is the median of the first, middle, and last values in the list. The median of three values is the value that is neither the largest nor the smallest. For example, the median of 3, 7, and 1 is 3, because 3 is neither the largest nor the smallest. The median of 9, 4, and 6 is 6, because 6 is neither the largest nor the smallest. The median of 2, 8, and 5 is 5, because 5 is neither the largest nor the smallest. The median of 1, 9, and 4 is 4, because 4 is neither the largest nor the smallest. The median of 6, 2, and 8 is 6, because 6 is neither the largest nor the smallest. The median of 5, 1, and 9 is 5, because 5 is neither the largest nor the smallest. The median of 4, 6, and 2 is 4, because 4 is neither the largest nor the smallest. The median of 8, 5, and 1 is 5, because 5

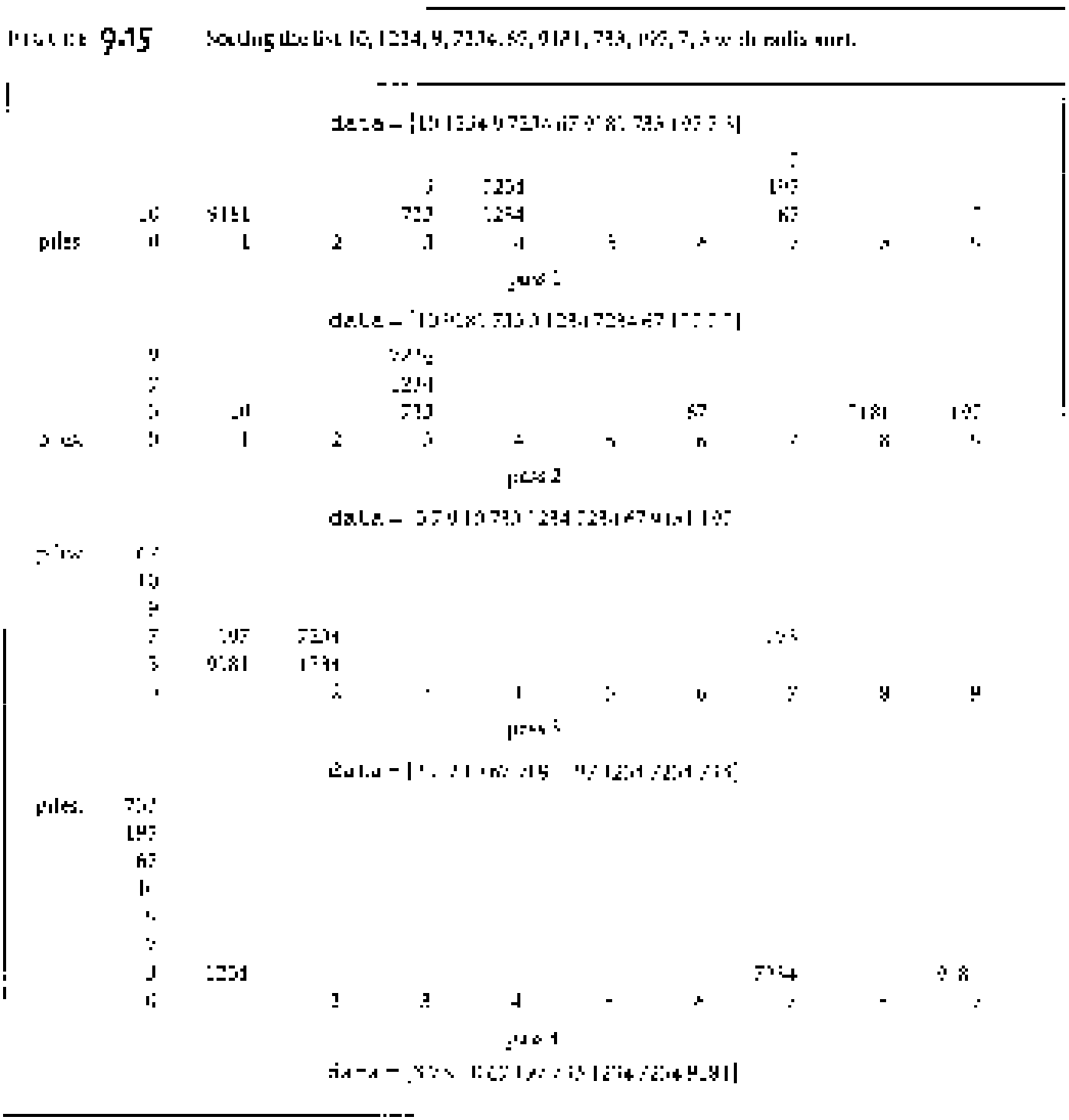


Figure 9.15 Sorting the list 10, 1234, 9, 7234, 67, 9181, 733, 199, 7, 3 with radix sort.

Listing 9.4 An implementation of radix sort

```
void radixsort(long data[], int n) {
    register int i, j, k, factor;
    const int radix = 10;
    const int digits = 10; // the maximum number of digits for a long
    Queue<long> queues[radix];   // auxiliary
    for (i = 0, factor = 1; i < digits; factor *= radix, i++) {
```

```
        for (j = 0; j < n; j++)
            a[count[(data[j] / factor) % radix]++] = queues[i].dequeue().data[j];
        for (j = k = 0; j < radix; j++)
            while (!queues[j].empty())
                data[k++] = queues[j].dequeue();
    }
}
```

This algorithm is clearly an O(n) algorithm. Each pass requires n enqueue operations and n dequeue operations, for a total of 2n operations per pass. The number of passes is determined by the number of digits in the largest key. If the largest key is a 6-digit number, then 6 passes are required. Thus, the total number of operations is 12n. Since the number of operations is directly proportional to n, the algorithm is O(n).

Unfortunately, this algorithm requires a lot of space. In addition to the array of n elements, it requires space for 10 queues, each of which can grow to size n. Thus, the algorithm requires space for 11n elements. One way to reduce the space requirement is to use linked lists instead of queues. Each list requires only enough space for the number of elements in the list. Since the sum of the sizes of the lists cannot exceed n, the total space for the lists cannot exceed n. Thus, the space requirement is reduced to 2n. The following algorithm implements this idea:

```
void bucketSort(long data[], int n) {
    register int i, j, k, pmax = 1;
    int[] count = new int[10];
    Queue[] queues = new Queue[2];
    for (i = 0; i < 2; i++)
        queues[i] = new Queue(n, data);
    for (j = 1; j <= pmax; j *= 10) {
        for (i = 0; i < 10; i++)
            count[i] = 0;
        while (!queues[0].empty())
            count[(queues[0].front() / j) % 10]++;
        for (i = 1; i < 10; i++)
            count[i] += count[i-1];
        while (!queues[0].empty()) {
            i = (queues[0].front() / j) % 10;
            queues[1].enqueue(queues[0].dequeue());
            if (count[i] > pmax)
                pmax = count[i];
        }
        for (i = 0; i < 10; i++)
            count[i] = count[i] - queues[1].size();
        while (!queues[1].empty())
            queues[0].enqueue(queues[1].dequeue());
    }
    k = 0;
    while (!queues[0].empty())
        data[k++] = queues[0].dequeue();
}
```

Division is performed by the bitwise shift operator (>>). The variable pmax is the number of the largest digit. After each pass, pmax is multiplied by 10. Thus, if data[j] is 1234, then data[j] / pmax is 1 on the first pass, 12 on the second pass, 123 on the third pass, and 1234 on the fourth pass. The expression (data[j] / pmax) % 10 is the digit in the ones position, the tens position, the hundreds position, and the thousands position, respectively. Division is used instead of shifting in this algorithm because the base is 10 rather than 2. The algorithm is still O(n).

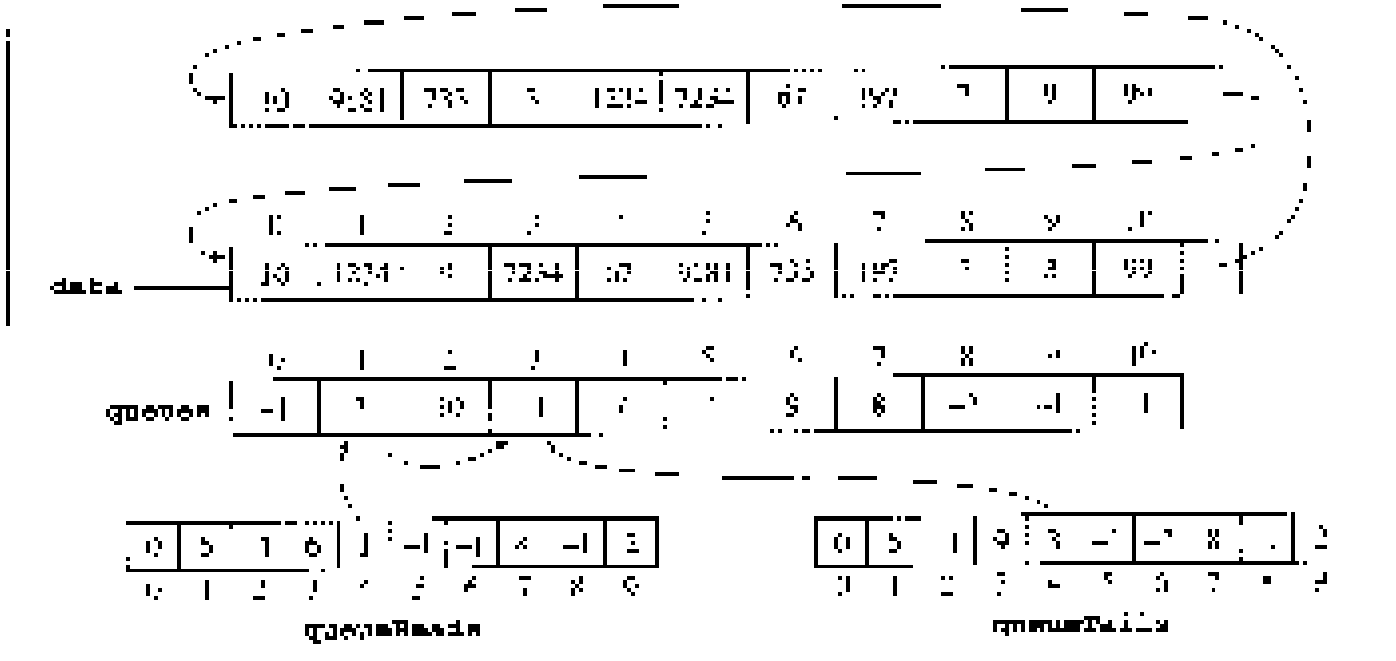
Quicker operations can sometimes be a larger number of moves. In addition, the radix sort is much slower than radix sort because the queues are implemented as linked lists and for each item in the list a node must be removed from the list to be enqueued and attached to the queue. Before it is dequeued back to the original array, the node has to be detached from the queue and disposed of. Although the dynamically allocated portions are conceptually impressive, it does not make the operations on queues quicker. It slows them down.

A better implementation is to use arrays of size n for each queue and to copy between these queues only once. The efficiency of the algorithm depends only on the number of exchanges (copying) and not upon queues. However, it requires a large number and a large amount of data has to be moved. A radix sort requires enough space to hold n numbers ($n - 1$ in the queues, not all queues will be used) may be unacceptable if large.

A better solution uses one integer array queues of size n to represent the linked lists of indexes of numbers belonging to the ten queues. Each of the ten queues includes an index of the first number in queues which belongs to this queue, or -1 if it is empty. queues[i] contains the index of the last number in the queue. Figure 9.16 illustrates the situation before the first pass for the queues. Each number has a link to the next number in the queue, with -1 as the last digit. For example, queues[3] contains 5 as the index of the next number in data with 3 as the last digit. queues[1] contains 0 as the index of the next number in data with 1 as the last digit. queues[7] contains -1 as the last digit.

The next stage moves data back to the original array in order, gathered up queue by queue. It copies the data from the original array to a temporary storage and then back to

Figure 9.16 An implementation of radix sort



[illegible] To avoid the excessive copying, two arrays can be used, [illegible] a two-element [illegible]. After copying, the pointer [illegible] is moved to the next array, and the array [illegible] the cost of the average [illegible]. The improvement is significant, and the new implementation runs at least two times faster than the implementation that was proposed (Figure 9.18).

FIGURE 9.17 Demonstration of sorting functions.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // greater<>

using namespace std;

class Person {
public:
    Person(char *n = "", int a = 0) {
        name = new char[strlen(n)+1];
        strcpy(name,n);
        age = a;
    }
    bool operator==(const Person& p) const {
        return strcmp(name,p.name) == 0;
    }
    bool operator<(const Person& p) const {
        return strcmp(name,p.name) < 0;
    }
    bool operator>(const Person& p) const {
        return !(*this == p) && !(*this < p);
    }
private:
    char *name;
    int age;
    friend ostream& operator<<(ostream& out, const Person& p) {
        out << "(" << p.name << "," << p.age << ")";
        return out;
    }
};

bool f1(int n) {
    return n < 5;
}
```

Continues

Program 9.17 continued

```
template<class T>
void printVector(char *s, const vector<T> v) {
    cout << s << " = [";
    if (v.size() == 0) {
        cout << "]\n";
        return;
    }
    for (vector<T>::const_iterator i = v.begin();
                                   i < v.begin()+v.size()-1; i++)
        cout << *i << ",";
    cout << *i << "]\n";
}

void main() {
    int a[] = {1,4,2,6,7,3,5};
    vector<int> v1(a,a+7), v2(a,a+7), v3(6,9), v4(6,4);
    vector<int>::iterator i1, i2, i3, i4;
    partial_sort(v1.begin()+1,v1.begin()+3,v1.end());
    print_vector("v1",v1);                          // v1 = [1,2,3,6,7,5,4]
    partial_sort(v2.begin()+1,v2.begin()+4,v2.end(),greater<int>());
    print_vector("v2",v2);                          // v2 = [1,7,6,5,3,2,4]
    i1 = partial_sort_copy(v2.begin(),v2.begin()+4,v3.begin(),v3.end());
    print_vector("v3",v3);                          // v3 = [1,5,6,7,9,9]
    cout << *i1 << " " << i1 << endl;               // 7 9
    i4 = partial_sort_copy(v1.begin(),v1.begin()+4,v4.begin(),v4.end(),
                           greater<int>());
    print_vector("v4",v4);                          // v4 = [6,5,2,1,9,9]
    cout << *i4 << " " << i4 << endl;               // 1 9
    i1 = partition(v1.begin(),v1.end(),3);          // v1 = [1,2,5,4,7,6,5]
    print_vector("v1",v1);
    cout << *(i1-1) << " " << i1 << endl;           // 4 7
    i2 = partition(v2.begin(),v2.end(),bind2nd(less<int>(),3));
    print_vector("v2",v2);                          // v2 = [1,4,2,5,3,6,7]
    cout << *(i2-1) << " " << i2 << endl;           // 3 5
    sort(v1.begin(),v1.end());                      // v1 = [1,2,3,4,5,6,7]
    sort(v1.begin(),v1.end(),greater<int>());       // v1 = [7,6,5,4,3,2,1]

    vector<Person> pv1, pv2;
    for (int i=0; i < 20; i++) {
        pv1.push_back(Person("Name",60 - i));
        pv2.push_back(Person("Name",60 - i));
    }
```

Program 9.17 continued

```
    sort(pv1.begin(),pv1.end());        // pv1 = {(Josie,41),...(Tess,86)}
    sort(pv2.begin(),pv2.end(),cmp2);// pv2 = {(Tess,86),...(Josie,41)}

    vector<int> heap1, heap2, heap3{9,8,7}, heap4{9,8,7};
    for (i = 1; i <= 7; i++) {
        heap1.push_back(i);
        push_heap(heap1.begin(),heap1.end());
        printVector("heap1",heap1);
    }
    // heap1 = {1}
    // heap1 = {2,1}
    // heap1 = {3,1,2}
    // heap1 = {4,3,2,1}
    // heap1 = {5,4,2,1,3}
    // heap1 = {6,4,5,1,3,2}
    // heap1 = {7,4,6,1,3,2,5}
    cout<<heap1.back()<<heap1.end();// heap1 = {1,2,3,4,5,6,7}
    for (i = 1; i <= 7; i++) {
        heap2.push_back(i);
        push_heap(heap2.begin(),heap2.end(),greater<int>());
        printVector("heap2",heap2);
    }
    // heap2   {1}
    // heap2   {1,2}
    // heap2   {1,2,3}
    // heap2   {1,2,3,4}
    // heap2   {1,2,3,4,5}
    // heap2   {1,2,3,4,5,6}
    // heap2   {1,2,3,4,5,6,7}
    sort_heap(heap2.begin(),heap2.end());    // heap2   {7,6,5,4,3,2,1}
    make_heap(heap3.begin(),heap3.end());    // heap3   {7,6,5,1,4,2,3}
    sort_heap(heap3.begin(),heap3.end());    // heap3   {1,2,3,4,5,6,7}
    make_heap(heap4.begin(),heap4.end(),greater<int>());
    cout <<"heap4"<<heap4;                   // heap4 = {1,4,2,6,7,3,8}
    sort_heap(heap4.begin(),heap4.end(),greater<int>());
    cout <<"heap4"<<heap4;                   // heap4 = {9,8,7,4,3,2,1}
}
```

9.4 Sorting in the Standard Template Library

The STL provides many sorting functions, particularly in the library <algorithm>. The functions are implementing some of the sorting algorithms discussed in this chapter: quicksort, heapsort and mergesort. A program in Figure 9.17 demonstrates these functions. For descriptions of the functions see also Appendix B.

The first set of functions are partial sorting functions. The first version picks k = middle - first smallest elements from a container and puts them in order in the range [first, middle). For example,

```
partial_sort(v1.begin(), v1.begin()+3, v1.end());
```

takes the three smallest integers from the entire vector v1 and puts them in the first three cells of the vector. The remaining integers are put in cells from last to end. In this way v1 = {1,3,5,8,7,2,5} is transformed into v1 = {1,2,3,8,7,5,5}. This version orders elements in ascending order. An ordering criterion can be changed if it is provided as the fourth parameter to the function call. For example,

```
partial_sort(v2.begin(), v2.begin()+3, v2.end(), greater<int>());
```

picks the largest three integers from the vector v2 and puts them in positions now to hold in descending order. The order of the remaining integers outside this range is not specified. This call transforms v2 = {1,4,6,6,4,2,7} into v2 = {7,6,6,1,4,2,4}.

The third version of partial sorting takes 4 — first1 and last1 or first2 — last2, whichever is smaller, first elements from the range [first1, last1), and writes them over elements in the range [first2, last2). The call

```
it = partial_sort_copy(v2.begin(), v2.begin()+4, v3.begin(), v3.end());
```

takes the first four integers in vector v2 = {7,6,6,1,4,2,4} and puts them in ascending order in the first four cells of v3 so that v3 = {0,0,0,0,0,0,0} is transformed into v3 = {1,2,4,6,0,0,0}. As an extra, an iterator is returned that refers to the first position after the last copied number. The fourth version of the partial sorting function is similar to the third, but it takes a function predicate relation which elements are sorted. The program in Figure 9.17 also demonstrates the partition function which orders two ranges with respect to one another by putting in the first range numbers for which a one argument Boolean function is true; numbers for which the function is false are in the second range. The call

```
it = partition(v1.begin(), v1.end(), f);
```

uses an explicitly defined function f() and transforms v1 = {1,2,3,6,4,2,0} into v1 = {1,2,3,4,2,6,3} by putting all numbers less than six in front of numbers that are not less than six. The call

```
it = partition(v2.begin(), v2.end(), bind2nd(less<int>(), 6));
```

accomplishes the same for v2 with the built-in function bind2nd which binds a second argument of the function less<int>() to the value 6, thus creating a unary function that works just like f().

```
    public static void main(String[] args) {
        int[] a = {1,3,5,7};
        int[] b = {0,2,4,6,8};
        int[] c = merge(a, b);
        System.out.println("c.length = " + c.length);
        for (int i = 0; i < c.length; i++)
            System.out.println("c[" + i + "] = " + c[i]);
    }
}
```

The output from this program is

```
c.length = 9
c[0] = 0
c[1] = 1
c[2] = 2
c[3] = 3
c[4] = 4
c[5] = 5
c[6] = 6
c[7] = 7
c[8] = 8
```

Exercise R-9.7 provides a stable sorting function for sorting objects that are not comparable, but which have keys that are comparable. The idea is to create a new array of entry objects, each storing an element of the original array and its index in that array. The entries are then sorted based on the keys of the elements, with the index being used as a tiebreaker in case two keys are equal. This technique ensures that the sorting algorithm will treat two objects as being equal only if they are the same object. Because no two objects will ever be considered equal, this technique can be used to convert any sorting algorithm into a stable sorting algorithm. Although this technique is not useful for sorting arrays of integers, it is useful for sorting two equal arrays of objects that are equal to each other. For example, if `Movie[] a` and `Movie[] b` are arrays such that `a[i].equals(b[i])` for `i = 0, ..., a.length - 1`, then `a` and `b` will be sorted identically by `StableSort.sort(a, comparator)` and `StableSort.sort(b, comparator)`, for any comparator for `Movie` objects. This technique can be used to create a stable sorting algorithm. Note, however, that for small numbers of objects, `sort()` is also a good choice, but for small numbers insertion sort is used, not quick sort (see Figure 9.2 and Section 9.3.1).

9.5 Concluding Remarks

Figure 9.15 compares the running times for various sorting algorithms on different numbers of integers on a personal computer. The algorithms are listed in the order of their asymptotic worst-case running times. The times are given in seconds, and the tests were performed on a machine with a 2.66 GHz Intel Core 2 Duo processor. For each algorithm, the sorting rate is given for sorting an array of 8000 integers and for sorting an array of 1,000,000 integers. Recalling that $n \log n$ grows slightly faster than n, we would expect that the sorting rate should decrease for the algorithms with $O(n \log n)$ worst-case running time as n grows from 8000 to 1,000,000. Indeed, this is the case for the algorithms that are based on the divide-and-conquer technique. For example, heap sort requires 0.55 seconds to sort 8,000 integers and 1.55 seconds to sort 1,000,000 integers, which is a sorting rate that slows by a factor of 2.8 as n grows by a factor of 125. For insertion sort, which runs in $O(n^2)$ time in the worst case, the sorting rate slows by a factor of 1,950.25 = 2.2 as n grows by a factor of 125, which is consistent with the $O(n^2)$ running time.

The table in Figure 9.15 indicates that even the fundamentally inefficient sorting methods, which are squared algorithms, slow down approximately by a factor of 4 when the size of the data is doubled, which is the same behavior as for the asymptotically faster methods. The worst-case complexity is $O(n^2)$, which is approximately 2. This is also true for the four implementations of insertion sort whose complexity equals $O(n \log n)$. The table also shows that quicksort is the fastest algorithm among all sorting methods, since it runs at least twice as fast as any other algorithm.

FIGURE 9.18 Comparison of run times for different sorting algorithms and different numbers of integers to be sorted.

	10,000				40,000			
	Ascending	Random	Descending		Ascending	Random	Descending	
insertionSort	0	0.15	0.32		0	2.12	5.08	
selectionSort	0.12	0.17	0.13		2.08	2.33	2.08	
bubbleSort	0.23	1.08	2.63		8.33	18.08	41.31	
shellSort	0.01	0.01	0.01		0.04	0.05	0.04	
heapSort	0.04	0.04	0.04		0.14	0.14	0.14	
mergeSort	0.02	0.03	0.02		0.07	0.08	0.07	
quickSort	0.03	0.03	0.03		0.12	0.13	0.12	
quickSort2	0.03	0.03	0.03		0.12	0.13	0.12	
radixSort	0.04	0.04	0.04		0.15	0.16	0.15	
insertionSort2	0.01	0.07	0.14		0.03	0.29	0.57	
selectionSort2	0.06	0.08	0.06		0.25	0.31	0.25	
bubbleSort2	0.11	0.53	1.06		0.44	2.12	4.24	
shellSort2	0.01	0.01	0.01		0.03	0.04	0.03	
heapSort2	0.03	0.03	0.03		0.12	0.13	0.12	
mergeSort2	0.02	0.03	0.02		0.06	0.07	0.06	
quickSort3	0.02	0.02	0.02		0.09	0.09	0.09	
quickSort4	0.02	0.02	0.02		0.09	0.09	0.09	
radixSort2	0.03	0.03	0.03		0.12	0.13	0.12	
insertionSort3	0	0.08	0.16		0	0.31	0.63	
selectionSort3	0.06	0.08	0.06		0.25	0.31	0.25	
bubbleSort3	0.11	0.53	1.06		0.44	2.12	4.24	
shellSort3	0.01	0.01	0.01		0.03	0.04	0.03	
heapSort3	0.03	0.03	0.03		0.12	0.13	0.12	
mergeSort3	0.02	0.03	0.02		0.06	0.07	0.06	
quickSort5	0.02	0.02	0.02		0.09	0.09	0.09	
quickSort6	0.02	0.02	0.02		0.09	0.09	0.09	
radixSort3	0.03	0.03	0.03		0.12	0.13	0.12	
insertionSort4	0	0.08	0.16		0	0.31	0.63	
selectionSort4	0.06	0.08	0.06		0.25	0.31	0.25	
bubbleSort4	0.11	0.53	1.06		0.44	2.12	4.24	
shellSort4	0.01	0.01	0.01		0.03	0.04	0.03	
heapSort4	0.03	0.03	0.03		0.12	0.13	0.12	
mergeSort4	0.02	0.03	0.02		0.06	0.07	0.06	
quickSort7	0.02	0.02	0.02		0.09	0.09	0.09	
quickSort8	0.02	0.02	0.02		0.09	0.09	0.09	
radixSort4	0.03	0.03	0.03		0.12	0.13	0.12	

9.6 Case Study: Adding Polynomials

Adding polynomials is a computer algebra operation. The user types in two polynomials, and the computer displays their sum. Polynomials may contain several variables, raised to various powers, and their coefficients may be integers or floating-point numbers. Computer algebra systems usually accept input in a fairly natural form. For example, the user might enter

$$3x^2y + 5xy^2 - 6xy^3$$

but $3x^2y$ and $5xy^2$ or $2xy^2$ and $3xy^2$ cannot be combined algebraically unless the first pair of terms have identical variables, and the variables in the second pair are raised to identical powers. We would like to write a program that computes the sum of two polynomials entered by the user. For example, if

$$3x^2y + 5xy^2 - 6xy^3 + 1$$

and

$$2xy^3 - 4x^2y + 7x - 6xy^2 + 4$$

are entered, the program should output

$$2x^2y^3 - 4x^2y + 7x - xy^2 + 5$$

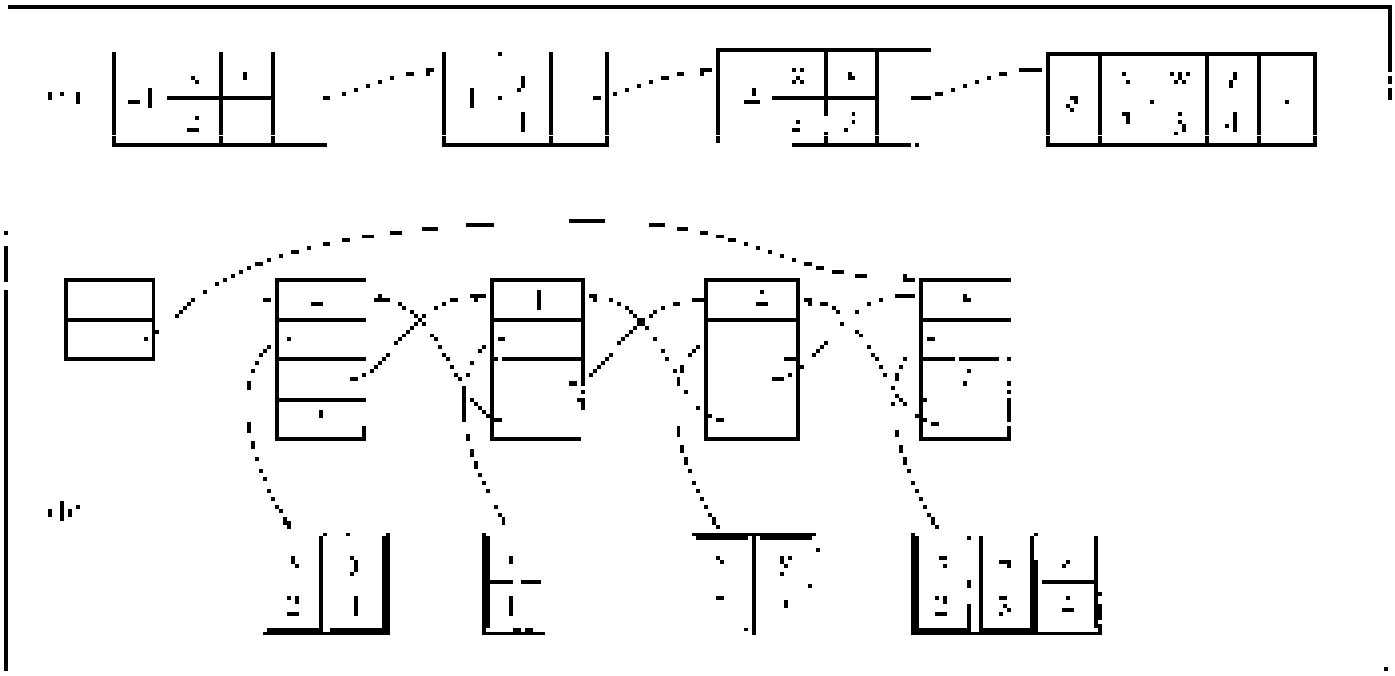
To be more exact, the input and output for this program should be as follows:

```
Enter two polynomials ended with a semicolon:
3x^2y^3 + 5x^2y^3 - 6x^2y^3 + 3;
  2x^2y^3 - 4x^2y + 7x - 6x^2y^2 + 4;
The result is:
   4x^2y^3 - 4x^2y^2 + 7x + 5
```

It has to be observed that the order of variables in a term is irrelevant; for example, xy^2 and y^2x represent exactly the same term. In practice, before any addition is performed, the program should ensure that the variables in each term are raised to the same (correct) powers. This, then, are two major issues to be implemented: ordering variables in each term of both polynomials and adding the polynomials. But before we embark on the problem of implementing the algorithms, we have to decide how to represent polynomials in C++. Out of many possibilities, a linked list representation is chosen with each node on the list representing one term. A node contains information about a specific set of variables and exponents included in the term. Because each variable and its exponent belong together, they are also kept together in an object of type `VarAndExp`. Also, because the number of variables can vary from one term to another, a vector of the `VarAndExp` objects is used to store in one node information concerning variables in one term. A polynomial is simply a linked list of such nodes. For example,

$$xy^2 + y^3 - 3x^2y^2 + 8x^2y^3$$

is represented by the linked list shown in Figure 9.6, which looks like the list in Figure 9.5b



The first operation to perform is to evaluate the polynomials in order to obtain numerical terms. After a polynomial is entered, each term is stored separately in sorting order in a linked list accessible from the header of the list.

The second task is to add the values of like terms. The process begins by creating a linked list consisting of copies of nodes of the polynomials to be added. In this way the two polynomials are not affected and can be used in other operations.

Now, addition is reduced to a simple traversal of the linked list, adding equal terms. Recall that except for the coefficient, two terms have equal exponents and redundant nodes must be eliminated. For example, if the list being processed is as in Figure 9.20a, then the resulting list in Figure 9.20b results from the simplification operation.

When combining like terms, remember that zero-valued terms should be omitted. If the coefficient is zero, a term is omitted. If a term is a constant, the term is printed, but the coefficient is not printed (except for the sign), unless the term has just a zero coefficient. For exponents greater than one, the exponent is printed.

Another printing challenge is to display terms of a polynomial, that is, zero-entry a somewhat data type and print a minus

$$\text{[illegible]}$$

and then display it.

$$\text{[illegible]}$$

The sorting routine may be invoked to present the polynomials in sorted order.

Figure 9.21 shows the complete output for the program to add polynomials.

FIGURE 9.20 [illegible] list representing the expression [illegible] and a linked list representing a simplified version of this expression, $4x^3y^2 + 1$.

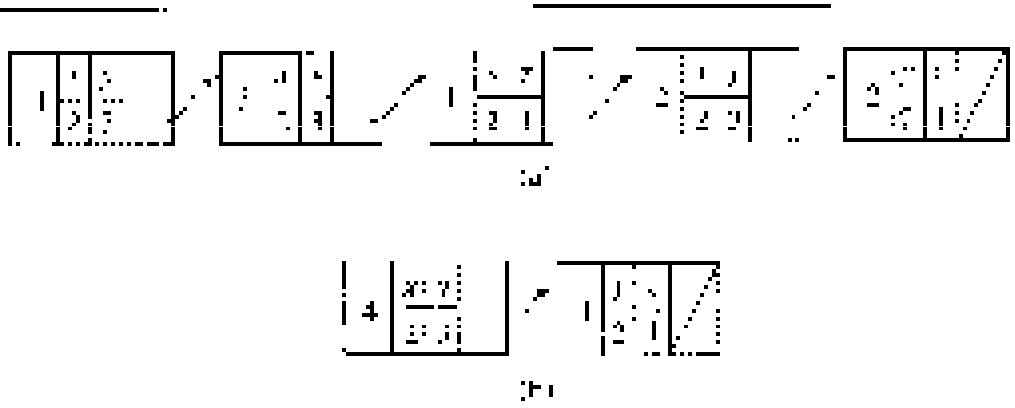


FIGURE 9.21 Implementation of program to add polynomials.

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <vector>
#include <stack>
#include <new>
using namespace std;

const int MAXEXPONENT = 15;

class Variable {
public:
    char id;
    int exp;
    Variable() { // required by vector<>
    }
    Variable(char c, int i) {
        id = c; exp = i;
    }
    bool operator== (const Variable &v) const {
        return id == v.id && exp == v.exp;
    }
    bool operator< (const Variable &v) const { // used by sort()
        return id < v.id;
    }
};
```

Figure 9.21 [illegible]

```
class Term {
public:
    Term() {
        coef_ = 0;
    }
    int coef();
    const VariableMap vars;
    bool operator==(const Term& term) const;
    bool operator!=(const Term& term) const { // required by GCC
        return !(*this == term);
    }
    bool operator<(const Term& term) const;
    bool operator>(const Term& term) const { // required by GCC
        return term < *this && term != *this;
    }
    int min(int x, int m) const {
        return (x < m) ? x : m;
    }
};

class Polynomial {
public:
    Polynomial() {
    }
    Polynomial operator+(const Polynomial&) const;
    void error(char* s) {
        cerr << s << endl; exit(1);
    }
private:
    TermSet terms;
    friend ostream& operator<<(ostream&, Polynomial&);
    friend ostream& operator<<(ostream&, const Polynomial&);
};

// two terms are equal if all variables are the same and
// corresponding variables are raised to the same powers
// the first one in the map determines a term is smaller
// than comparison, since it stores coefficient of the term

bool Term::operator==(const Term& term) const {
    for (int i = 0; i < min(vars.size(), term.vars.size()); ++
```

Figure 9.21 (continued)

```
                    vars[i] == term.vars[i]; i++);
    return i == vars.size() && vars.size() == term.vars.size();
}

bool Term::operator<(const Term& term2) const { // used by sort();
    if (vars.size() == 0)
        return false;           // *this is just a coefficient;
    else if (term2.vars.size() == 0)
        return true;            // term2 is just a coefficient;
    for (int i = 0; i < min(vars.size(),term2.vars.size()); i++)
        if (vars[i].id < term2.vars[i].id)
            return true;        // *this precedes term2;
        else if (term2.vars[i].id < vars[i].id)
            return false;       // term2 precedes *this;
        else if (vars[i].exp < term2.vars[i].exp)
            return true;        // *this precedes term2;
        else if (term2.vars[i].exp < vars[i].exp)
            return false;       // term2 precedes *this;
    return (vars.size() < term2.vars.size()) ? true : false;
}

Polynomial Polynomial::operator+(const Polynomial& poly2) const {
    Polynomial result;
    list<Term>::iterator p1, p2;
    bool match;
    for (p1 = terms.begin(); p1 != terms.end(); p1++) // create a new
        result.terms.push_back(*p1);                  // poly from
                                                      // copies of *this
    for (p1 = poly2.terms.begin(); p1 != poly2.terms.end(); p1++)
        result.terms.push_back(*p1);                  // and poly2;
    for (p1 = result.terms.begin(); p1 != result.terms.end(); ) {
        for (p2 = p1, p2++, match = false; p2 != result.terms.end();
             p2++)
            if (*p1 == *p2) {             // if two terms are equal (except
                p1->coeff += p2->coeff;   // for the coefficient), add the
                result.terms.erase(p2);   // two coefficients and erase
                if (p1->coeff == 0)       // a redundant term; else
                result.terms.erase(p1);   // one of p1 or p2 is retained
                match = true;             // term is gone, break;
                                          // erase() terminates loop 1;
```

Listing 9.21 continued:

```
        }
        if (previous)        // restore previous from the key string
            pi = result.varmap.begin();  // to key used previously
        else pi++;
    }
    result.varmap.clear();
    return result;
}

istream& operator>>(istream& in, Polynomial& poly) {
    char ch, sign, newTerm, ins;
    int exp;
    Term term;
    in >> ins;
    while (true) {
        newTerm = 0;
        if (isalpha(ins) && ins != 'x' && ins != 'X') {
            polyParseError("Wrong character entered");
        sign = 1;
        while (ins == '-' || ins == '+') { // Merge consecutive signs
            if (ins == '-')
                sign *= -1;
            ins = in.get();
            if (isspace(ins))
                in >> ins;
        }
        if (isalpha(ins)) {              // Term if its sufficient;
            in.putback(ins);
            in >> term.coeff;
            ch = in.get();
            term.coeff *= sign;
            newTerm = 1;
        }
        else term.coeff = sign;
        for (int i = 0; isalpha(ch); i++) { // process this term
            id = ch;                        // get a variable name
            ch = in.get();
            if (isalpha(ch)) {              // and an exponent if any;
                in.putback(ch);
                in >> exp >> ch;
            }
```

Figure 9.21 (continued)

```
            else cap   1:
            terms.vec.push_back(Variable(i,cap));
        }
        poly.terms.push_back(term);// and I should it in the linked list;
        term.vec.resize(0);
        if (isspace(ch))
            in >> ch;
        if (ch     ';')              // false if a semicolon is entered:
            if (coeff!=0 || i > 0)
                break;
            else polynomial("term is missing");  // e.g., 2x - ; or
                                                 // just  :
        else if (ch != '+' && ch != '-' + );     // e.g., 2x  4y;
            polynomial("wrong character entered");
    }
    for (list<Term>::iterator t = poly.terms.begin();
                              t != poly.terms.end(); t++)
        if (t->vec.size() > 1)
            sort(t->vec.begin(),t->vec.end());
    return in;
}

istream& operator>> (istream& in, Polynomial& poly)
    int afterfirst=0, i;
    for (list<Term>::iterator pol = poly.terms.begin();
                              pol != poly.terms.end(); pol++) {
        out.put(' ');
        if (pol->coeff < 0)                  // put '-' before polynomial
            out.put('-');                    // and between terms (if
                                             // needed);
        else if (!start_with_term)           // don't put '+' in front of
            out.put(' ');                    // polynomial;
        afterfirst=1;
        if (abs(pol->coeff) != 1)            // print a coefficient
            out << '  << abs(pol->coeff);    // if it is not 1 or -1, or
        else if (pol->vec.size() == 0)       // it's term has only a
                                             // coefficient
            out << '1';
        else cur.put(' ');
```

Program 9.27 (continued)

```
        for (i = 1; i < pol.vars.size(); i++) {
            out << pol.vars[i].id;      // print a variable name
            if (pol.vars[i].exp != 1)   // and an exponent, only
                out << pol.vars[i-1].exp; // if it is not 1;
        }
    }
    out << endl;
    return out;
}

void main()
{
    Polynomial pol1, pol2;
    cout << "Enter two polynomials, each ended with a semicolon:\n";
    cin >> pol1 >> pol2;
    cout << "The result is:\n" << pol1 + pol2 << endl;
}
```

9.7 EXERCISES

1. Many operations can be performed faster on sorted data. For each of the following operations, explain how
   a. checking whether one word is an anagram of another word, e.g. plum and lump
   b. finding the union of two sets
   c. computing an average of values
   d. finding the median value in a collection
   e. finding the value which appears most frequently in the data
2. The function doubleSort() is inefficient because it continues execution after an array is sorted by performing unnecessary comparisons. Therefore, the number of comparisons is the best and worst cases is the same. The implementation can be improved by making a provision for the case when the array is already sorted. Modify doubleSort() by providing a flag in the outer for loop indicating whether or not it is necessary to make the next pass. Set the flag to true every time an interchange occurs which indicates that there is a need to scan the array again.

3. Will `doubleSort()` work properly if the inner loop

```
for (int i = n - 1; i > k; i--)
```

is replaced by

```
for (int i = n - 1; i > 0; i--)
```

What is the complexity of the new version?

4. In our implementation of bubble sort, a sorted array was created by bubbling up the smallest element. What modifications are needed to make it work by bubbling down the largest element?

5. A cocktail shaker sort is designed by combining bubble up and bubble down. It works by bubbling up the smallest element, then bubbling down the largest element, then bubbling up the second smallest element, then bubbling down the second largest element, and so forth. Implement this algorithm and analyze its complexity.

6. Insertion sort can be viewed as a bubble up process. Write a program to verify that the number of comparisons and the number of moves in insertion sort are the same as in bubble sort.

7. Prove Theorem 9.1 for all the possible values of n_1 and n_2 when $n_1 \neq n_2$.

8. What is the worst-case time for the `merge` method in Listing 9.4? Is it possible to improve the `merge` method to make it run faster?

9. What would be the problem if the following statement is used to replace lines 21–23 in Listing 9.5?

```
list = merge(sortedList(list1), sortedList(list2));
```

10. Implement the `sortedList` method.

11. Show that for any positive integer n, the number of comparisons in `mergeSort` is $\leq 2n \log_2 n$.

12. Implement a nonrecursive version of merge sort. The algorithm can be described as follows: Merge every two consecutive sorted lists, each of size 1, into a sorted list of size 2. Merge every two consecutive lists of size 2 into a sorted list of size 4. The resulting lists are sorted lists of size 4, except perhaps the last one, which may be smaller. Merge every two consecutive lists of size 4 into a sorted list of size 8. Continue the process until the list is sorted. Note that this is a bottom-up approach. Implement the algorithm without using recursion. Your program should contain a method for each of the following tasks: One method merges two sorted lists, another method repeatedly merges every two consecutive lists of a specified size, and a third method repeatedly calls the second method with a list size of 1, 2, 4, 8, and so on. Your program should not copy elements from the original list to a temporary list.

13. The `mergeSort` program requires additional space in order to store the temporary lists during the sort. Is it possible to sort a list in place without using any additional memory? Write a program to implement a sort method that uses only a constant amount of additional memory. (*Hint*: Divide the list into two halves, recursively sort the first half, and merge the first half with the second half. The trick is how to merge the two halves without using additional memory. One approach is to swap the elements in the first half with the elements in the second half to ensure that all the elements in the first half are smaller than those in the second half. Then merge the two halves. Implement this algorithm and investigate its execution time. Another way to make an in-place merge sort is by modifying the data structure that is called a

implement it. A version that disregards the runs by always dividing arrays into almost equal sections is referred to as straight merging.

14. To avoid doubling the workspace needed when arrays are sorted with mergesort, it may be better to use a linked list of data instead of an array. In small increments, this approach is faster. Implement this technique and discuss its complexity.

15. Which sorting algorithms are stable?

16. Consider a sleep sorting algorithm in which application sorts the elements of an *n*-element array where integers in the range 0, 1, ..., *m* are stored. Insertion sort is applied to two elements of the array that are *k* and the middle element, then to three elements, separated by the distance *k*, etc., and finally to every element. Compare the complexity of this approach to

9.8 PROGRAMMING ASSIGNMENTS

1. At the end of the section discussing index sorting, three techniques for changing the bound were mentioned, using a randomly chosen element from the first and a median element of the first, middle, and last elements of the array. Implement these two versions of quicksort, apply them to large arrays, and compare their running times.

2. The board of education maintains a number of substitute teachers in the area. If a temporary replacement is necessary for a class, a substitute is available, and there is room in the school, that person is hired. Write a menu-driven program maintaining this database.

   The data should include the first and last names of the substitute teachers, an indication of whether or not they are currently working (yes or no), and a list of numbers which represent the subjects they are qualified to teach. Suppose the

Subject	Number
Art	1
Biology	2
Chemistry	3
Computer Science	4
English	5
French	6
German	7
History	8
Mathematics	9
Physical Education	10
Physics	11
Spanish	12

   Create a menu-driven program with options to add a substitute, remove a substitute, change the working status of a substitute, and display the names of substitutes who are qualified to teach a given subject. Also, display the names of all substitutes who are currently working.

FIGURE 9.22 Data structures used by the house of cousins to substitute numbers

Row Index	Can Juice	Grape Mango	Pear Guava	Hibiscus Key	Rose Kumquat	Lychee Tangerine	Noni Watermelon	Salacca Yuzu	Durian Soursop
0	✓								
1									
2									
3									
4									
5									

First prepare for each name in your binary search tree. Both first and last names should be substituted. Proceed as in the algorithms discussed in this section. Then store the binary tree in the function hashcode with each name stored in the index corresponding to that object. This is the field of each node of the tree location in hashcode of the character string object. Note that Euler's theorem should be able to prove the hypothesis. Fig. 9.22 shows the order of the names and trees accessible from the object as created by the following function.

Allow the user to view the tree as requested. If the user chooses the exit option, choose, and all entries from names back to substitutes, by time each created accessibly information.

3. Implement different versions of Shell sort by using simple sorts and different sequences and different processes for the calculation each of which can be written, but choose the following sequences:

   a. $h_1 = 1$, $h_{i+1} = 2h_i + 1$ and stop with h_t for which $h_{t+2} \geq n$ (Knuth)

   b. $2^i - 1$ (Hibbard)

   c. $2^i + 1$ (Papernov and Stasevich)

   d. Fibonacci numbers

   e. $\frac{1}{2}$ is the first increment and then $h_i = 2.2h_{i-1}$ (Divide by 2.2)

   Run all these versions for at least five values of n in the range 1000, 5000, 10,000, 50,000, and 100,000. Tabulate and plot the results. Try to approximate their work as a function of n, assuming the complexity of these versions is

4. Extend the program from the case study to include polynomial multiplication.
5. Extend the program from the case study to include polynomial differentiation. For instructions, see the exercises in Chapter 1.

Bibliography

Sorting Algorithms

[illegible], Christian Antipo Engineering C++ Data Structures, Prentice Hall, 1995

[illegible], Donald E. The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd Ed. Reading, MA: Addison-Wesley, 1998.

[illegible], Harold. Structures of Computer Computations, 2nd Ed. Hinsdale, IL: The Dryden Press, 1978.

[illegible], Karl. Data Structures and Algorithms, Vol. 1: Sorting and Searching, 2nd Ed. Berlin: Springer-Verlag, 1984.

[illegible], Robert. Algorithms in C++, Reading, MA: Addison-Wesley, 1992.

[illegible], Robert, and Kevin Wayne. Algorithms, 4th Ed. Boston: Addison-Wesley Professional, 2011.

Shell Sort

[illegible], Marcin, and Jakub Wojciech Kowalski. "Improving Shellsort Using Knuth's Gap Sequence." Proceedings of the International Multiconference on Computer Science and Information Technology, 2008, pp. 251–258.

[illegible], Robert. "Analysis of Shellsort and Related Algorithms." Proceedings of the Fourth Annual European Symposium on Algorithms, 1996, pp. 1–16.

Incerpi, Janet, and Robert Sedgewick. "Improved Upper Bounds on Shellsort." Journal of Computer and System Sciences, Vol. 31, No. 2, 1985, pp. 210–224.

Shell, Donald L. "A High-Speed Sorting Procedure." Communications of the ACM, Vol. 2, No. 7, 1959, pp. 30–32.

Weiss, Mark Allen, and Robert Sedgewick. "Tight Lower Bounds for Shellsort." Journal of Algorithms, Vol. 11, No. 2, 1990, pp. 242–251.

Merge Sort

[illegible], Robert. "Quicksort." Software—Practice and Experience, Vol. 7, No. 1, 1977, pp. 1–11.

Jiang, Tao, Ming Li, and Paul Vitányi. "A Well-Characterized Approximation for the Expected Number of Comparisons in Mergesort." Theoretical Computer Science, Vol. 305, No. 1–3, 2003, pp. 69–90.

Knuth, Donald E. The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd Ed. Reading, MA: Addison-Wesley, 1998.

Sedgewick, Robert, and Kevin Wayne. Algorithms, 4th Ed. Boston: Addison-Wesley Professional, 2011.

Williams, J. W. J. "Algorithm 232: Heapsort." Communications of the ACM, Vol. 7, No. 6, 1964, pp. 347–348.

Quicksort

Hoare, Charles A. R. "Algorithm 64: Quicksort." Communications of the ACM 4.7 (1961): 321.

Hoare, Charles A. R. "Quicksort." Computer Journal 5.1 (1962): 10–15.

Sedgewick, Robert. Quicksort. New York: Garland, 1980.

Mergesort

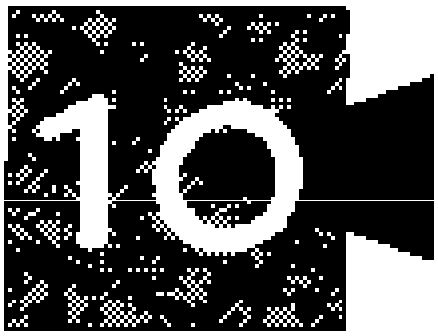
Knuth, Donald. "Von Neumann's First Computer Program." Computing Surveys 2.4 (1970): 247–260.

Slow Sorting

Bulterman, R. "Slow Sorting." Communications of the ACM 16.12 (1973): 787–790.

Decision Trees

Moret, B. M. E. "Decision Trees and Diagrams." Computing Surveys 14.4 (1982): 593–623.



Hashing

But even if this table can accommodate all the variables in the program, how can we design a function h which assigns a complete and immediately distinct location associated with each variable? All the letters of the variable name can be added together and the sum can be used as an index. In this case, the code needs 3782 cells for a variable X made out of 31 letters: "X" = 2 + 122 = 3782. But even with this sparse, the function does not return unique values. For example, h("abc") = h("acb"). This problem is called *collision* and it is discussed later. The worst hash function does not look well to avoid collisions. Avoiding collisions can be achieved by making the function more sophisticated, but this sophistication should not go too far because the computational cost in determining $h(K)$ can be prohibitive and less sophisticated methods may be faster.

10.1 Hash Functions

The purpose of hash functions that can be used to assign positions to records in a table of organization *hash tables*. The number of perfect hash functions is the same as the number of different placements of these items in the cells and is equal to $n!$. For example, for a file of records and a 100 cell array, there are $100! \approx 9^{157}$ hash functions out of which "only" 10^{40} are in a minimum, are perfect. Most of these functions are too unwieldy for practical applications and cannot be represented by a concise formula. However, even among functions that can be expressed with a formula, the number of possibilities is vast. This section discusses some specific types of hash functions.

10.1.1 Division

A hash function must guarantee that the number it returns is a valid index to one of the table cells. The simplest way to accomplish this is to use division modulo: TSize = sizeof(Table); $h(K) = K \bmod TSize$. If K is a number, it is best if TSize is a prime number. Otherwise, $h(K) = gK \bmod p$ may. TSize or some prime $p >$ TSize can be used. However, composite numbers may work as well as prime divisors provided they do not have prime factors less than, say, 20. The division method is usually the preferred choice for the hash function if very little is known about the keys.

10.1.2 Folding

In this method, the key is divided into several parts. Addition conveys the true meaning of the word *fold*. These parts are combined or folded together and are often transformed in a certain way to create the target address. There are two types of folding: *fold shift* and *fold boundary folding*.

The key is divided into several parts and these parts are then processed using a simple operation. Then addition or combination in a certain way. In shift folding, they are put under one another and then processed. For example, a social security number such as 123-45-6789 can be divided into three parts: 123, 456, 789, and

These parts can be added. The resulting number, 1368, can be divided modulo 1000. The size of the table is 1000, the first three digits can be used from the result. To be sure, the division can be done in many different ways. Another possibility is to divide the same number 123456789 into five parts (say, 12, 34, 56, 78, and 9) and add them and divide the result modulo 1000.

With boundary folding, the key is seen as being written on a piece of paper which is folded on the positions between different parts of the key. In this way, every other part will be written in the reverse order. Consider the same three parts of the SSN: 123, 456, and 789. The first part, 123, is taken in the same order. Then the piece of paper with the second part is folded underneath it so that 456 is aligned with 123, which is the same as 654. In reverse order. When the folding continues, 789 is aligned with the previous two parts. The result is 123 + 654 + 789 = 1566.

In both variations, the key is usually divided into even parts of some fixed size plus some possible remainder. This process is simple and fast, especially when bit vectors are used instead of numerical values. A typical version of shift folding is obtained by applying the exclusive or operation (^).

In the case of strings, one approach processes all characters of the string by "scrunching" them together and using the result for the address. For example, for the string "abcd," "a" << 24 ^ "b" << 16 ^ "c" << 8 ^ "d". However, this simple method results in addresses between the numbers 0 and 127. For better results, chunks of strings are "scrunched" together rather than single characters. These chunks are composed of the number of characters equal to the number of bytes in an integer, since an integer in a C++ implementation for the IBM PC computer is two bytes long. A "shift" – "xb" works most likely like modulo operation. Such a function is used in the hash map in this chapter.

10.1.3 Mid-Square Function

In the mid-square method, the key is squared and the middle or mid part of the result is used as the address. If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding. In a mid square number function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys. For example, if the key is 3121, then $3121^2 = 9740641$ and for the 1000 values in 3121 = 406, which is the middle part of 3121. In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If we assume that the size of the table is 1024, then in the example the binary representation of 3121 is 110000110001 and the binary 1000010100000001 is found with the middle part shown in bold. This middle part, the binary number 0100000000, is equal to 512. This part can easily be extracted by using a mask and a shift operation.

10.1.4 Extraction

In the extraction method, only a part of the key is used to compute the address. For the social security number 123-45-6789, this method might use the first four digits (1234), the last four (6789), the first two combined with the last two (1289), or some other combination. Each time, only a portion of the key is used. If this portion is carefully

[illegible] key only in the original key. For example, all keys with the same first letter go to the same position. Unless the frequency for each letter is known and taken into account, the resulting dictionary may not be very efficient. For example, consider an English dictionary stored in an array with 26 positions. The first letter of each word is used to assign words to positions, with all words beginning with 'a' stored in the first position, those beginning with 'b' in the second, and so on. Unless we know ahead of time that there are many more words beginning with 'x' than with 'z', for example, we must assume that each position in the array will require roughly the same amount of space. Another problem with this function is that it is case sensitive. Unless case is ignored, words beginning with 'A' will be stored in a different part of the array than those beginning with 'a'. Finally, this function is inappropriate for variable-length strings, because short words will access only a small portion of the array. Ideally, a hash function should be chosen so that each position in the array has an equal probability of being visited by the hash function. Section 10.3 discusses some approaches to designing good hash functions.

10.1.5 Radix Transformation

Using the radix transformation, the key K is converted into another number. K is expressed using a numerical system with a different radix. If K is the decimal number 345, then its value in base 9 numerals is 423. This value is then used modulo T (the table size), and the resulting number is used as the address for the location to which K should be hashed. Collisions, however, cannot be avoided. If T is 100, then 345 and 245 (which are not synonyms) will be assigned to the same location. 345 and 1291 are synonyms because 291 decimal is 423 in base 9 numeral system, and both 345 and 323 evaluate to 23 when divided modulo 100.

10.2 Collision Resolution

Note that a hash function and hash array are inappropriate for applications where duplicate key values are not allowed. Any key that hashes to a location already occupied by a different key value will be treated as a collision, even if the two key values are identical. Thus, hash tables are inappropriate for applications that do not allow duplicate key values. (Alternatives for storing duplicate key values in hash tables are discussed briefly in Section 10.2.1.)

Hash functions must be chosen carefully because poor hash functions can result in many collisions. Ideally, a hash function should distribute records evenly among the positions in the table so as to minimize collisions. Collisions are likely even under the best circumstances—for example, no matter how many records are in the table, the probability that a new record will be assigned to the first position in the array by a hash function that distributes records evenly among all positions in the table is $1/T$. Good hash functions do not try to avoid collisions, but rather try to assign records to locations within the table in such a way that the number of collisions is not made worse by any bias in the distribution of key values. Collisions should occur only when the table is relatively full.

Given a function to compute hash addresses, we can implement hash table insertion, deletion, and retrieval operations. For the moment we assume that there is space available in the array such that a collision can be resolved by placing the record in the next available position in the array. Insertion of a record requires that the hash function be computed to find the home position of the record. If another record already occupies that position, then the array is searched sequentially from there until a free position is found. If the end of the array is reached before a free position is found, then the search continues from the beginning of the array. Retrieval works in a similar way. First, the hash function is computed to check the home position. If the record there has the correct key value, then the search is complete. Otherwise, the array is searched sequentially until the record is found. If the search reaches a free position in the array, then the record is not in the table. If the search returns to the home position, then the table is full.

Deletion is difficult. Naively removing a record makes it impossible to retrieve any record that had been placed in the array because it collided with the deleted record. Thus, it is necessary to mark the record as deleted rather than to remove it completely. Otherwise, the deletion will shorten the sequence of records that must be checked when searching for a record that collided with the deleted record. Over time, the array may fill with deleted records, leading to poor performance. Ideally, the array should be rehashed periodically, removing the deleted records.

There are a number of strategies that attempt to avoid hashing multiple keys to the same location. Only a few of these methods are discussed in this chapter.

10.2.1 Open Addressing

In the open addressing method, when a key collides with another key, the collision is resolved by finding an available cell within the array (other than the position to which the colliding key is originally hashed). If position i is accessed, then the positions to probe are

$$\text{probe}(k,0) = p(k), \text{probe}(k,1) = p(k)+1, \text{probe}(k,2) = p(k)+2, \ldots$$

and so on until an available cell is found. (The same positions are probed repeatedly as the table is filled.) Function p is a probing function, k is a key, and the number of the probe is the second parameter.

The simplest method is linear probing, for which $p(k) = k$, and for the ith probe, the position to be tried is $(k+i) \bmod M$. As in linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position specified by the hash function until an empty cell is found. If the end of the table is reached, the search continues from the beginning of the table. The search terminates—unsuccessfully—when the cell preceding the one from which the search started is probed. Linear probing, however, has a tendency to create clusters in the table, degrading performance. Figure 10.1 shows an example where the keys 5, 8, and 11 hashed to the positions 5, 8, and 11, respectively. Key 18 hashes to position 5, which is occupied by 5. Since the next position, 6, is occupied by 8, it is stored there. Next, 4 is stored in position 4 because it hashes to 4. Key 24 hashes to position 5, which is the same address as 5, 8, 11, and 18 already stored in the table. Because 5 is occupied, 24 is stored in the first cell after 18, which is 7. Key 21 hashes to position 8, which is occupied by 8. Since 9 is occupied by 18, 21 is stored in position 10. Key 2 hashes to position 2, which is occupied by 11. Since 3 is occupied by 4, 2 is stored in position 12 (the first position after the end of the table).

Another method that reduces collisions (clusters) uses a much smaller chance of being filled for a cell that is i cells away from a cell that is already filled. For example, i^2 is used. Other methods, such as i^3, have been tried. If i^2 is used, then the probe sequence is $k+1^2, k+2^2, k+3^2, \ldots$. Although this method does not guarantee that all cells in the table will be examined before the search is terminated, it does provide good performance. Experiments have shown that this method improves the performance of the hash table for storing and retrieving data. The problem of clustering is known as secondary clustering. A solution can be found in a more sophisticated technique, double hashing.

The probing function must be such that it visits all the cells in the table. A simple function that does this is

$$p(k) = (p(k) + i^2) \bmod M \quad \text{for } i = 1, 2, \ldots, M-1$$

This function visits all the cells in the table, but it does not necessarily visit them in sequence. Another function that does this is

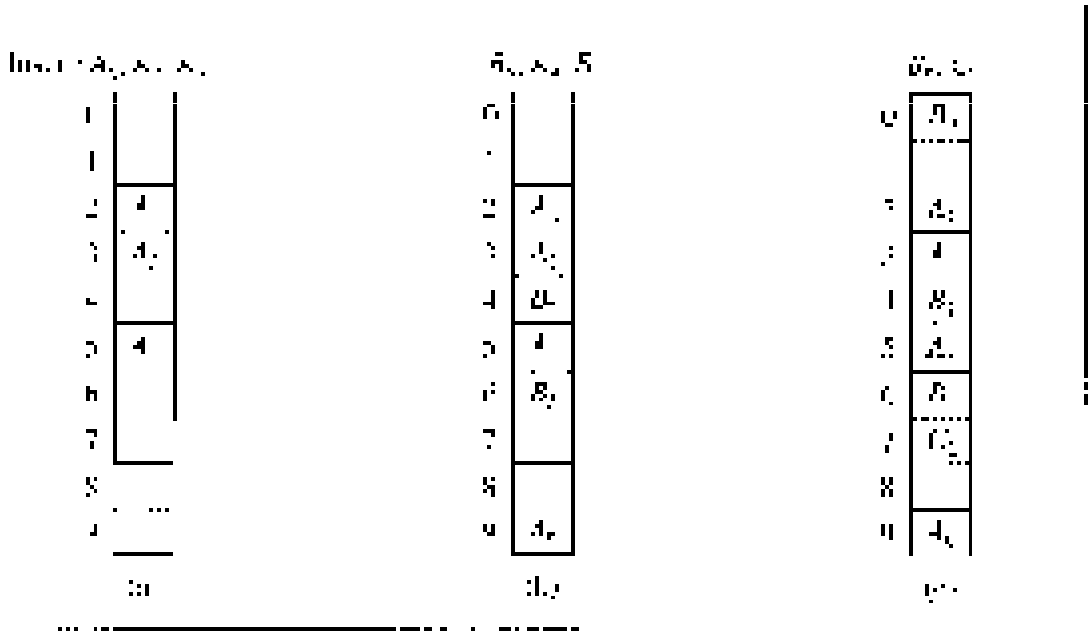
$$p(k) = (p(k) + i \times h(k)) \bmod M \quad \text{for } i = 1, 2, \ldots, M-1$$

In this function, $h(k)$ is a second hash function. A simple but effective function is

$$h(k) = 7 - (k \bmod 7)$$

$h(k)$ must never be zero. Experiments have shown that good functions for $h(k)$ are of the form

FIGURE 10.1 Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed.



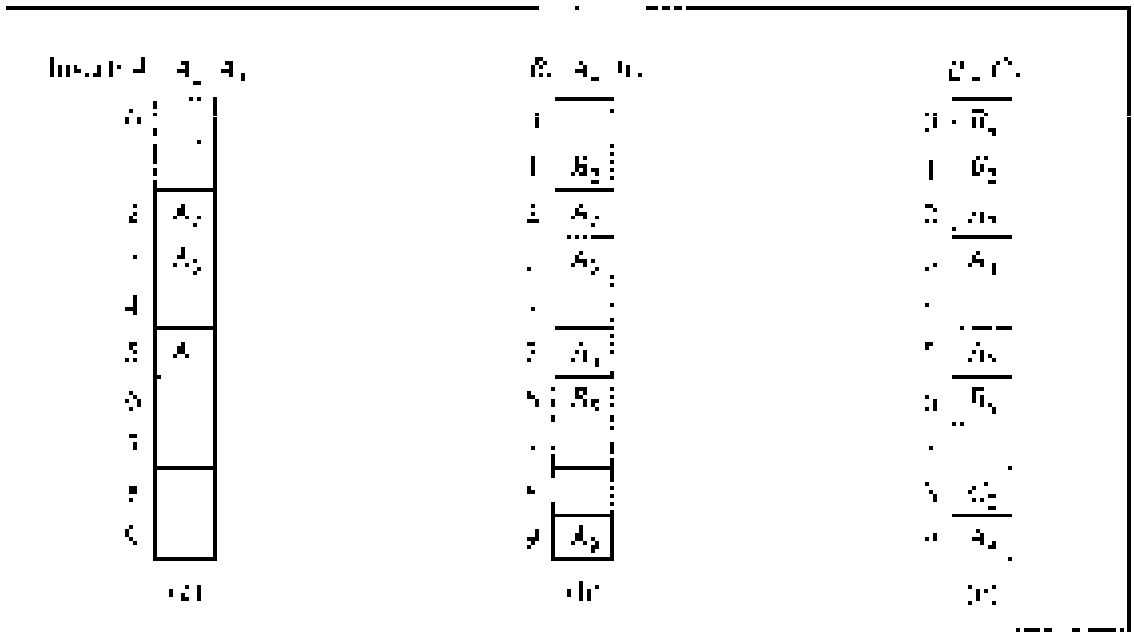
(a) (b) (c)

...

...

...

FIGURE 10.2 The quadratic probing collision algorithm



This is not an optimal. The probes which reach the same location are $c_1 \cdot 1 + c_2 \cdot 1^2$, $c_1 \cdot 2 + c_2 \cdot 2^2$, $c_1 \cdot 3 + c_2 \cdot 3^2$, and the probe sequences for which

$$i \bmod TSize = j \bmod TSize$$

That is,

$$i^2 - j^2 \bmod TSize$$

In this case,

$$i^2 - j^2 = (i - j)(i + j) \bmod TSize = 0$$

and it must be true that $(i + j) \bmod TSize = 0$.

Although quadratic search gives much better results than linear probing, the problem of cluster buildup is not avoided altogether: Any two keys that map to the same location, for which probe sequences are used. Such clusters are called secondary clusters. These secondary clusters, however, are less harmful than primary clusters.

Another possibility is to have p use a random number generator (Morris 1968), which eliminates the need to take special care about the table size. This approach prevents the formation of secondary clusters, but it carries a problem with computing the same probing sequence for the same key. If the random number generator is initialized at the first invocation, then different probing sequences are generated for the

```
[illegible]
```

The problem of secondary clustering is less severe than primary clustering. This is because probing sequences tend to spread out more than primary clustering. For example, if $h(K_1) = h(K_2)$, then the probing sequences for K_1 and K_2 will be identical. However, if $h(K_1) = h(K_2) - 1$, then the probing sequences will be very different. Quadratic probing is implemented in the following function:

```
[illegible]
```

The problem of secondary clustering is less severe than primary clustering. This is because probing sequences tend to spread out more than primary clustering. For example, if $h(K_1) = h(K_2)$, then the probing sequences for K_1 and K_2 will be identical. However, if $h(K_1) = h(K_2) - 1$, then the probing sequences will be very different. Quadratic probing is implemented in the following function:

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

FIGURE 10.3 Formulas approximating load factor hashing methods, the average number of probes for successful and unsuccessful searches. [Knuth, 1998]

	linear probing	quadratic search[1]	double hashing
successful search	$\frac{1}{2}\left(1+\frac{1}{1-LF}\right)$	$1-\ln(1-LF)-\frac{LF}{2}$	$\frac{1}{LF}\ln\frac{1}{1-LF}$
unsuccessful search	$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF}-LF-\ln(1-LF)$	$\frac{1}{1-LF}$

$$\text{loading factor} \quad LF = \frac{\text{number of elements in the table}}{\text{table size}}$$

[1] These formulas assume that the collision resolution strategy uses a secondary clustering algorithm, which causes secondary clusters to arise and subside as probing progresses from one primary cluster to the next.

Too many elements in the table increase the likelihood that clusters will form during primary or secondary clustering, and the more likely it is that these clusters are large.

Consider the case where linear probing is used for collision resolution. If K is not in the table, then starting from the position h(K), all consecutive occupied cells are checked; the longer the clusters, the longer it takes to determine that K is not in the table. In the extreme case, when the table is full, we have to check all the cells starting from h(K) and ending with h(K) − 1 mod TSize. Therefore, the search time increases with the number of elements in the table.

There are formulas which approximate the number of times a table is accessed during successful searches for different hashing methods. These formulas were developed by Donald Knuth and are mentioned by [Vitter and Flajolet] to be "among the earliest in the analysis of algorithms." Figure 10.3 contains these formulas. Figure 10.4 contains a table showing the number of searches for different percentages of occupied cells. The table in Figure 10.4 was derived from the formulas from Figure 10.3 to provide an approximation of the number of searches. This is particularly relevant for the higher percentages. For example, if 90% of the cells are occupied, then linear grouping requires 50 probes to determine that the key being searched for is not in the table. However, for the full table, the number is 10, not 100.

For the lower percentages, the approximations given by these formulas are close to the actual values. The table in Figure 10.4 indicates that if linear probing results, on average, less than two trials to find an element in the table. Since it is the number of probes, not the time, that is used to find an element, linear probing requires that elements in the table be grouped to keep performance at an acceptable level. Hashing by division generally works well for only a percentage of the table. The percentage is dependent on the specific application and the load factor. Double hashing requires more than two probes for load factors above 50% but rarely requires more than eight tries. Double hashing requires that the size of the table be a prime number to help guarantee that all the table can be used. It also requires a function that will produce a different increment for each of the original positions that are computed. The increment function for the second probe is then used for all successive probes.

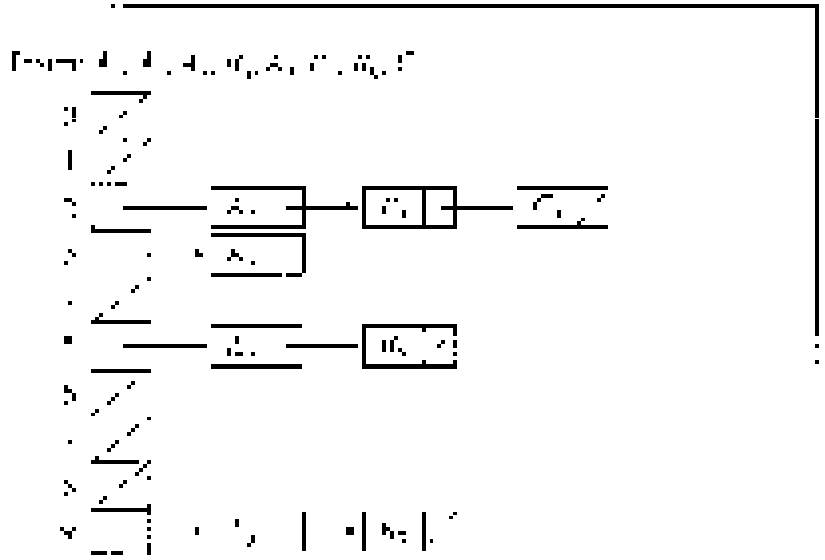
FIGURE 10.4 The average number of successful searches and unsuccessful searches for different collision-resolution methods.

LF	Linear probing		quadratic science		double hashing	
	Successful	unsuccessful	successful	unsuccessful	successful	unsuccessful
0.05	1.0	1.1	1.0	1.1	1.0	1.1
0.10	1.1	1.1	1.1	1.1	1.1	1.1
0.15	1.2	1.3	1.2	1.3	1.2	1.3
0.20	1.3	1.3	1.3	1.3	1.3	1.3
0.25	1.3	1.4	1.3	1.4	1.3	1.4
0.30	1.4	1.5	1.4	1.5	1.4	1.5
0.35	1.5	1.7	1.5	1.7	1.5	1.7
0.40	1.7	1.9	1.7	1.9	1.7	1.9
0.45	1.8	2.2	1.8	2.2	1.8	2.2
0.50	2.0	2.5	1.9	2.5	1.9	2.5
0.55	2.2	3.1	2.1	3.0	2.1	3.0
0.60	2.5	3.6	2.3	3.6	2.3	3.6
0.65	2.8	4.6	2.6	4.4	2.6	4.4
0.70	3.3	5.5	2.9	5.3	2.9	5.3
0.75	3.8	7.5	3.3	7.1	3.3	7.1
0.80	4.5	10.4	3.8	9.7	3.8	9.7
0.85	5.5	15.2	4.4	14.1	4.4	14.1
0.90	7.5	25.5	5.2	24.0	5.2	24.0
0.95	10.5	50.5	6.4	49.0	6.4	49.0

10.2.2 Chaining

Keys do not have to be stored in the table itself. For example, each position of the table can be associated with a linked list or chain of structures whose data fields store keys or references to keys. This method is called *separate chaining*, and a table of references (pointers) is called a *pointer table*. In this method, the table can never overflow since the list can be conceptually extended under the arrival of new keys, as illustrated in Figure 10.5. Further linked lists are very fast methods, but increasing the length of these lists can significantly degrade retrieval performance. Performance can be improved by

FIGURE 10.5 Inserting colliding keys into the same linked list.



in order in all these lists so that unsuccessful searches are expensive. Search is not required in most cases of using self-organizing linked lists (Pugh 1985).

This method requires additional space for maintaining pointers. The table stores only pointers and each cell requires one pointer overhead. Therefore, if n keys are to be stored, $n + M$ pointers are needed, which can be a significant overhead for very demanding space requirements.

A variation of chaining called *coalesced hashing* (or *coalesced chaining*) combines linear probing with chaining. In this method, the first available position is found for a key colliding with another key, and the index of this position is stored with the key stored in the table. In this way, sequential searches for keys in the table can be avoided by directly jumping to the next element on the linked list. Each position in the table stores a record with two members: one for a key and next with the index of the next key which is hashed to this position. Available positions are also linked by `next`. The last element in the table is the end of a chain. This method requires $2M$ extra words of memory for the table in addition to the space required for the keys. This is less than for chaining, but the table size limits the number of keys that can be hashed into the table.

An overflow area of some fixed size can be allocated to store keys that don't have room in the table. This area should be located dynamically if implemented as a linked list.

Figure 10.5 illustrates an example where coalesced hashing puts a colliding key in the last position of the table. In Figure 10.5(a), a collision occurs in Figure 10.5(b), h_3 is put in the next cell of the table which is found occupied by k_6 when it arrives. Hence k_3 is stored at the first accessible location from location 5. In Figure 10.5(c), two new colliding keys are added to the corresponding slots.

FIGURE 10.6 Coalesced hashing puts colliding keys in the lowest available position of the table

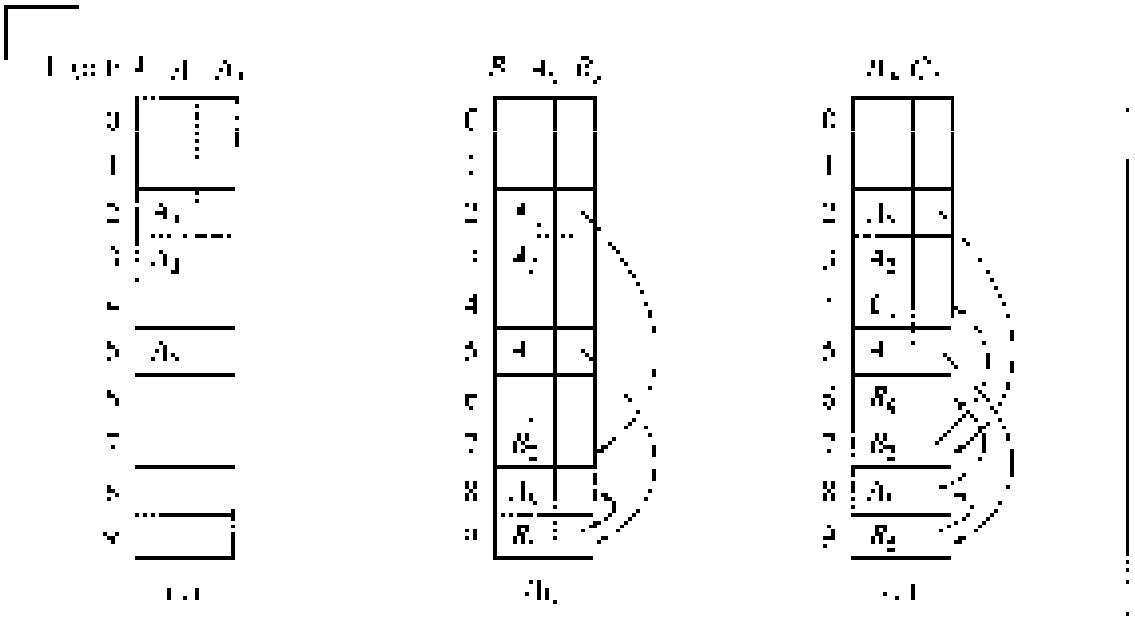


Figure 10.7 illustrates coalesced hashing with the same data as in Figure 10.5. Notice that the first collision (when A_1 is inserted) causes it to be placed in the last available position of the table (position 9). The second collision (when A_2 is inserted) causes it to be placed in position 8. The third collision (when A_3 is inserted) causes it to be placed in position 7. The fourth collision (when A_4 is inserted) causes it to be placed in position 6, and so on. Only when the table is completely filled will a key be placed in its home position.

10.2.3 Bucket Addressing

Another way to reduce the collision problem is to store more than one key per address in the table. This solution is called bucket addressing. A bucket is a block of storage space large enough to store multiple records.

By storing multiple records in each bucket, the problem of collisions is not totally avoided. If a bucket is already full when a key is hashed to it, that key must be stored somewhere else. By comparison to the open addressing approach, the collision resolution strategy for bucket addressing is to store the key in the next bucket in sequence if the home bucket is full. (Wrap around from the last bucket to the first bucket is used.) If the next bucket is also full, the search continues to the next bucket, and so on, until a bucket with an available slot is found. If the search comes back to the home bucket, then the table is full.

The collision resolution algorithm must be changed for bucket addressing. Instead of marking each slot to indicate whether the slot is occupied, we must keep track of how many records are stored in each bucket. This can be done by storing a count field with each bucket. When searching for a key, the count field tells us how many slots in the bucket to look at before concluding that the key is not present. Similarly, when inserting a key, the count field indicates which slot in the bucket is available for the new key. Figure 10.8 illustrates how a bucket addressing implementation of coalesced hashing would look.

FIGURE 10.7 Collision resolution by chaining.

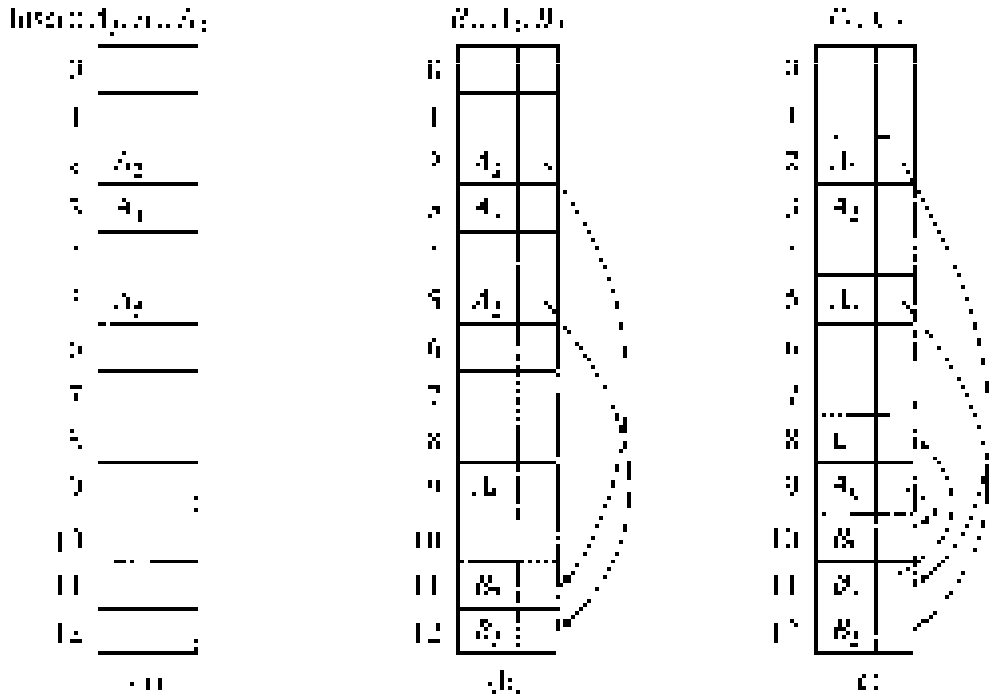


FIGURE 10.8 Collision resolution with two-record buckets and overflowing buckets.

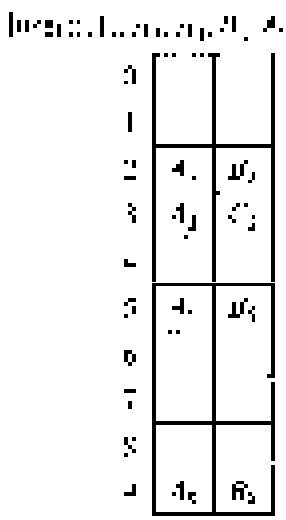
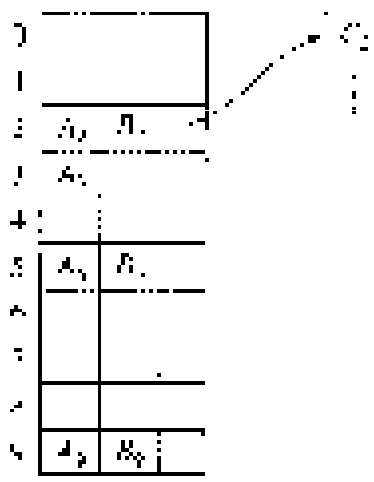


Figure 10.9 Collision resolution with buckets and overflow area.



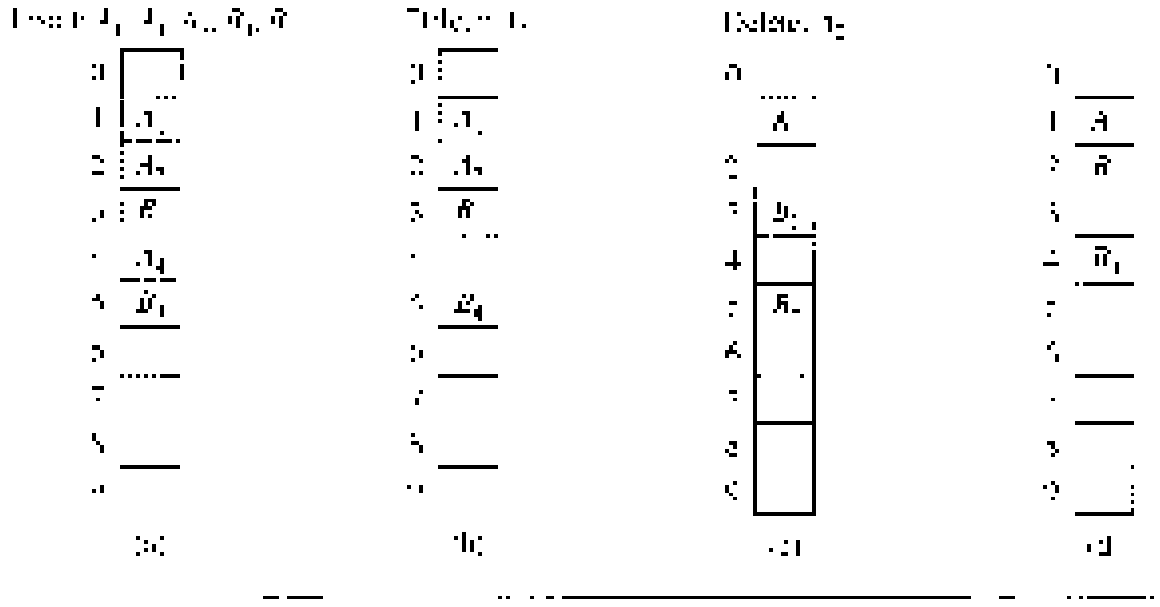
10.3 Deletion

How can we remove data from a hash table? With a chaining method, deleting an element leads to the deletion of a node from a linked list including the element. For other methods, a deletion operation may require more careful rearrangement of the table entries, except for the rare occurrence when a perfect hash function is used.

Consider the table in Figure 10.10a in which five keys are stored using linear probing. The keys have been entered in the following order: A_1, A_2, A_3, B_1, B_2. After A_2 is deleted and position 4 is freed (Figure 10.10b), we try to find B_2 by first checking position 4. But this position is now empty, so we may conclude that B_2 is not in the table. The same result occurs after deleting A_3 and marking position 2 as empty (Figure 10.10c). Then the search for B_2 is unsuccessful, since if we are using linear probing, the search terminates at position 2. The situation is the same for the other open addressing methods.

If we leave deleted keys in the table with no markers indicating that they are no valid elements of the table, any subsequent search for these entries does not terminate prematurely. When a new key is inserted, it is assigned a key which is truly unoccupied. However, for a large number of deletions and a small number of additional insertions, the table becomes overloaded with deleted records, which increases the search time because the open addressing method treats the deleted elements. The table should be compacted after a certain number of deletions by moving undeleted elements so that cells occupied by deleted elements are freed. Cells with deleted elements which are not overwritten by this process are marked as deleted, as described in the next section.

FIGURE 10.10 Linear search in the situation where both insertion and deletion of keys are permitted



10.4 Perfect Hash Functions

All the cases discussed so far assume that the body of data is not precisely known. Therefore, before a function can be selected, it must be considered ideal in the sense that it immediately locates a key in its proper position and avoids collisions. In most cases, some collision would occur, so it must be included, since, strictly speaking, a key would not be found in its correct location. Also, the size of the table is usually determined by the number of keys in the data. Moreover, the table is assumed to be large enough to accommodate all the incoming data. Moreover, the table is assumed to be sufficiently free of collisions. A larger table has a smaller number of collisions, provided the hash function is selected appropriately. However, the table size must be limited. All these assumptions are valid only if the body of data to be stored in the table is known precisely and does not change.

In many applications, however, the body of data is not fixed, and a hash function cannot be devised after the data is known. Such a function must be chosen carefully before the data is entered. Ideally, the function must be perfect. A hash function is perfect if it maps each key into a unique location in the table so that no two keys are mapped to the same location. Such a hash function is also called a one-to-one function. Whether or not a perfect hash function is used, it is always desirable to minimize the number of collisions and to have a good method for resolving collisions when they occur.

Perfect hash functions are rare because they require a precise knowledge of the body of data to be stored in the table. Even if such knowledge is available, finding a perfect hash function is not always easy. Nevertheless, perfect hash functions are desirable because they guarantee efficient access to the data.

Algorithms for creating perfect hash functions usually require lengthy computations to find a perfect hash function. As already mentioned for 50 elements and a 100-cell array, one in 10 million is perfect. Other functions have collisions.

10.4.1 Cichelli's Method

One algorithm for constructing minimal perfect hash functions was developed by Richard J. Cichelli. It is based on the relative values of the letters in the words. The function is of the form

```
hashval = length(word) + g(first character) + g(last character)
```

where g is the function to be constructed. The function g assigns values to letters so that the resulting function returns unique hash values for all words in a predefined set of words. The values assigned by g to particular letters do not have to be unique. The algorithm uses three parts: construction of the letter occurrences, ordering the words, and searching. The last step is the heart of this algorithm and uses an auxiliary function try(). Cichelli's algorithm for constructing g and h is as follows:

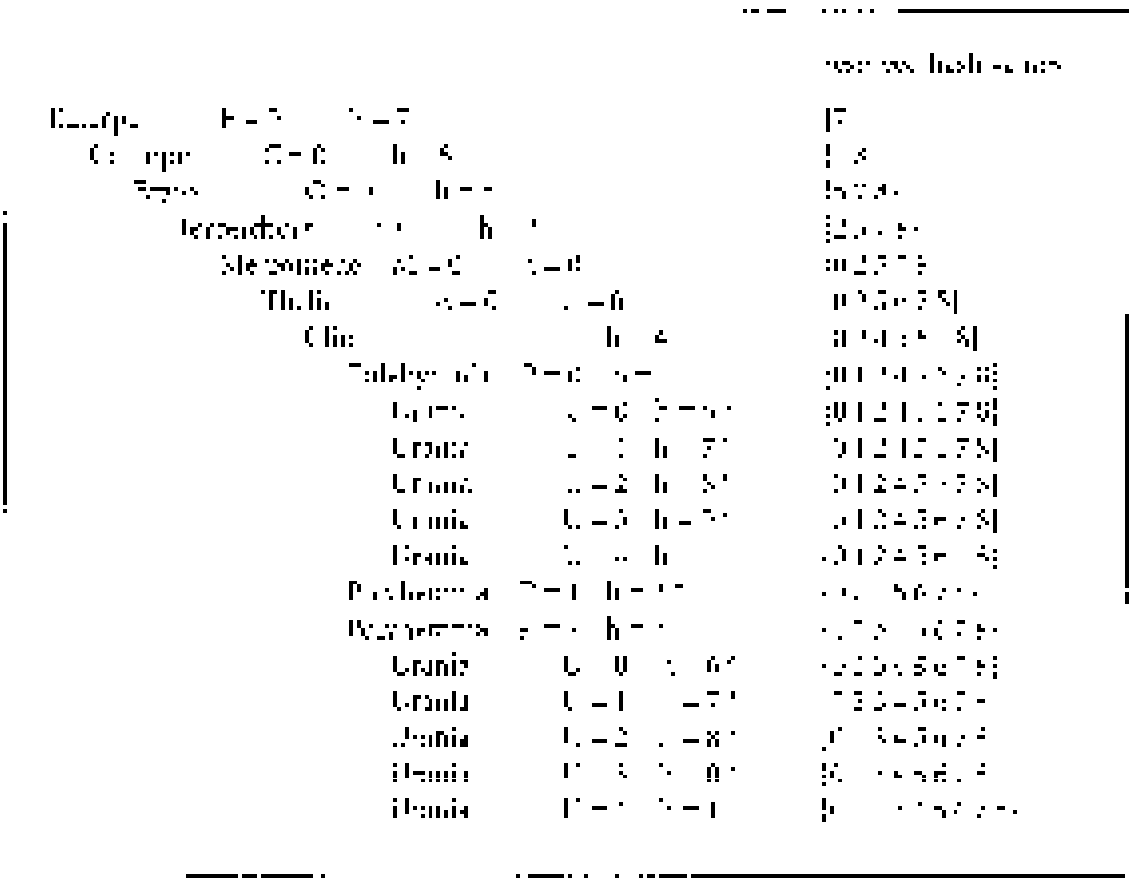
```
Choose a suitable MAX;
Compute the occurrences of each first and last letter in the set of all words;
Order all words in accordance to the occurrences of the first and the last letters;
Search(wordlist)
If wordlist is empty
   Exit;
Word = first word from wordlist;
Wordlist = wordlist without the first word just used;
If the hash value of word can be assigned a value
      Try(word, 1, 1); // 1 signifies that values already assigned
      If success
         Search(wordlist);
      Put word in the beginning of wordlist and mark its hash value;
Else if either the first or the last letter has not been used
   For each a,z in {1... MAX}
      Try(word,a,z);
      If success
         Search(wordlist);
         Put word in the beginning of wordlist and mark its hash value;
Else if either the first or the last letter has been used
   For each a in {1... MAX}
      Try(word,a,1) or Try(word,1,z);
      If success
        Search(wordlist);
         Put word in the beginning of wordlist and mark its hash value;

Try(word,Ta,Tz){ // Ta and Tz are parameter values
   If h(word) has been assigned a value
         return failure;
```

```
    if word = '' then exit;
      assign the initial value and/or function designator of function identifier;
      while
      not eof(input)
    do begin
  end;
```

We can use the algorithm in the solution to Exercise 8.1 for the purpose of the minimal perfect hash function. Since the Modula-2 program is very long, we have not included it here. A simple solution that does not require the minimal perfect hash function is to use a binary search tree. Each node in the tree contains a word and the number of the word in the dictionary. To look up a word, we start at the root of the tree. If the word we are looking for is alphabetically before the word at the root, we go to the left child of the root; otherwise we go to the right child. We continue in this way until we find the word we are looking for or we encounter a leaf node. If we encounter a leaf node, the word is not in the dictionary. Figure 8.1 shows a binary search tree for the words in Exercise 8.1. The number of comparisons required to look up a word is at most the depth of the tree. For example, looking up the word *for* requires three comparisons, while looking up the word *the* requires only two comparisons.

Now we present the assembly language program for the minimal perfect hash function. Figure 8.2 shows the assembly language program for the function `Hash` in Exercise 8.1. First the word is looked up in the dictionary. If the word is found, the function returns the number of the word in the dictionary. Otherwise, the function returns 0. For example, if the word is found in the dictionary, the function returns 1. If the word is not found in the dictionary, the function returns 0. For example, if the word is not found in the dictionary, the function returns 0. For example, if the word is not found in the dictionary, the function returns 0. For example, if

FIGURE 10.11 Subsequent invocations of the servicing procedure with Algorithm 10.2 (Cichelli's algorithm) assign the indicated values to letters and to the set of reserved word hash values. The last call indicates failure.



(e.g., Lewis and Cook 1988). The problem with this approach is that it is difficult to find a generally applicable grouping function aimed at finding minimal perfect hash functions.

Both these ways of modifying hash functions and combining them are potentially successful. If the same Cichelli's algorithm is used. Although Cichelli concludes his article with the adage "When all else fails, try brute force," the attempts to modify his approach included devising more efficient searching algorithms that circumvent the need for brute force. One such approach is incorporated in the following algorithm.

10.4.2 The FHCD Algorithm

An extension of Cichelli's approach was proposed by Sebastian Sager. The FHCD algorithm is an extension of Sager's method and is discussed in the next section. The FHCD algorithm (Jonnet et al. 1992) searches for minimal perfect hash functions of the form

$$h(\text{word}) = (g(\text{word}[1]) + g(\text{word}[2]) + g(\text{word}[3])) \bmod m$$

(modulo m hash), where g is a function to be determined by the algorithm and m is the number of words in the reserved word list. The variables $\text{word}[1]$, $\text{word}[2]$, and $\text{word}[3]$ are the first, second, and third characters of the word, respectively. Each function g is represented as a table of characters $c_1, c_2, \ldots, c_n$ and corresponding

to a prime table size p. (Recall that universal hash functions are obtained according to the formula

$$h_0 = ((a_0 k_0 + \cdots + a_r k_r) \bmod p) \bmod n$$
$$h_1 = ((a_1 k_0 + \cdots + a_{r+1} k_r) \bmod p) \bmod n$$
$$h_2 = ((a_2 k_0 + \cdots + a_{r+2} k_r) \bmod p) \bmod n$$

where n is the number of all words in the hashed dictionary, a parameter usually found in the process, and $a_0, a_1, a_2, \ldots$ are numbers generated in table T for r. The function g is found in three steps: mapping, ordering, and searching.

In the mapping step, a simple, undirected graph $G = (V, E)$ is created. The randomness of function h essentially guarantees the uniqueness of these triples (though they can be unique, new values h_i are generated. Next, a dependency graph is built. It is a bipartite graph with half of its vertices corresponding to the h_0 values and labeled 0 through $n-1$, and the other half to the h_1 values and labeled 0 through $n-1$. Each word corresponds to an edge of the graph between the vertices $h_0(\text{word})$ and $h_1(\text{word})$. The mapping step is expected to run in $O(n)$ time.

As an example, we again use the set of names of the nine Muses to generate three values h_i for standard hash function h as given above, and store those triples in a table. The set of nine triples is computed, as shown on page 521. Figure 14.14b contains a corresponding dependency graph with $n = 5$. Note that some vertices cannot be connected to any other vertices, and some pairs of vertices can be connected by more than one arc.

In the ordering step, we assign all the vertices so that they can be partitioned into a series of levels. After a certain number q of vertices is established, then a level N_q of vertices is defined to be a set of those vertices with the property that each vertex in N_q is connected to one vertex in N_{q-1} and to no other vertex in N_{q-1}. The sequence is initialized with a vertex of maximum rank in G. Then, for each successive position q of the vertex, a vertex in N_q is selected from among the vertices that are connected to the vertex in N_{q-1} which has maximal degree. When no such vertex can be found, any vertex of maximal degree which is not among the already selected vertices is used to start a new component.

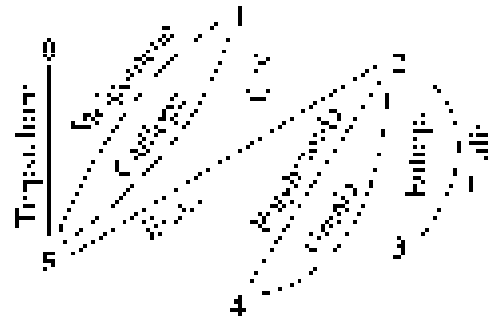
In the last step, we map the vertices to the keys level by level. The first vertex in N_0 is mapped to 0, and the next vertex in N_1 to 1, etc. For the other vertices, because of the construction of the graph, we have the following relation: $g(v_{q+1}) = (\text{hit}[v_{q+1}] - g(v_q)) \bmod n$. This means that $g(v_{q+1})$ has the value that, when added to $g(v_q)$, results in $\text{hit}[v_{q+1}]$. (We already have defined g on the vertex v_q which has already been processed.) Applying this rule to the case when v_{q+1} is the vertex v_1, for which g would be undefined, $g(v_{q+1})$ is given. The second part of the formula is found to ensure that the values always remain in the range $0, \ldots, n-1$. Because the function h is universal, the values will be evenly distributed over the hash table, but a collision may occur and the function g may not exist.

The search step for the given word starts with computing the triple (h_0, h_1, h_2) and then the value $h = (g(h_0) + g(h_1)) \bmod n$. If this value is equal to the index of the edge corresponding to the word, then the search is successful; otherwise, the word is not in the table. For example, for the word Thalia, $(h_0, h_1, h_2) = (3, 4, 2)$, and $h = (g(3) + g(4)) \bmod 5 = (1 + 3) \bmod 5 = 4$, which is the index of the edge corresponding to Thalia.

FIGURE 16.12 Applying the FHCD algorithm to the names of the nine Muses.

Value of i	h_0	h_1	h_2
0 (Calliope)	0	1	2
1 (Clio)	2	1	4
2 (Erato)	1	2	5
3 (Euterpe)	0	2	4
4 (Melpomene)	0	3	0
5 (Polyhymnia)	0	3	1
6 (Terpsichore)	0	1	3
7 (Thalia)	0	2	3
8 (Urania)	0	2	4

(a)



(b)

Level	Node	Area
0	2	
1	3	Erato
2	7	Thalia, Melpomene
3	4	Clio, Polyhymnia, Urania
	5	Euterpe, Terpsichore
4	0	Terpsichore

(c)

Level	Vertex	g-value	
0	2	2	
1	3	3	h_0(Erato) $= 0 + 2 \cdot 1 \bmod 9 = 2$
2		4	h_0(Thalia) $= 0 + 2 \cdot 1 \bmod 9 = 2$
3	1	4	h_0(Melpomene) $= 0 + 3 \cdot 1 \bmod 9 = 3$
3	4	2	h_0(Clio) $= 0 + 2 \cdot 1 \bmod 9 = 2$
3	4	2	h_0(Polyhymnia) $= 0 + 2 \cdot 1 \bmod 9 = 2$
3	4	3	h_0(Urania) $= 0 + 2 \cdot 1 \bmod 9 = 2$
4	3	4	h_0(Euterpe) $= 0 + 2 \cdot 1 \bmod 9 = 2$
4	5	4	h_0(Terpsichore) $= 0 + 2 \cdot 1 \bmod 9 = 2$
4	5	1	h_0(Thalia) $= 0 + 2 \cdot 1 \bmod 9 = 2$

(d)

Location	g
0	1
1	4
2	3
3	4
4	2
5	0

(e)

From this, $h(\text{Erato}) = (g(1) + g(2)) \bmod 9 = (4 + 3) \bmod 9 = 7$. This means that position 2 of the hash table is no longer available. The new g value, $g(1) = 4$, is retained for later use.

Now $i = 4$ and we start with $S_4(y) = \{\text{Calliope, Melpomene}\}$. The v_0 values for both words are retrieved from the table of nodes, and the $g(h_0)$ values are equal to 0 for both words, so $h_0 = v_0$ for both of them. Now we must find a common $g(h_1)$ value

such that the function does not produce collisions. Such a function is called a perfect hash function. Perfect hash functions are desirable because they guarantee that only one probe will be necessary to access any entry. Unfortunately, perfect hash functions are rare. However, if the set of possible keys is known in advance (as is the case with the reserved words in a compiler), then a perfect hash function can always be found. Figure 10.5 shows two values of the function g. The program can store and use these values to compute the perfect hash function for any key. However, if the set of keys is not known in advance, it is hard to find and store a perfect hash function. Even worse, adding a new key may require recomputing the entire function and reorganizing the table. Therefore, perfect hash functions are usually cost effective only when the set of keys is static, frequently used, and known in advance. Even then, the cost of finding a perfect hash function must be weighed against the cost of using a slightly less efficient function.

10.5 Hash Functions for Extendible Files

All the methods discussed so far work on tables of fixed size. This is a reasonable assumption for compiler symbol tables but is too restrictive for many applications. After all, tables grow and shrink dynamically as records are inserted and deleted. Data structures must be able to adapt to this change in size. Some hashing techniques can be used in this situation, such as open addressing or hashing with chaining, but some of them work poorly in a dynamic environment. New techniques have been developed which specifically take into account the variable size of the table or file. We can distinguish two classes of such techniques: directory schemes and directoryless schemes.

In the directory schemes, access is mediated by an access to a directory or an index of keys or file structures. There are several techniques and modifications to this technique in this category of the directory schemes. We mention only three: dynamic hashing (Krot. 1978), virtual hashing (Larson 1978), and extendible hashing (Fagin et al. 1979). All three methods distribute keys among buckets in a similar fashion. The main difference is the structure of the index (directory). In expandable hashing, or virtual hashing, a binary tree is used as an index of buckets. On the other hand, in extendible hashing, a directory of records is kept in a table.

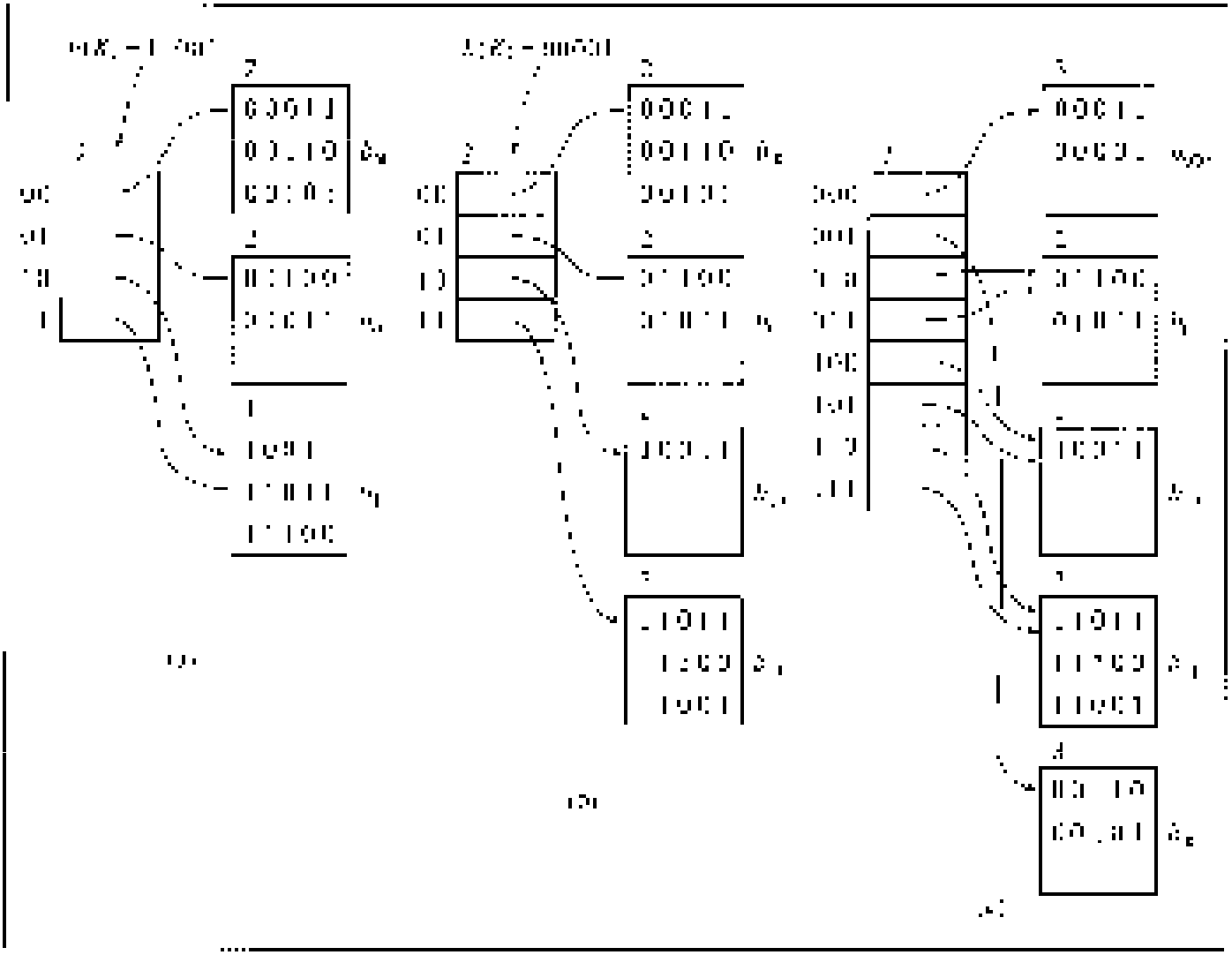
One directoryless technique is *linear hashing*, defined as "one hashing which may dynamically change its hashing function" (Litwin 1980). This concept of hashing has been somewhat of a misnomer for the lack of a directory. An example of this approach is linear hashing (Litwin 1980). In the following pages, one method from each category is discussed.

10.5.1 Extendible Hashing

Assume that a hashing technique is applied to a dynamically changing file composed of buckets, and each bucket can hold only a fixed number of items. Extendible hashing accesses the data stored in buckets indirectly through an index that is dynamically adjusted to reflect changes in the file. The characteristic feature of extendible hashing is the organization of the index, which is a two-dimensional table.

A hash function applied to a certain key indicates a position in the index and not in the file or table as always. Values returned by such a hash function are called pseudokeys. In this way, the file responds to changes in size when data are added to or

FIGURE 10.13 An example of extendible hashing.



...dex directory from it since these changes are indicated in the node. Only one hash function can be used, but depending on the size of the index only a portion of the address is actually used. A simple way to achieve this effect is by looking at the address as a string of bits from which only the leftmost bits can be used. The number is called the depth of the directory in Figure 10.13a. Its depth is equal to two.

As an example, assume that the hash function h generates addresses of five bits. Let the key in the string 01011 and the depth is two. Then the two leftmost bits, 01, are considered as the position in the directory containing the pointer to a bucket in which the key can be found or into which it is to be inserted. In Figure 10.13 the values of 2 are shown in the buckets. But these values only represent the keys that are actually stored in these buckets.

Each bucket has a local depth associated with it that indicates the number of leftmost bits in which the keys in the bucket have the same value. For example, the bucket B_1

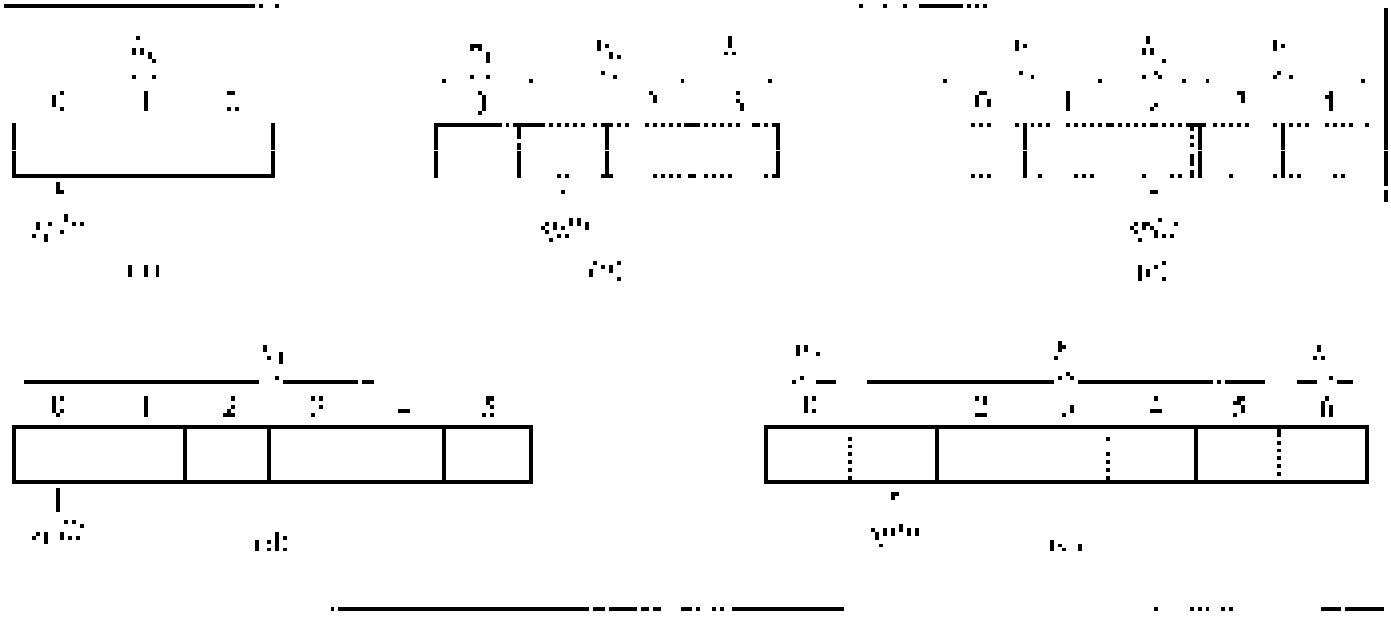
holds all keys for which h(K) starts with 00. More important, the local depth indicates which of the buckets can be accessed from one location in the directory or from two. In the first case, when the local depth is equal to the depth of directory, it is necessary to change the size of the directory after the bucket is split in the case of overflow. When the local depth is smaller than the directory depth, splitting the bucket does not require changing all of the pointers pointing to this bucket so that they point to the newly created one. Figure 10.10(b) illustrates this case. After a key with h value 11001 arrives, it is two first bits (since depth = 2) direct it to the fourth position of the directory from which it is sent to the bucket b_3, which contains keys whose h value starts with 11. An overflow occurs and it is split into b_3, the new name for the old bucket and b_4. The local depths of these two buckets are set to two. The pointer from position 11 points now to b_4, and the keys from b_3 are redistributed between b_3 and b_4.

The situation is more complex if overflow occurs in a bucket with a local depth equal to the depth of the directory. For example, consider the case when a key with h value 10001 arrives at the table in Figure 10.10(b) and is hashed through position 10 into the bucket b_2. A split occurs, but the directory has no room for the pointer to the new bucket. As a result, the directory is doubled in size so that its depth is now three. Now b_2 becomes b_{100} with a new local depth, and the new bucket is b_{101}. All the keys from b_2 are redistributed between the new buckets. Those whose h value starts with 1000 became elements of b_{1000}, the remaining keys whose h value starts with 1001 are elements of b_{1001}. The elements of the new directory have to be set to their pointers, which, having undergone doubling, now look as follows: 000 and 001 point to b_0; 010 points to b_1; 011 points to b_2; 100 points to b_{1000}; 101 points to b_{1001}; 110 points to b_3; and 111 points to b_4. All the pointers except for the pointer from 100 to b_{1000} and from 101 to b_{1001} have the same values as before the overflow occurred.

The following specification serves as a basis for writing executable insertion:

```
insert(key, record)
    h = h(key);
    p = directory[depth(directory) bits from left of h];
    if space available in bucket b_p pointed to by p
        place record here;
    else split bucket b_p and b_q and p;
        if depth(b_p) < depth(directory)
            distribute keys from b_p between b_p and b_q;
            store pointers to b_p and b_q in directory[p] and directory[p + 1];
        else double the directory and increment depth;
            call insert(key, record);
```

An important advantage of the approach just described is that it requires at most two accesses to the file if the directory resides in RAM memory. Because the directory is relatively small, it can be kept in main memory. The cost of updating and doubling it is negligible for large files. The main disadvantage is that the directory can grow exponentially if the distribution of keys is highly nonuniform. For example, if the keys in the file we are considering start with 0, then the directory will have to be doubled even if the number of buckets is small. Such a situation is unacceptable because the directory will occupy most of the



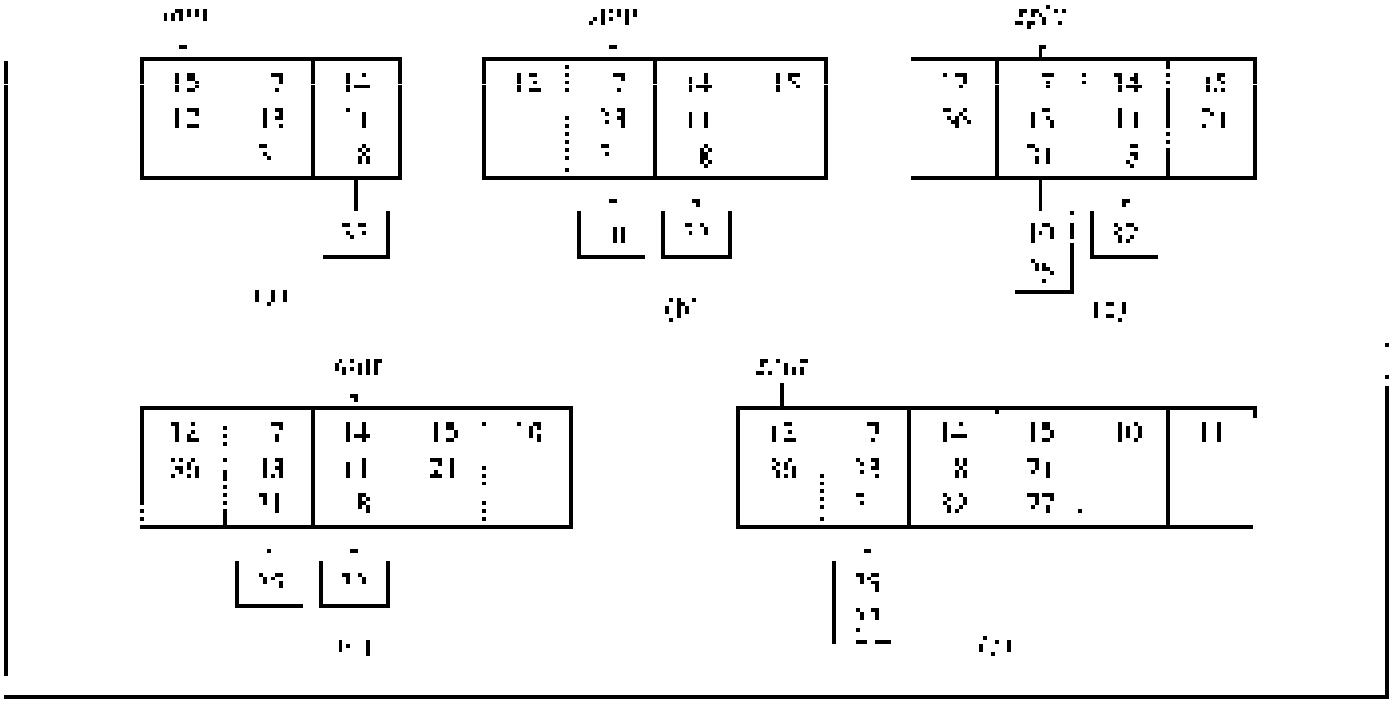
the key was split. The new value of the pointer is found by appending a 1 bit to the end of the bit sequence. For example, the pointer value for 101 is 1101, and the pointer value for 111 is 1111. The pointer value for 000 remains 000 because the bit appended to the end of the bit sequence is a 0. Figure 10.14(c) shows the directory after the bucket pointed to by 101 has been split. The bucket pointed to by 1101 has been created, and the keys in the original bucket for 101 have been distributed between the original bucket and the new bucket. Note that the bucket pointed to by 001 has not been split yet, so both 001 and 1001 point to the same bucket.

Now it is clear why the directory is doubled when $i = j$. If $i = j$, then only one pointer to bucket b exists. After b is split, two pointers are required to point to the two buckets. Doubling the directory accomplishes this task. Subsequent splits of buckets do not require the directory to be doubled, as other pointers to the buckets exist.

10.5.2 Linear Hashing

Extendible hashing allows the hash table to expand without overflow chaining, but it requires storage space for an entire directory. Linear hashing, developed by Witold Litwin, avoids this necessity because new buckets are created by splitting one bucket in a predetermined linear way, so there is no need to retain indexes. Instead, a pointer `split` indicates which bucket is to be split next. After the bucket pointed to by `split` is split, the keys in that bucket are distributed between that bucket and the newly created bucket, which is added to the end of the table. Figure 10.14 continues a sequence of four splits in which $i = 3$. Initially, the pointer `split` is zero. If a hashing function exceeds a certain level, a new bucket is created, keys from bucket zero are distributed between bucket zero and bucket one, and `split` is incremented. How is this distribution performed? If only one hash function is used, then it cannot map keys from buckets zero to bucket zero before and after splitting. This means that one function is not sufficient.

Figure 10.15 Inserting keys to buckets and overflow areas with [illegible] addressing techniques.



At each level of splitting, linear hashing applies two hash functions $h_{level}(K)$ and $h_{level+1}(K)$, for $h_{level}(K) = K \bmod (TSize \cdot 2^{level})$. The first hash function, h_{level}, addresses keys to buckets which have not yet been split at the current level. The second function, $h_{level+1}$, is used for locating keys to already split buckets. The algorithm for inserting keys is as follows:

```
initialize split = 0; level = 0;

insertHashingTwoLevel(K)
   if h_level(K) < split      // bucket h_level(K) has been split
      hashAddress = h_level+1(K);
   else hashAddress = h_level(K);
   insert K into corresponding bucket or overflow area if possible;
   while (keyNotInserted && bucket or overflow area is full)
      create a new bucket with index split + TSize * 2^level;
      redistribute keys between bucket split and bucket split + TSize * 2^level;
      split++;
      if split == TSize * 2^level // all buckets on the current
                                  // level have been split;
         level++;                 // proceed to the next level
         split = 0;
      insert K into corresponding bucket;
```

It may still be necessary when we split a bucket. Note that k/n is the expected number of records in a bucket, and the loading factor is used to decide whether or not to split a bucket. This threshold has to be kept somewhat low, and is empirically determined by the programmer using the system. For this reason, many methods have been developed to allow splitting to be postponed, at least temporarily, while other buckets are searched for space. One such method, called *overflowing*, is discussed in the next section. Consider the situation in Figure 10.15(a) in the figure. Now $h_2(K)$ is used to split $b_2(K)$ into $b_2(K)$ and $b_3(K)$. Suppose that there are a large number of records in $b_2(K)$, so that the loading factor is greater than the acceptable threshold. Instead of splitting $b_2(K)$ immediately, we may choose to allow overflowing to occur. Suppose that there is room in $b_3(K)$ for a few more records, but not enough room for all the records that would hash to $b_2(K)$. Then the overflow records may be stored in a special overflow bucket, and will be retrieved whenever $b_2(K)$ is searched. Figure 10.15(b) illustrates this idea. Note that the overflow bucket is associated with $b_3(K)$ because $b_3(K)$ is the bucket that would have been split next. Overflowing can be repeated until $b_3(K)$ is full, at which time $b_3(K)$ and its overflow bucket will be split. Of course, it is possible that too many records will hash to the overflow bucket, causing it to overflow as well. In this case, another overflow bucket can be created, and so on. Note that overflowing results in fewer splits, but requires additional accesses to the overflow buckets during retrieval. This is a typical space–time trade-off: By splitting frequently, we conserve space but must search through fewer records on the average. By splitting rarely, we save time but waste space. The constants in this trade-off will depend on the application.

Another approach to controlling splitting allows records to be hashed initially to a relatively small number of buckets, with splitting occurring when some predetermined threshold is reached, such as 75% of the buckets filled. Then, rather than splitting the bucket that has overflowed, a bucket is added at the end of the directory, and some bucket (usually the one about to overflow) is split and half its contents moved to the new bucket. This method has been called *linear hashing* by Litwin (1980) and *spiral storage* by Martin (1979). Many variations on the idea have been explored, including schemes for controlling the size of buffers holding overflow records. Larson (1980) provides a simulation analysis of several versions of linear hashing, and Mullin (1985) develops a tidy mathematical model.

The following example illustrates the linear hashing technique. Assume that we have a hash function $h(K)$ that produces integers uniformly distributed between 0 and 63. Thus the hash function is appropriate for a file with eventually 64 buckets. Assume that the bucket capacity is 4 records, and that we will start with 4 buckets in the file. Thus we will use the hash function $h(K) \bmod 4$ to assign records to the initial buckets. Figure 10.16 shows the file after the records with keys 2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 37, 41, 43, and 47 have been inserted. Note that the records are not sorted by key within the buckets.

K	$h(K)$	Number of Buckets	Number of Cells	Loading Factor	Split	Hash Functions	
		4	0 – 3	9/16 = 56%	0	$h \bmod 4$	$h \bmod 8$
0		4	4 – 7	8/16 = 50%	1	$h \bmod 4$	$h \bmod 8$
		5	8 – 11	10/20 = 50%	2	$h \bmod 4$	$h \bmod 8$
21	1	5	12 – 15	11/20 = 55%	3	$h \bmod 4$	$h \bmod 8$
35	3	6	16 – 19	12/24 = 50%	0	$h \bmod 8$	$h \bmod 16$
33		6	20 – 23	13/24 = 54%	1	$h \bmod 8$	$h \bmod 16$
		6	24 – 27	14/24 = 58%	2	$h \bmod 8$	$h \bmod 16$
57	1	7	28 – 31	15/28 = 54%	3	$h \bmod 8$	$h \bmod 16$
57	1	7	32 – 35	16/28 = 57%	4	$h \bmod 16$	$h \bmod 32$
		8	36 – 39	17/32 = 53%	5	$h \bmod 16$	$h \bmod 32$

Note that linear hashing requires the use of some overflow area because the order of splitting is predetermined. In the case of files, this area must be on the disk as well. This area can be explicit and different from buckets, but it can be introduced somewhat in the spirit of coalesced hashing by utilizing empty space in the buckets (Mullin, 1985). In a directory scheme, on the other hand, an overflow area is not necessary, although it can be used.

As in a directory scheme, linear hashing increases the address space by splitting a bucket. It also redistributes the keys of the split bucket between the buckets that result

from the split. Because no indexes are maintained with linear hashing, the method is faster and requires less space than previous methods. The increase in efficiency is particularly noticeable for large files.

10.6 Case Study: Hashing with Buckets

The most serious problem to be solved in programs that rely on a hash function to insert and retrieve items from an unordered block of data is resolving collisions. Depending on the technique, allowing deletion of items from the table can significantly increase the complexity of the program. In a case study, a program is developed that allows the user to insert and retrieve items from the file name interactively. This file-name-to-number conversion is frequently undertaken alphabetically at the end of the assembly. The program is written with the hash function in the first module, which is used to map the name of the symbolic variable into a file of buckets, initialized to empty. The number of the bucket is passed to the corresponding bucket in this file and stored in the file-name-to-number table. The file of buckets can be deleted when the assembly is complete.

The variable-length character string is the input to the hash algorithm. The hash function must be able to handle any reasonable length string and use as much of the string as necessary to produce a good distribution. Normally, a series of names with regularity in the pattern of characters should be spread evenly by the hash function. It is unlikely that the function will be perfect, but the function should minimize the number of collisions and the number of buckets that must be searched when a collision occurs. If the end of the table is reached during a search, the search wraps around to the beginning of the table. The search is considered unsuccessful only when the entire table has been searched.

After the initialization, each name is scanned. The name is converted to a number by the hash function. If the bucket is empty, the name is inserted. Otherwise, the bucket is searched sequentially for the name. If the name is found, the number is returned. If the end of the bucket is reached, the search continues sequentially at the beginning of the bucket. If the starting point is reached, the bucket is full and the program terminates with an error message.

If the user chooses to erase the undefined entries at overflow and to reference count, then all undefined entries of outbuf are sorted using an external sort. The sort is called both to outbuf and to a temporary file. The file contains the number of entries in outbuf followed by the entries. The sort is used to sort the entries by the name field. After the sort, the file is read back into outbuf. The reference count is then set to 1 for each entry.

After this indirect sorting is accomplished, the data in outbuf are sorted in alphabetical order. This is accomplished by transferring entries from outbuf to an array, using the count included in parentheses. That is, by going down the array and retrieving the entry in outbuf through the address stored in the array. Finally, the file of names is deleted and all open files are closed.

Here is an example. If the contents of the original file are

```
Adam 123-4567          Brenda 345-6789        Brandon 789-7890
Charles 987-1122       Jennifer 765-4567      Kathleen 901-5678
Patrick 789-6532       Raymond 890-9012       Theresa 567-7890
```

The hashing generates the numbers:

```
Kathleen 901-5678      Jennifer 765-4567      |
Adam 123-4567          Brenda 345-6789        |
Raymond 890-9012       Theresa 567-7890       |
```

and the file overflows:

```
Brenda 345-6789        Charles 987-1122       |
Jennifer 765-4567      Patrick 789-6532       |
```

The overflow buckets are included in the hash table because this provides more space for buckets and helps keep overflows small. The hash table is

```
Adam 123-4567  Kathleen 901-5678  Jennifer 765-4567  Brenda 345-6789  Theresa 567-7890
Charles 987-1122  Patrick 789-6532  Raymond 890-9012
```

and the overflow:

```
Brandon 789-7890       |Kathleen 901-5678       |
Jennifer 765-4567      |Brenda 345-6789         |
Raymond 890-9012       |Theresa 567-7890        |
```

and overflow:

```
Brandon 789-7890       |Charles 987-1122        |
Jennifer 765-4567      |Patrick 789-6532        |
```

Assuming a multiplicative hash of Maggie 987-9876 and deletion of Brenda 345-6789 happens only sometime:

```
Brandon 789-7890       |Charles 987-1122        |
Maggie 987-9876        Patrick 789-6532       ||
```

After the overflow buckets are compacted, the buckets from overflow are double-checked to make sure no high-use buckets

```
Kathleen 901-5678      Jennifer 765-4567      |
Adam 123-4567          Theresa 567-7890       |
Raymond 890-9012       Patrick 789-6532       |
Charles 987-1122       Maggie 987-9876        |
Patrick 789-6532
```

The table is stored and the outcomes:

```
Adam 123 4567          |Jennifer 765 4567       |
Charles 987 1122       |Kathleen 901 5678       |
Maggie 987 9876        |Patrick 789 6532        |;
Raymond 890 9012       |Theresa 567 7890        |]
```

Figure 11.5 contains the code for this program.

FIGURE 16.16 Implementation of hashing using buckets.

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h> // unlink(), rename()

const int BucketSize = 2, DataSize = 3, StrLen = 20;
const int RecordLen = StrLen;

class File {
public:
    File() { empty='#'; delMarker='~'; }
    void processFile(char*);
private:
    const char empty, delMarker;
    long *pointers;
    ostream& out(ostream&, const long*);
    int  hash(char*);
    void swap(long& a, long& b) {
        long tmp = a; a = b; b = tmp;
    }
    File& get(char*);
    File& insert(char line[]) {
        get(line); insertion(line);
    }
    File& insertion(char*);
    File& remove(char*);
    File& update(int,int,int);
    File& operator[](int);
    File& sort(File&);
    File& combine(File&,File&);
};

unsigned long File::hash(char *str) {
    unsigned long sum = 0, pack;
    int i, n = strlen(str); // examine least 4 bytes;
    for (pack = str[0]; i<n && isalnum(str[i]); i++) pack <<= 8;
    for (i = 0; i < sizeof(n); i++) {
```

Figure 10.16 continued

```
        for (pack = 0; i < 3; i++, k--) {
            pack = (unsigned char) s[i];      // include s[i] in the
                                              //(k)th pack
            if (i == 3 || i == elements - 1) { // byte of pack
                k--;
                break;
            }
            pack <<= 8;
        }                 // put at most 3 bytes from s
        for (; k >= 0; k--)  // make remaining bytes in pack
            pack <<= 8;      // zero
        return (pack & 0x7fffffff) % bucketsize; // nonnegative
    } // return hash value for s in one of bucketsize buckets

    void FileSystem::printBucket(int i) {
        cout << "Bucket number " << i << endl;
        char line[81];
        for (int k = 0; k < recordsize; k++) {
            line[k] = ' ';
        line[recordsize] = '\0';
    }

    void FileSystem::insert(char *name) {
        int address = hash(name), counter = 0;
        char nameInRecord[81];
        name[recordsize] = '\0';
        bool done = false, inserted = false;
        seekg(address*bucketsize,ios::beg);
        while (!done && getline(name,recordsize)) {
            if (name[0] == empty || name[0] == deleted) {
                seekg(address*bucketsize+counter*recordsize,ios::beg);
                cout << line << endl;
                done = inserted = true;
            }
            else if (strcmp(name,line) == 0) {
                cout << line << " is already in the file\n";
                return;
            }
            else counter++;
            if (counter == bucketsize)
```

FIGURE 10.16 *continued*

```
            data    temp;
        else cout << "Item " << item << " is already in the table\n";
    }
    if (!isDeleted) {
        done = false;
        counter = 2;
        cout << "Collision occurs\n";
        cout << "Linear probing:\n" << "Trying ";
        while (!done && counter <= tableSize - 1) {
            if (table[hashVal + counter * delta].info == NULL
                || table[hashVal + counter * delta].info == DELETED)
                done = true;
            else if (table[hashVal + counter * delta].info == item) {
                cout << "Item " << item << " is already in the table\n";
                return;
            }
            else counter++;
            cout << counter << ", ";
        }
        cout << endl;
        if (done) {
            table[hashVal + counter * delta].info = item;
            table[hashVal + counter * delta].count = 1;
        }
        else cout << "Table is full\n";
    }
}

void FileCount::remove(int item) {
    int hashVal = hash(item), counter = 0;
    bool done = false, removed = false;
    int delta = 1 + hash2(item) % 97;
    cout << "Collision occurs\n";
    cout << "Linear probing:\n";
    cout << "Trying ";
    cout << hashVal << ", ";
    while (!done && table[hashVal].info != NULL && counter <= tableSize - 1) {
        counter++;
        cout << hashVal + counter * delta << ", ";
        if (table[hashVal + counter * delta].info == item) {
            table[hashVal + counter * delta].info = DELETED;
            table[hashVal + counter * delta].count = 0;
            done = removed = true;
        }
        else done = true;
    }
```

Program 10.16 (continued)

```
        if (pointer == bucketSize)
            done = true;
        else outfile.seekg(pointer * recordSize, ios::beg);
    }
    if (!found) {
        done = false;
        numRecs = 0;
        overflowBucket();
        overflow.seekg(0, ios::beg);
        while (!done && overflow.get((char*)&next, sizeof(int))) {
            if (next == num) {
                overflow.clear();
                overflow.seekg(pointer * sizeof(int), ios::beg);
                overflow.put((char*)&lastPtr, sizeof(int));
                done = !(pointer == -1);
            }
            else pointer = next;
            overflow.seekg(pointer * recordSize, ios::beg);
        }
    }
    if (!found)
        cout << "Item " << num << " is not in database\n";
}

void HBucket::getAttributes(int key, int &num, float &price) {
    char rec[recordSize], *p = rec;
    int recordNum = key * recordSize;
    recStream.seekg(recordNum, ios::beg);
    recStream.read(rec, recordSize);
    memcpy(&num, p, sizeof(num)); p += sizeof(num);
    memcpy(&price, p, sizeof(price));
    cout << "Record " << key << " contains " << num << " " << price << endl;
    cout << "Enter new values: ";
    cin >> num >> price;
    recStream.seekg(recordNum, ios::beg);
    recStream.write(rec, recordSize);
}
```

Figure 10.16 (continued)

```
        previous = last.next;
    }

    void rehash(int low, int high) {
        int pivotItem;
        if (low < high) {
            pivotItem = table[low];
            hash(low, high, pivotItem);
            rehash(low, pivotItem - 1);
            rehash(pivotItem + 1, high);
        }
    }

    void writeSortedFile() {
        char recordFromFile[];
        RandomAccessFile pointers[];   // pointers[0] corresponds to file 0 of numbers;
        int recordNum = 0;             // put data from numbers file in sorted order
        for (int i = 0; i < pointers.length; i++) {       // in file sorted;
            pointers[i].seek(pointers[i].readInt());
            pointers[i].readFully(recordFromFile);
            sorted.write(recordFromFile);
        }
    }

    // Data from overflow file and table are written to sorted file and
    // prepared for external sort by reading positions of file data records in
    array;

    void writeRecords() {
        int counter = hashTable.length;
        char recordFromFile[];
        rec[recordNum] = '\0';
        overflow.seek(0);
        overflow.seek(0);
        while (overflow.getFilePointer() < overflow.length()) { // traverse file
            if (rec[0] != deletedKey) {          // overflow or table only
                recordNum++;                     // write (recordNum, item);
                sorted.write(rec, recordNum, recordFromFile);
            }
        }
        pointers = new RandomAccessFile[];
```

FIGURE 10.16 (continued)

```
    outfile.seekg(0,ios::beg);     // load in array positions available
    int number;                    // no valid data stored in current file;
    for(int i = 0; i < counter; i++) {
        outfile.seekg(positions[i],ios::beg);
        outfile.read(rec,recordsize);
        if (rec[0] != '*') rec[0] = delMarker;
            buckets[number] = i;
    }
    positions[0] = counter; // store the number of data in position 0;
}

void File::processAll(char *filename) {
    ifstream in(filename);
    if (!in.fail()) {
         cout << "cannot open " << filename << endl;
         return;
    }
    char command[maxLen];
    outfile.open(outfile,ios::in|ios::out|ios::binary);
    sorted.open("sorted",ios::in|ios::out|ios::binary);
    overflow.open("overflow",ios::in|ios::out|ios::binary);
    for (int i = 1; i <= MAXLINE/(sizeof(int)+recordsize); i++)
                                          // initialize
        outfile << empty;                 // overflow
    char line[recordsize];
    line[recordsize] = '\0';
    while (in.getline(line,recordsize)) // load infile to overflow
        insert(line);
    printf("overflow",overflow);
    printf("sorted",sorted);
    while (strcmp(command,"exit")) {
        cout << "Enter command (insert, remove, or exit): ";
        cin.getline(command,maxLen);
        if (!strcmp(command,"insert"))
             insert(line);
        else if (!strcmp(command,"remove"))
             remove(line);
        else if (strcmp(command,"exit"))
             cout << "Wrong command entered, please retry.\n";
    printf("outfile",output);
```

Program 10.16 continued.

```
    printf("overflow",overflow);
    }
    cout<<"\n";
    exit(1);
    printf("searched",searched);
    outfile.close();
    sorted.close();
    overflow.close();
    function();
                url = argv[filename];
                remove("sorted",fileName);
}

void main(int argc, char *argv[]) {
    char filename[30];
    if (argc != 2) {
        cout << "Enter a file name: ";
        cin.getline(name,30);
    }
    else strcpy(filename,argv[1]);
    ifile fc(name);
    fc.close(); remove(filename);
}
```

It has been found to be useful in the program to apply overly complicated hash functions. For example, the hash value corresponding to the string "ABCDEFGHIJ" is the concatenation "ABCD" "EFGH" "IJ" that is, an hexadecimal notation 0x41424344 0x45464748 0x0000494a. It may appear that the same value can be generated with the function

```
unsigned long hash(char *s) {
    unsigned long remainder;
    for (int i = 0; strlen(s); i += 4) {
        int r = *reinterpret_cast<unsigned long*>(s);
        if (strlen(s) != 0) {
            strcpy(reinterpret_cast<char*>(&remainder),s);
            r ^= remainder;
        }
        return (r % tableSize) * bucketSize * recordSize;
    }
```

The problem with this simpler function is that it may return different values for the same string. The outcome depends on the way numbers are stored on a particular system. In other words, it depends on the "endianness" employed by the system. For systems that is big endian, it stores the most significant byte in the lowest address, that is, numbers are stored "big end first." In a little endian system, the most significant byte is in the highest address. For example, number 0x12345678 is stored as 0x12345678 in the big endian system—first the contents of the highest order byte, 0x12, then the contents of the lower order byte, 0x34, and so on. On the other hand, the same number is stored as 0x78563412 in the little endian system—first the contents of the lowest order byte, 0x78, then the contents of the higher order byte, 0x56, and so on. Consequently, on the big endian systems, the statement

```
sum |= *(int *)prefix;  // assumes integers are 4 bytes long
```

works with even the values up to 0x20202020 (" ") but with values of 0x21212121 ("!!!!") or greater the system interprets the first four characters as a single integer value, rather than as four characters. Even with values smaller than 0x20202020, the order of the characters is reversed. On a little endian system, the same four characters are stored in reverse order, lowest order byte first. Therefore, to prevent this system dependence, we must first extract the characters individually, one at a time, after which the contents of each can be shifted to the left by eight bits to make room for another character. This is exactly what the bigEndianValue() function does.

10.7 Exercises

1. What is the maximum number of keys which can be stored in the hash table pictured in Figure 10.5? How many buckets are there? What is the load factor?

2. Consider the following hash algorithm (Bell and Kaney 1978). Let Q and C be the quotient and remainder, respectively, obtained by dividing the product of the ASCII values of the characters in the key by a prime number. The hash value is then computed by the following formula:

$$h(K) = \begin{cases} \text{[illegible]} & \text{[illegible]} \\ \text{[illegible]} & \text{[illegible]} \end{cases}$$

   What is the recommended value of TableSize? Explain how this algorithm works.

3. Is there any way to avoid rehashing when the table becomes full? Explain how this might be done.

4. Devise a hash function for converting strings to integers. Assume that all strings are made up of the printable characters in the ASCII collating sequence. Strings with a blank are to be ignored by the algorithm.

5. Trace the execution of the sorting algorithm pictured in Figure 10.4 on the following list: 5, 15, 37, 25, 79, 55, 75, 65, 42, 60, 58, 30, 89, 85, 45, 10.

6. Is the following statement legal? Explain what it does.

```
(*(void (*)())0)();
```

7. Apply the [illegible] algorithm to the array [illegible] with [illegible] and then with $c = 3$. What is the answer of the value of [illegible] operations if it is performed?

8. Formally specifying the hash function used in [illegible] hashing table by [illegible] changes in which cases? Why is that?

9. Consider an implementation of [illegible] hashing that allows [illegible] to point to only one [illegible]. Then [illegible] pointers so that all pointers in the [illegible] are connected. Call the [illegible] pointers. What keyword should [illegible]? What are the advantages and disadvantages of this implementation?

10. How would the directory used in extendible hashing be updated after splitting a [illegible] bucket if $b[k] = 3$ and the [illegible] bucket is the directory for the first and last buckets?

11. Explain the significance of [illegible] in extendible hashing and [illegible].

12. What is the purpose of [illegible] in providing [illegible] keys over [illegible] in order to hash up to the [illegible] of [illegible]?

13. Apply the [illegible] to [illegible] hash up to keys 5, 28, 19, 15, 20, 33, 12, and 54 in an initially empty table with [illegible] and $c = 1$. Then [illegible] the [illegible] and [illegible]. What problem can occur? Can this problem be [illegible] by [illegible] the table?

14. Outline an algorithm to [illegible] a key from a table using [illegible] hashing. What [illegible] must the insertion have?

15. The function [illegible] in the case study [illegible] the [illegible] operator [illegible] to find all the characters in [illegible]. Would [illegible] be to replace it by a [illegible] and [illegible]?

10.8 Programming Assignments

1. Analyze [illegible] in this chapter. Hashing techniques used [illegible] tables tend to [illegible] rapidly deteriorate [illegible] to [illegible] relatively small percentage of the table is [illegible]. This problem can be solved using [illegible] to [illegible] [illegible] by [illegible] better hash functions. [illegible] for [illegible]. Write a program to [illegible] the [illegible] of various [illegible] with the [illegible] hashing method. Have your program write a table similar to the one in Figure 9.14, which gives the averages for successful and unsuccessful [illegible] in the table. Use functions for operations on strings so that [illegible] can be changed [illegible]. Here are some examples that [illegible] should be displayed [illegible]:

   a. First attempt = Second attempt [illegible]

   b. Find [illegible] = Last character + length of [illegible]

   c. Function [illegible] = [illegible]
      [illegible] = [illegible] Remainder of [illegible]

2. Another way of improving the performance of hashing is to allow reorganization of the hash table during insertions. Write a program that compares the performance of linear probing with the following reorganization hashing methods:
   a. Last-come-first-served hashing (described briefly in the section on open addressing). The element that is stored in the last position examined in a search operation is moved to the next unoccupied position (Vitter 1980).
   b. Robin Hood hashing (Celis et al. 1985). The number of positions two colliding keys are away from their home positions are counted, and the search continues for an open position for the key closer to its home position (Celis et al. 1985).
3. Write a program that inserts records in a file and retrieves and deletes them using virtual extendible hashing or the linear hashing technique.
4. Extend the program presented in the case study by creating a linked list of overflow buckets associated with each bucket of the intermediate file structure. Note that if a bucket has no empty slots the search continues in the overflow area. In the extreme case, it may mean that the bucket holds only deleted items and new items are inserted in the overflow area. Therefore, it may be advantageous to have a purging function that, after a certain number of deletions, is automatically invoked. This function transfers items from the overflow area to the main file which are hashed to buckets with deleted items. Write such a function.

Bibliography

Bell, James R., and Kenneth C. Kapeller. "The Linear Quotient Hash Code." Communications of the ACM 13, no. 11 (November 1970): 675–677.

Carter, J. Lawrence, and Mark N. Wegman. "Universal Classes of Hash Functions." Proceedings of the Ninth ACM Symposium on Theory of Computing, Boulder, Colorado, 1977, 106–112.

Cichelli, Richard J. "Minimal Perfect Hash Functions Made Simple." Communications of the ACM 23, no. 1 (January 1980): 17–19.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computer Science Technical Report Series, CS-TR-1537, University of Maryland, College Park, Maryland, August 1985.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computing Surveys 20, no. 3 (September 1988): 173–189.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computer Science Technical Report Series, CS-TR-1537, University of Maryland, College Park, Maryland, August 1985.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computing Surveys 20, no. 3 (September 1988): 173–189.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computer Science Technical Report Series, CS-TR-1537, University of Maryland, College Park, Maryland, August 1985.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computing Surveys 20, no. 3 (September 1988): 173–189.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computer Science Technical Report Series, CS-TR-1537, University of Maryland, College Park, Maryland, August 1985.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computing Surveys 20, no. 3 (September 1988): 173–189.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computer Science Technical Report Series, CS-TR-1537, University of Maryland, College Park, Maryland, August 1985.

Cichelli, Richard J., and Thomas O. Winbigler. "Minimal Perfect Hash Functions Revisited." Computing Surveys 20, no. 3 (September 1988): 173–189.

Larson, Per-Å., "Dynamic Hash Tables," Communications of the ACM 31 (1988), 446–457.

Larson, Per-Å., "Dynamic Hash Tables," Communications of the ACM 31 (1988), 446–457.

Lewis, Ted G., and Cook, C. R., "Hashing for Dynamic and Static Internal Tables," IEEE Computer (October 1988), 45–56.

Litwin, Witold, "Virtual Hashing," Proceedings of the Fourth International Conference on Very Large Databases (1978), 212–223.

Litwin, Witold, "Linear Hashing: A New Tool for File and Table Addressing," Proceedings of the Sixth International Conference on Very Large Databases (1980), 212–223.

Lomet, David B., "Bounded Index Exponential Hashing," ACM Transactions on Database Systems 8 (1983), 136–165.

Lum, V. Y., Yuen, P. S. T., and Dodd, M., "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," Communications of the ACM 14 (1971), 228–239.

Morris, Robert, "Scatter Storage Techniques," Communications of the ACM 11 (1968), 38–44.

Nishihara, S., and Ikeda, H., "Hash Tables in Distributed Processing," IEEE Transactions on Software Engineering SE-11 (1985), 234–242.

Ogata, W. Y., and Stahl, F. A., "Redundancy and Performance of Parallel Hashing Algorithms," Proceedings of the 1981 International Conference on Parallel Processing (1981), 188–190.

Rabin, Michael O., "Probabilistic Algorithms," in Traub, J. F., ed., Algorithms and Complexity: New Directions and Recent Results, Academic Press, New York (1976), 21–39.

Radke, Charles E., "The Use of Quadratic Residue Research," Communications of the ACM 13 (1970), 103–105.

Sager, Thomas J., "A Polynomial Time Generator for Minimal Perfect Hash Functions," Communications of the ACM 28 (1985), 523–532.

Severance, Dennis G., and Lohman, Guy M., "Differential Files: Their Application to the Maintenance of Large Databases," ACM Transactions on Database Systems 1 (1976), 256–267.

Tharp, Alan L., File Organization and Processing, New York: Wiley, 1988.

Vitter, Jeffrey Scott, and Chen, Wen-Chin, Design and Analysis of Coalesced Hashing, New York: Oxford University Press, 1987.

Data Compression

Transfer of information is essential for the proper functioning of any structure on any level and any type of organization. The faster an exchange of information occurs, the smoother the structure functions. Improvement of the rate of transfer can be achieved by improving the medium through which data are transferred or by changing the data themselves so that the same information can be transmitted within a shorter time interval.

Information can be represented in a form which exhibits some redundancy. For example, in a database, it is enough to say about a person that he is "M" or else "F," instead of spelling out the whole words, "male" and "female," or to use 1 and 2 to represent the same information. The number one hundred twenty-eight can be stored as 80 (hexadecimal), 128, 200000 (binary), CXXVIII, ρκη (in the Greek language used letters as digits for numbers), or 128 (as text). If numbers are stored as the sequence of digits representing them, then 80 is the shortest form. Numbers are represented in binary form in computers.

11.1 Conditions for Data Compression

When transferring information, the choice of the data representation determines how fast the transfer is performed. A judicious choice can improve the throughput of a transmission channel without changing the channel itself. There are many different methods of data compression (or encoding) that reduce the size of the representation without affecting the information itself.

Assume that there are s different symbols used to code messages. For a binary code, $s = 2$, for Morse code, $s = 3$ (the dot, the dash, and the mark separating the sy-

quences of dots and dashes that represent letters. Assume also that all symbols m_i in the incoming set M have been independently chosen and are known to have probabilities of occurrence $P(m_i)$, and the symbols are coded with strings of 0s and 1s. Then $P(m_1) + \cdots + P(m_n) = 1$. The information content of the set M, called the *entropy* of the source M, is defined by

$$I_{avg} = P(m_1)b(m_1) + \cdots + P(m_n)b(m_n) \tag{11.1}$$

where $b(m_i) = -\lg P(m_i)$, which is the minimum length of a codeword for symbol m_i. Claude E. Shannon established in 1948 that formula 11.1 gives the best possible average length of a codeword when the set of symbols and the probabilities of their use are known. No data compression algorithm can be better than I_{avg}, and the closer it is to this number, the better is its compression rate.

For example, if there are three symbols m_1, m_2, and m_3 with the probabilities .25, .75, and .5, respectively, then the lengths of the codewords assigned to them are:

$$-\lg(P(m_1)) = -\lg(.25) = \lg\left(\frac{1}{.25}\right) = \lg(4) = 2 \text{ bits}$$

$$-\lg(P(m_2)) = \lg(1.33\ldots) = .415$$

and the average length of a codeword is

$$I_{avg} = P(m_1) \cdot 2 + P(m_2) \cdot .415 + P(m_3) \cdot 1 = 1.3$$

We should now be in a position to begin to develop algorithms that minimize the average codeword length by devising a method that uses as few bits as possible to represent the symbols that appear with the most frequency. With which symbol is one concerned? If a symbol m_i is used infrequently, $P(m_i)$ can be assumed to be small, and $b(m_i)$ can be assigned a large number of bits. Frequently used symbols will appear near the root of the tree.

Some general principles must be observed for the prefix property codes:

1. Each codeword must uniquely represent its corresponding symbol.
2. Decoding should not require any look-ahead; after reading each symbol it should be possible to determine whether the end of a string encoding a symbol of the original message has been reached. A code meeting this requirement is called a code with the prefix property and it means that no codeword is a prefix of another codeword. Therefore, no special punctuation is required to separate two codewords in a coded message.

The second requirement can be illustrated by three different encodings of three symbols as given in the following table.

Symbol	Code$_1$	Code$_2$	Code$_3$
A	1	1	11
B	0	01	10
C	10	10	00

The first code does not allow us to make a distinction between AB and C—both are coded as 10. The second code does not have this ambiguity, but it looks ahead, as the 1222 The first 1 can be decoded as A. The following 0 that A was appropriately chosen and B should have been decoded

A is a proper prefix of the third symbol, namely 1. Similarly, A is a proper prefix of the fourth symbol, but the first symbol is another 1. Hence, the first term was wrong—as Alex knew all along. The proper short message is 1 B. All these problems arise because both code 1 and code 1 violate the prefix property. By symmetry, any other ambiguously decoded message.

For an optimal code, two more requirements are specified next.

3. The length of the codeword for a given symbol s_i should not exceed the length of the codeword of a less probable symbol s_j; that is, if $p(s_i) < p(s_j)$, then $l(s_i) \geq l(s_j)$ for $1 \leq i, j \leq n$.
4. In an optimal encoding system, there should not be any unused short codewords either as stand-alone encodings or as prefixes for longer codewords since this would mean that longer codewords were created unnecessarily. For example, the sequence of codewords 111, 1000, 1001, 101 for a certain set of five symbols is not optimal because the codeword 110 is not used anywhere; this encoding can be turned into an optimal sequence 01, 10, 11, 000, 001.

In the following sections, several data compression methods are presented. To compare the efficiency of these methods when applied to the same data, the same measure is used. This measure is the compression ratio (also called the *fraction of data reduction*), and it is defined as the ratio

$$\frac{\text{length(input)} - \text{length(output)}}{\text{length(input)}} \tag{11.2}$$

It is expressed as a percentage and tells by the amount of redundancy removed from the input.

11.2 Huffman Coding

The construction of an optimal code was developed by David A. Huffman, who utilized a tree structure to be constructed from a binary tree for a binary code. The algorithm is surprisingly simple and can be summarized as follows:

```
Huffman();
    for each symbol in the alphabet assign a weight and add to the list
        according to the probability of symbol occurrence
    while the weight of the root is less than 1
        select the two trees with the lowest probabilities p1 and p2 (p1 <= p2)
            and create a tree with p1 and p2 as its children and with
            the probability of the new tree equal to p1 + p2
        associate 0 with each left branch and 1 with each right branch
    create an unique codeword for each symbol by traversing the tree from the root
        to the leaf containing the probability corresponding to the
        symbol and by putting all encountered 0's and 1's together
```

The resulting tree is a prefix code. Huffman's algorithm

It should be noted that the algorithm is not deterministic in the sense of producing a unique tree because, for trees with equal probabilities at the roots, the algorithm does not prescribe their positions with respect to each other either at the beginning of during execution. If r_3 with probability p_3 is in the sequence of trees and the new tree r_4 is created with $p_4 = p_1$, should r_4 be positioned to the left of r_3 or to the right? Also, if there are three trees r_1, r_2, and r_3 with the same lowest probability in the ordered sequence, which two trees should be chosen to create a new tree? There are three possibilities for choosing two trees. As a result, different trees can be obtained depending on where the trees with equal probabilities are placed in the sequence with respect to each other. Regardless of the shape of the tree, the average length of the codewords remains the same.

To assess the compressing efficiency of the Huffman algorithm, a definition of the weighted path length is used, which is the same as Equation 11.1 except that $L(r_i)$ is interpreted as the number of 0s and 1s in the codeword assigned to symbol a_i by this algorithm.

Figure 11.3 contains an example for the five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09, respectively. The tree in Figures 11.3a and 11.3b are different in the way in which the two nodes containing a probability of .21 have been chosen to be combined with tree .19 to create a tree of .40. Regardless of the choice, the lengths of the codewords associated with the five letters A through E are the same, namely, 2, 2, 2, 3, and 3, respectively. However, the codewords assigned to them are slightly different, as shown in Figures 11.3c and 11.3d, which present alternative (and more commonly used) versions of the way the trees in Figures 11.3a and 11.3b were created. The average length for the latter two trees is

$$L_{\text{tree}} = .39 \cdot 2 + .21 \cdot 2 + .19 \cdot 2 + .12 \cdot 3 + .09 \cdot 3 = 2.2,$$

which is very close to 2.09 (only 5% off), the average length computed according to Equation 11.2:

$$L_{\text{ave}} = .39 \cdot 1.238 + .21 \cdot 2.252 + .19 \cdot 2.396 + .12 \cdot 3.059 + .09 \cdot 3.474 = 2.09$$

Corresponding letters in Figures 11.3c and 11.3d have been assigned codewords of the same length. Obviously, the average length for both trees is the same. But each way of building a Huffman tree, starting from the same data, should result in the same average length, regardless of the shape of the tree. Figure 11.2 shows two Huffman trees for the letters P, Q, R, S, and T with the probabilities .1, .1, .1, .2, and .5, respectively. Depending on how the lowest probabilities are chosen, different codewords are assigned to these letters with different lengths, at least for some of them. However, the average length remains the same and is equal to 2.0.

The Huffman algorithm can be implemented in a variety of ways, at least as many as the number of ways a priority queue can be implemented. The priority queue is the natural data structure in the context of the Huffman algorithm since it requires removing the two smallest probabilities and inserting the new probability in the proper position.

One way to implement this algorithm is to use a singly linked list of pointers to trees, which reflects closely what Figure 11.3a illustrates. The linked list is initially ordered according to the probabilities stored in the trees, all of them consisting of just a

FIGURE 11.3 Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09.

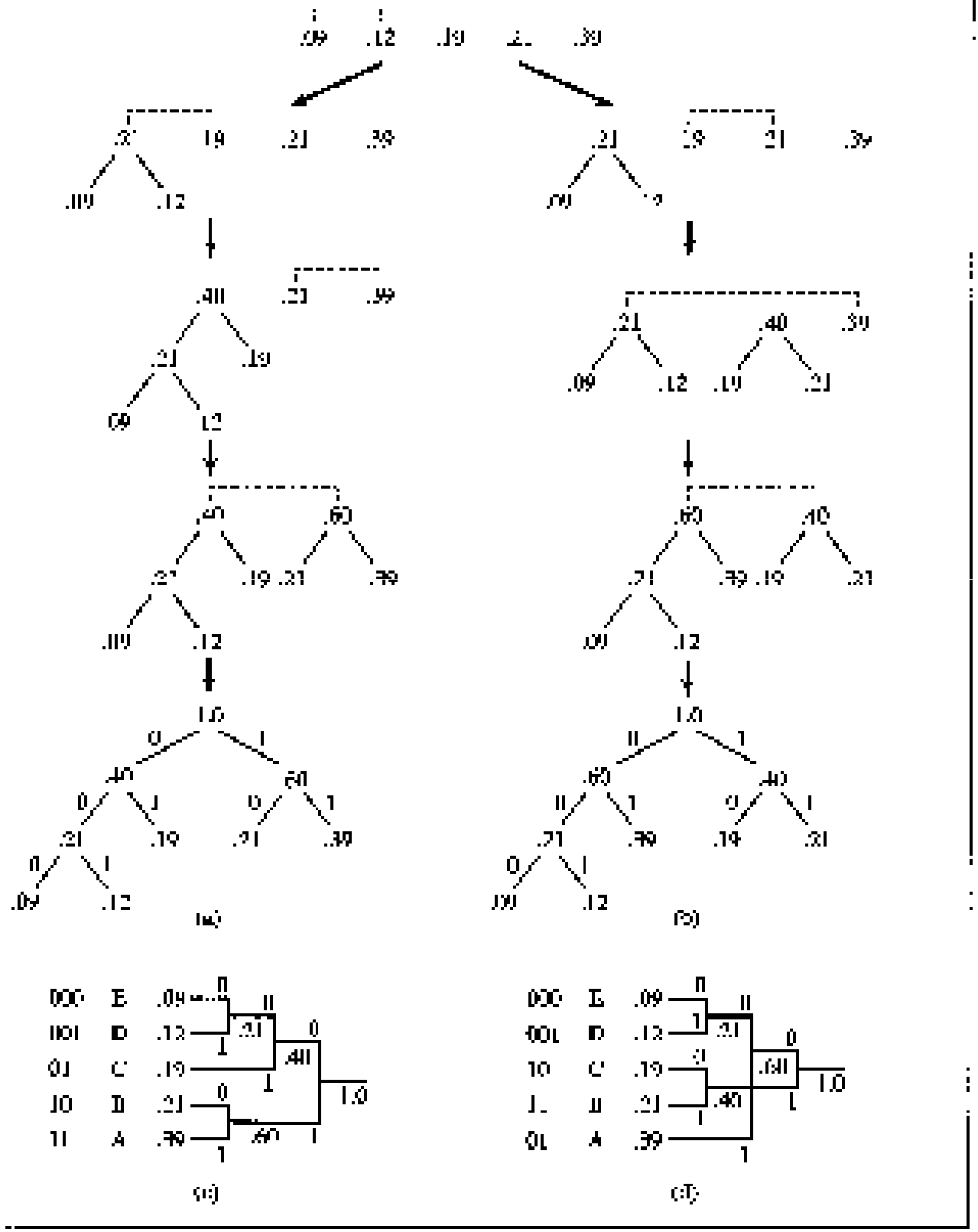
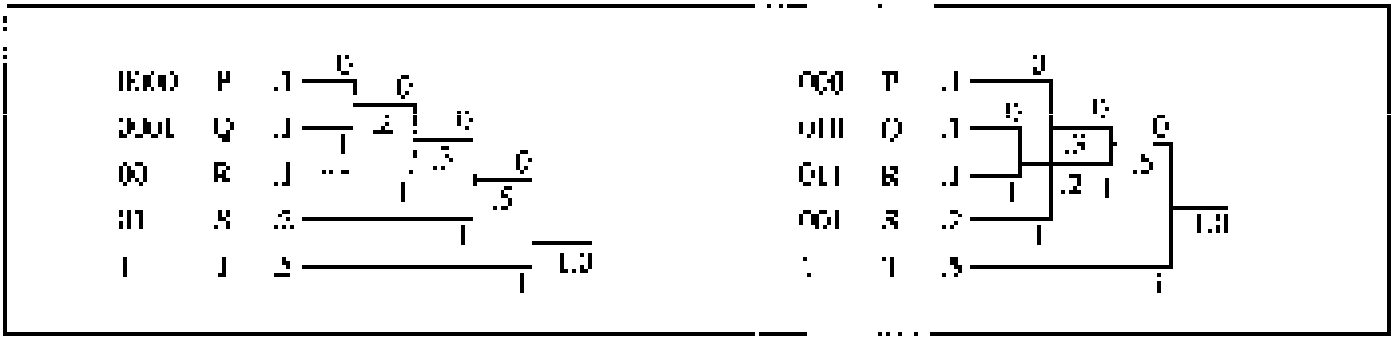


FIGURE 11.2 Two Huffman trees generated for letters R, Q, K, S, and T with probabilities .1, .1, .1, .2, and .5.



root. Then, repeatedly, the two trees with the smallest probabilities are chosen, the tree with the smaller probability is replaced by a newly created tree, and the node with the pointer to the tree with the higher probability is removed from the linked list. From trees having the same probability in the roots, the first tree encountered is chosen.

In another implementation, all probability nodes are first ordered, and that order is maintained throughout the invocation. From such an ordered list, the first two trees are always removed to create a new tree from them, which is inserted close to the end of the list. To that end, a doubly linked list of pointers to trees with immediate access to the beginning and to the end of that list can be used. Figure 11.3 contains a trace of the execution of this algorithm for the letters A, B, C, D, and E with the same probabilities as in Figure 11.1. Codewords assigned to those letters are also indicated in Figure 11.3. Note that they are different from the codewords in Figure 11.1, although their lengths are the same.

The two preceding algorithms build Huffman trees bottom-up by starting with a sequence of trees and collapsing them, one by one, into a gradually smaller number of trees and, eventually, to one tree. However, this tree can be built top-down, starting from the highest probability. But only the probabilities to be placed in the leaves are known. The highest probability, to be put in the root, is known if lower probabilities, in the root's children, have been determined, the latter are known if still lower probabilities have been computed and so on. Therefore, creating nondetermined nodes has to be deferred until the probabilities to be stored in them are found. It is very convenient to use the following recursive algorithm to implement a Huffman tree:

```
createLeafOfHuffmanTree(probs)
    declare the probabilities p1, p2, and the Huffman tree HTree;
    if only two probabilities are left in probs
        HTree = a tree with p1, p2 in the leaves and p1 + p2 in the root;
    else remove the two smallest probabilities from probs and assign them to p1 and p2;
        insert p1 + p2 in probs;
        HTree = createHuffmanTree(probs);
        in HTree make the leaf with p1 + p2 the parent of two leaves with p1 and p2;
    return HTree;
```

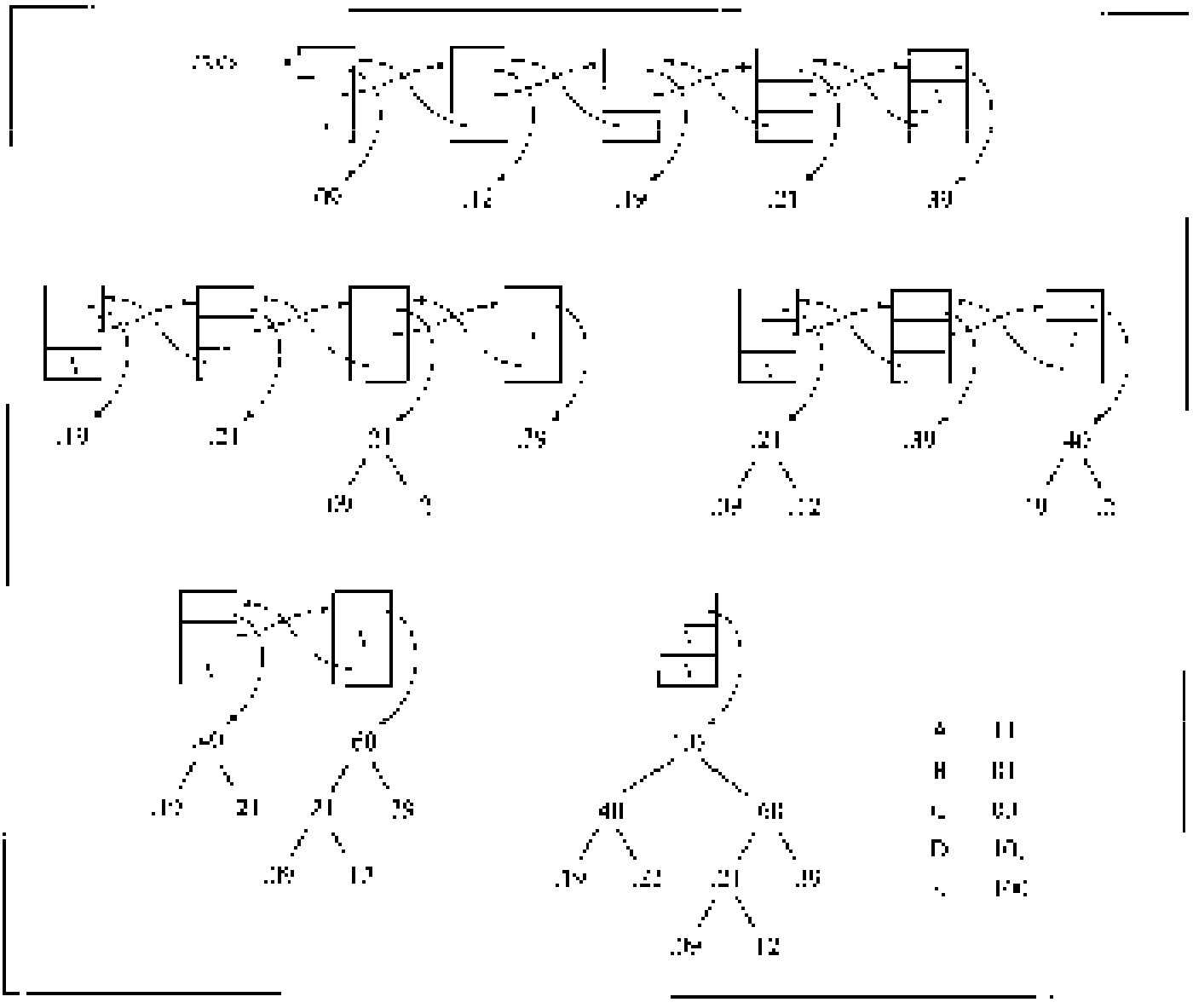
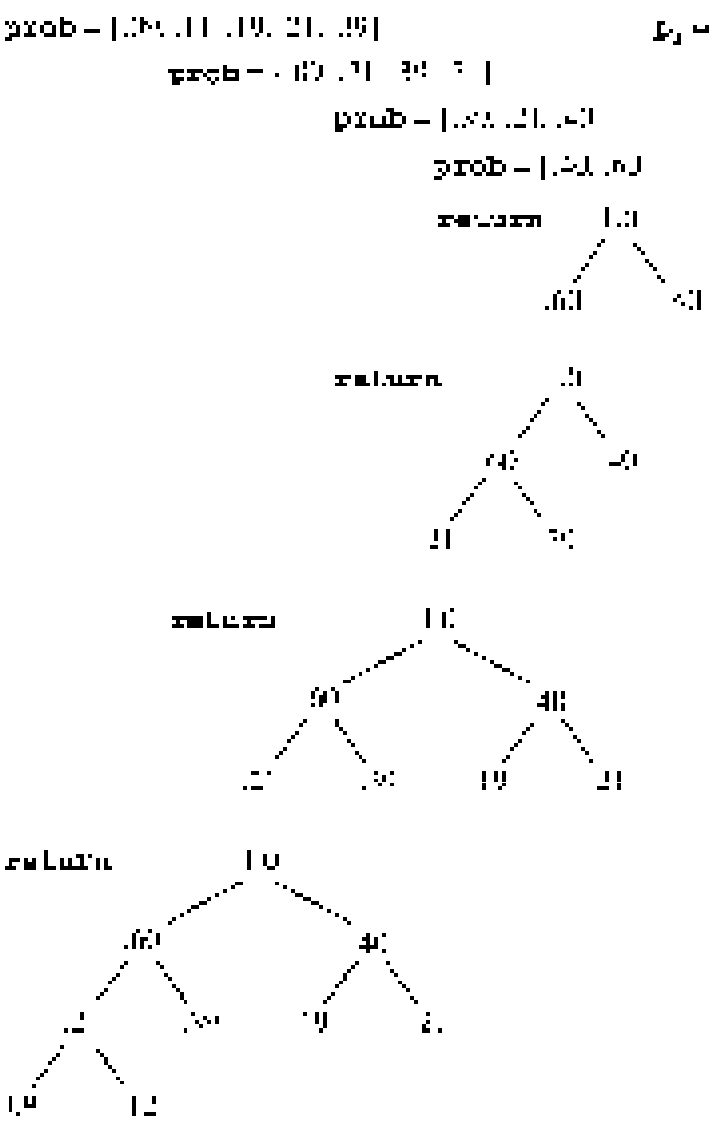


Figure 11.3 contains a summary of five of the execution of Jo's algorithm for the letters A, B, C, D, and E of the probabilities shown in Figure 11.1. Ordinarily indicates consecutive steps in the execution of the algorithm.

One implementation of a priority queue is a heap, which can also be used to implement this algorithm. In this application, each node has a smaller probability than the probabilities of its children, and hence the smallest probability is in the root. That node is simple to remove. But after it is removed, the root is empty. Therefore, the largest element in the heap is moved to the root and the heap property is restored. Then the second-smallest node can be removed from the root and replaced with a new node whose weight is the sum of the probabilities of the root and the probability previously removed.

Figure 11.4 Top-down construction of a Huffman tree using recursive implementation.



Afterward, the heap property has to be restored again. After one or several executions, the heap has one less node. Two probabilities from the previous heap have been removed and a new one has been added. But it is not enough to create the Huffman tree. The new probability is a parent of the probabilities just removed, and this information must be retained. To that end, three arrays can be used: indexes containing the indexes of the original probabilities and the probabilities created during the process of creating the Huffman tree; parents, an array of the original and newly created probabilities; and pointers, an array of indexes indicating the position of the parents of the elements stored in probabilities. A positive number in pointers indicates the left child, and a negative number indicates the right child. Codewords are created

be accumulating and is when going from leaves to the root, using the array pointers, which functions as an array of pointers. It is important to note that in this particular implementation probabilities are stored and used. The heap is actually made up of indexes to probabilities, and exchanges take place in indexes.

Figure 11.7 illustrates an example of using a heap in the Huffman algorithm. The heap is a tree structure in which each node is smaller than its children. Thus, the highest probability is put in the root node, as in step (a) of Figure 11.7. Next, the heap is restored, and the two smallest, $p_{(g)}$ and $p_{(k)}$, and the next probability, $p_{(m)}$, are in the nodes of the next two largest probabilities, as in steps (d), (e), (f), and (g). Finally, in step (h), the heap is complete when there is only one node in the heap.

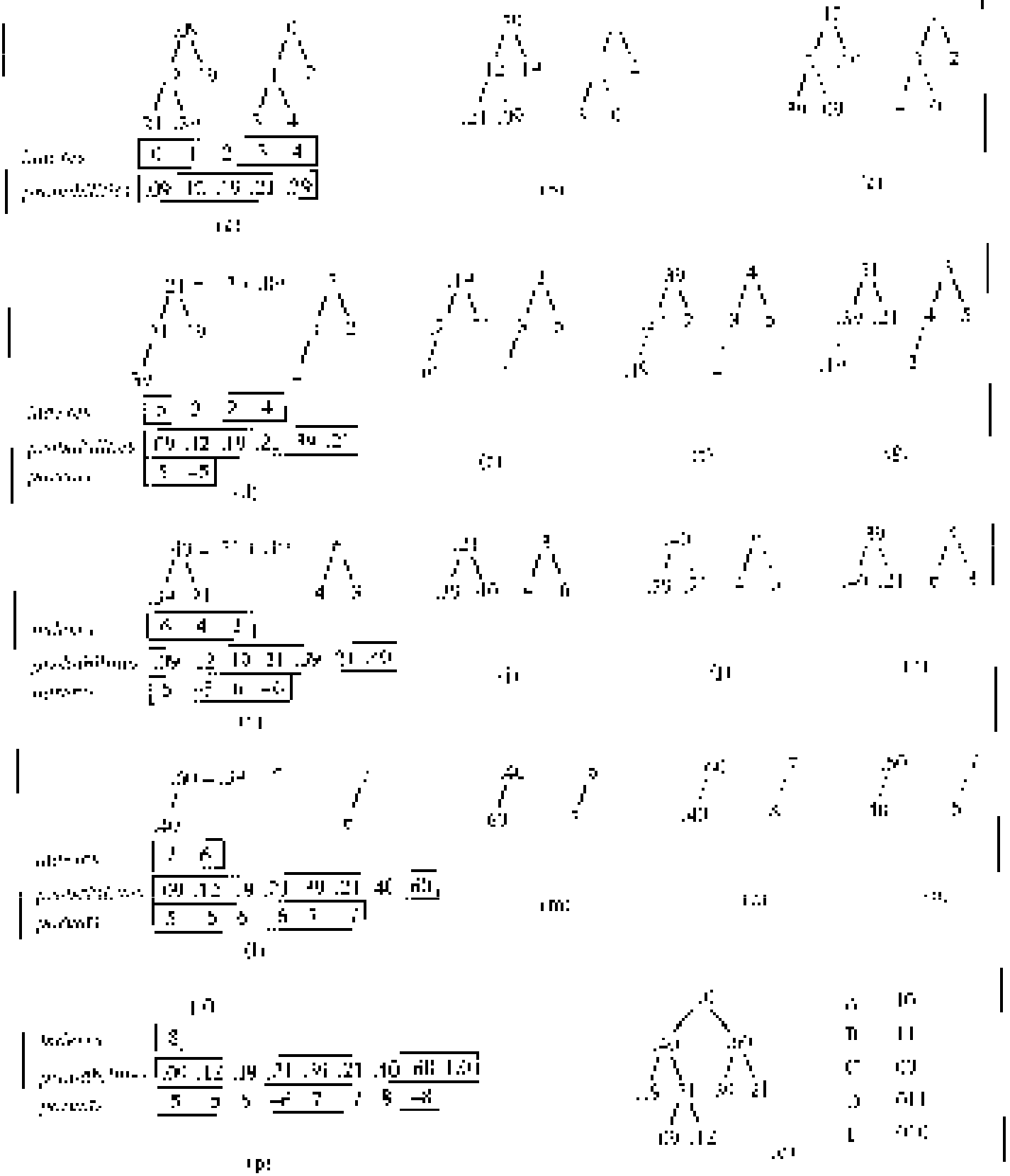
Using the Huffman tree, a table can be constructed which gives the equivalents for each symbol in terms of bits and the number of bits to be stored. Using the path leading to each of the leaves of the tree, bits are accumulated. The rules in Figure 11.6 will be used, and the resulting table is

A	1
B	01
C	0000
D	001
E	0001

The coding process is to encode each of the symbols to be stored into its equivalent. Instead of encoding ABACAD, the code is 1 01 1 0000 1 001. This is displayed with the spaces included for clarity. The original message is 6 characters long, or 48 bits. The coded value is 11 bits long. Thus, the compression ratio is 11/48, or about 23%. To decode this message, the tree is used. Using the whole Huffman tree can be constructed with the same probabilities that were used for coding, but instead of the probabilities, the symbols are stored in the nodes. Starting at the root, and moving to the left for 0 and to the right for 1, leads to a leaf node. The symbol stored in the leaf node is the symbol to be decoded. Next, the process starts over at the root, and the sequence 001 is decoded as B. The original message is decoded as BAD.

At this point, a question can be asked: Why would 1 01 1 0000 1 001 be stored instead of 1010000 1001? This is supposed to be a compression algorithm. Isn't the coded message longer than the original? Well, it is in this case. However, in which messages are each A, B, C, D, and E, single letters and not a sequence of characters, typically eight bits long? Instead, they could be represented as 3-bit values. Then, the original message ABACAD would require 18 bits. The coded message, 11 bits long, is still smaller. Even if the coded message is longer than the original, it is still possible that the message is not stored. Instead, a flag is set to indicate that the message is stored in its original form. ABACAD

FIGURE 11-5 Huffman algorithm—presented with a loop.



This example raises one problem: Both the encoder and the decoder have to use the same coding tree. The same Huffman tree. Otherwise, the decoding will be unsuccessful. How can the encoder let the decoder know which particular Huffman tree has been used? There are at least three possibilities:

1. Both the encoder and decoder agree beforehand on a particular Huffman tree and both use it for sending any message.
2. The encoder prepends to the Huffman-encoded message a short header describing the tree to be used (the message can be scanned in advance for this purpose). Both encoder and decoder have access to a collection of trees and the header specifies which tree to use.
3. The decoder constructs the Huffman tree during transmission and decoding.

The second strategy is more versatile, but its advantages are available only when large files are encoded and decoded. For an example, assume that the table of codewords and the coded message 10011111101 is being perceived as data compression. However, this file contains a message of 10,000 characters using the characters A through F, with spaces and punctuation. Using the probabilities mentioned earlier for these letters, we project that there are approximately 3000 As, 2100 Bs, 1500 Cs, 2000 Ds, and 900 Es. Hence, the number of bits needed to store this file is

$$3000 \cdot 2 + 2100 \cdot 2 + 1500 \cdot 2 + 1200 \cdot 3 + 900 \cdot 3 = 22{,}100 \text{ bits} = 2762.5 \text{ bytes},$$

which is approximately one fourth the 10,000 bytes required for sending the original file. Even if the code table is added to the file, this proportion is only minimally affected.

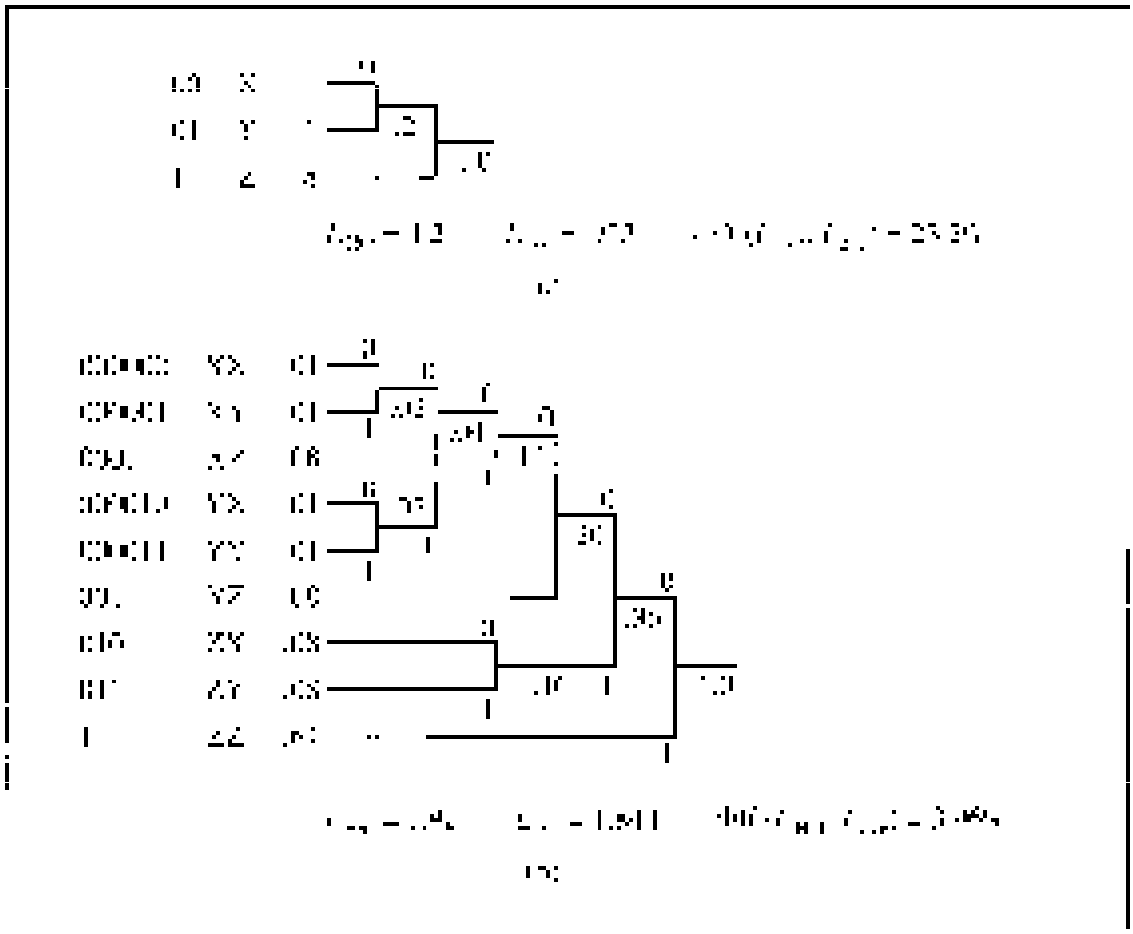
However, even with this approach, there may be some room for improvement. As indicated, an ideal compression algorithm should give the same average codeword length as computed from Equation 11.1. The symbols from Figure 11.1 have been assigned codewords whose average length is approximately 2.8 bits, which is the ideal 2.65. Sometimes, however, the difference is larger. Consider, for example, three symbols X, Y, and Z with probabilities 0.1, 0.2, and 0.7. Figure 11.5a shows a Huffman tree for these symbols, with codewords assigned to them. The average length according to this tree is

$$l_{\text{Huff}} = 2 \cdot 0.1 + 2 \cdot 0.2 + 1 \cdot 0.7 = 1.3$$

and the best expected average, l_{opt}, is 1.157. Therefore, there is a possibility we can improve the Huffman coding by approximately 0.143 bits using some fancy, clever, and complicated algorithm. However, such improvement is not possible because the average is below 1. How is this possible? As already stated, all Huffman trees reach the same average weighted path length. Therefore, no improvement can be expected if only the symbols X, Y, and Z are coded in the usual manner.

On the other hand, if all possible pairs of symbols are used for building a Huffman tree, the data rate can be reduced. Figure 11.5b illustrates such a procedure. Out of three symbols X, Y, and Z, nine pairs are created whose probabilities are computed by multiplying the original probabilities of both symbols. For example, since the probability for both X and Y is 0.1, the probability of pair XY is 0.01. The average l_{HL} is 1.92

FIGURE 11.6 Improving the average length of the codewords by applying the Huffman algorithm to pairs of letters instead of single letters



and the expected average L_2 is 1.84 for the previous P_{avg}, with the difference between these averages being 0.01. This represents a 10.2% improvement at the cost of including a larger conversion table (nine entries instead of seven) as part of the message to be sent. If the message is large and the number of symbols used in the message is relatively small, then the increase in the size of the table is insignificant. However, for a large number of symbols, the size of the table may be much too large to notice any improvement. For 26 large alphabetic characters, the number of pairs is 676, which is considered relatively small. But if 95 printable characters have to be distinguished in an English text (from the blank character, ASCII code 32, to the tilde, ASCII code 126, plus the carriage return character, then there are $95^2 = 9025$ possible pairs and 9025 codewords of the pairs. Many of these pairs are not likely to occur at all (e.g., XQ or ZJ), but even if only a few of them are found, the resulting table containing these pairs along with codewords associated with them may be too large to be useful.

Using pairs of symbols is a good idea even if the number of symbols is large. For example, a Huffman code can be constructed for all 256 symbols and for all pairs of symbols that occur in a file. Even more than 256 pairs of characters will rarely appear, and the encoding can be processed by combining two symbols at a time. Experimental experience

performed on an English text. Pixel progressive and adaptive Huffman image compression were also used. The compression ratios obtained were approximately 43%, 53%, and 58%, respectively. When single characters were used as the alphabet, the Huffman approach produced poor results (little or no compression), primarily because there are 256 characters in the ASCII code, and even a large book rarely uses more than 100 of them frequently. When pairs of characters were used, the compression ratios improved (approximately 27%, 45%, and 51%, respectively), but the number of different pairs was very large, and the storage requirements of the Huffman tree became impractical.

11.2.1 Adaptive Huffman Coding

The Huffman algorithm requires prior knowledge of the probabilities of the source sequence. One way to obtain this knowledge is to have both the encoder and the decoder count the occurrences of symbols (bytes) as they are transmitted or received. In a two-pass scheme, the encoder counts the occurrences in the first pass and then constructs the Huffman tree. In the second pass, it encodes the symbols. To maintain synchronization, the decoder also counts the symbol occurrences and constructs the Huffman tree after decoding each symbol. For each symbol, the decoder must therefore repeat some of the work performed by the encoder. The two-pass scheme is impractical for files of any size because both encoder and decoder must store the entire Huffman tree. In addition, the first pass does not do any compression at all. For these reasons, the two-pass scheme is rarely used. Instead, a one-pass method called *adaptive* Huffman coding is employed. This method is also known as *dynamic* Huffman coding. Adaptive Huffman coding was first suggested by Faller [Faller 73] and Gallager [Gallager 78] and then improved by Knuth [Knuth 85] and Vitter [Vitter 87]. The resulting algorithms are referred to as FGK and V (after their developers), respectively. The following description is based on [Vitter 87].

In adaptive Huffman coding, the encoder computes the Huffman tree based on the statistics of the symbols already encoded and the decoder keeps track of these statistics as well. Both encoder and decoder start with the same initial Huffman tree, which is modified as more symbols are processed. It is expected that as more and more symbols are read and encoded, the Huffman tree will approach more and more closely the optimum tree, since it will better reflect the statistics for the data. Note that the decoder does not need to be told explicitly about any changes in the tree because it can duplicate the encoder's operations and make the same changes in its own tree.

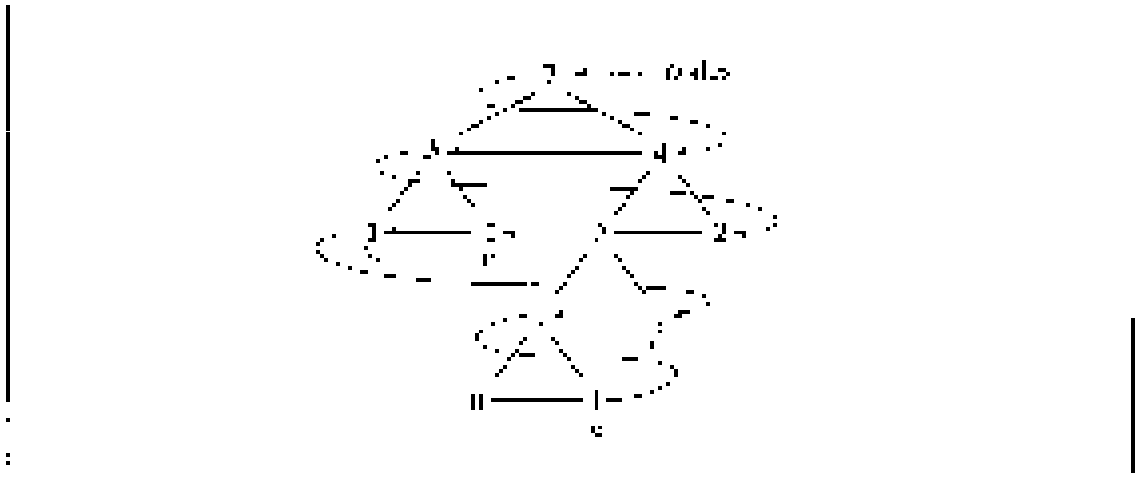
Initially, the Huffman tree consists of a single node that contains no information about the symbols and has a zero count. Symbols that are read for the first time are written to the output stream in uncompressed form. The encoder then adds a node for the symbol to the tree and sets its count to one. The next time this symbol is encountered, its count is incremented by one. As counts are increased, the relative ordering of the nodes may need to be changed (see below) to preserve the Huffman tree property. The decoder mirrors the encoder's operations, maintaining an identical Huffman tree.

The only subtle point in this method is how the encoder distinguishes between uncompressed codes and regular codes, and how the decoder knows when to read a code or a raw symbol. To make this easy, the encoder adds a special node to the tree, called the 0-node, whose purpose is to indicate any uncompressed symbol. The 0-node is never assigned any symbol, and its count is always kept at zero. A glance at the Huffman tree at any time during the encoding or decoding process will identify the 0-node as the node with zero count. Whenever the 0-node is output, the encoder knows that it must write an uncompressed symbol on the output stream. The decoder knows that it must read the next symbol in uncompressed form.

Assigning codes in adaptive Huffman coding is slightly different from the way codes are assigned in the static case. The rule is that nodes are listed in order of increasing weight, with leaves preceding internal nodes of the same weight. Figure 11.5a shows a possible Huffman tree for the string "aardvark" after the first seven characters have been input and encoded. The tree is valid since the weights are in increasing order. The next character is "r", so the second node on the list is incremented, changing the order of nodes. The result is shown in Figure 11.5b. Finally, Figure 11.5c shows the tree after the last node on the list is incremented.

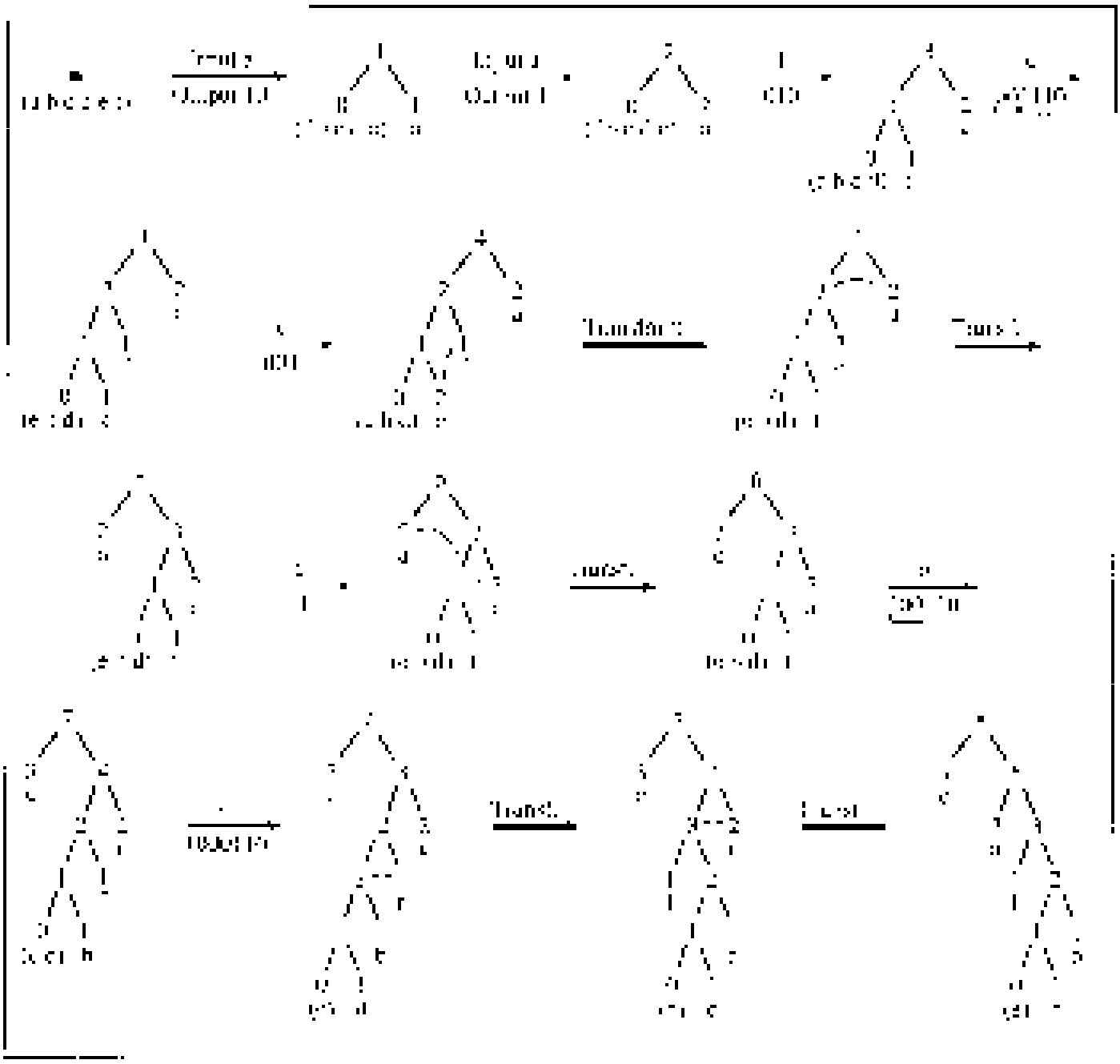
The encoder and decoder must also agree on how to update the tree. This is done in steps, with one step executed for each symbol input and encoded. Each step starts with the two nodes being checked to see if they are in the right order. If they are, the tree is not changed. If they are not in the right order, they are swapped, and the tree is checked again. Figure 11.6 shows the tree after the second "r" has been read from the input stream. The nodes have been swapped, and the tree is now in order. The next step checks the parent of the two nodes just swapped. Figure 11.7 shows the tree after the second "r" has been encoded. The next node on the list is the 0-node, so the encoder outputs the code of this node (0) followed by the uncompressed code of the symbol "r" (see Figure 11.7 for the codes used). The tree is updated in the same way as in the previous step. Figure 11.8 shows the tree after the "d" has been encoded. Figure 11.9 shows the tree after the "v" has been encoded. Figure 11.10 shows the tree after the "a" has been encoded. The code for "a" is now 10, a short code, because "a" is a frequent character. Figure 11.11 shows the tree after the "k" has been encoded. The code for "k" is 1111, the longest code in this tree.

Figure 11.7 Doubly linked list nodes created by Lempel-Ziv-Welch decoding algorithm.



All nodes also contain a pointer to the node in the tree that represents the parent of the node's symbol. The root node is the parent of itself. The algorithm for decoding a codeword begins at the root node. The first bit of the codeword is used to select either the left or right child of the root node. If the bit is a 0, the left child is selected; if the bit is a 1, the right child is selected. The algorithm then moves to the selected child node and repeats the process with the next bit in the codeword. When a leaf node is reached, the symbol stored in the leaf node is the symbol represented by the codeword. The algorithm then returns to the root node to begin decoding the next codeword in the input.

The Lempel-Ziv-Welch algorithm for decoding a codeword is similar to the algorithm for decoding a codeword in a static Huffman code. The difference is that the Lempel-Ziv-Welch algorithm must also update the tree. The tree is updated by adding two new nodes to the tree each time a codeword is decoded. One of the new nodes is assigned to be the right child of the node that was reached when the last bit of the codeword was decoded. The other new node is assigned to be the left child of the node that was reached when the last bit of the codeword was decoded. The symbol stored in the node that was reached when the last bit of the codeword was decoded is then stored in the right child node. The symbol stored in the left child node is the symbol that will be decoded next time the root node is reached. For example, suppose the letter 'a' is stored in node 4 and the letter 'b' is stored in node 5. If the codeword 01 is decoded, then node 4 will be reached when the last bit is decoded. Node 4 will then be updated to have a right child node containing the symbol 'a' and a left child node containing the symbol 'b'. Node 4 will also be updated to have no symbol stored in it. Finally, the root node will be reached when the last bit of the codeword is decoded. The root node will be updated to have a right child node containing the symbol 'a' and a left child node containing the symbol 'b'. The root node



list of unused symbols associated with its node. The two codewords for earlier parts of the codeword are marked in Figure 11.8 by underlining. Each time a symbol is removed from the list its node is replaced by the last symbol in the list. This distinction is not necessary and could be omitted from the algorithm description, but it does affect the codewords produced.

```
BUILDHUFFMANTREE(weight[0..n-1])
    for i ← 0 to n-1 do
        leaf[i] ← true
        symbol[i] ← i
    for i ← n to 2n-2 do
        leaf[i] ← false
```

```
if p is the 0-node
   c = c concatenated with the number of 1s representing position of s in 0-node and with 0;
   write the last symbol in 0-node over s in this node;
   create a new node q for symbol s and set its counter to 1;
   p = new node to become the parent of both 0-node and node q;
   counter(p) = 1;
   include the two new nodes to nodes;
else increment counter(p);
while p is not the root
   if p violates the sibling property
      if the leader of the block_i that still includes p is not parent(p)
         swap p with the leader;
   p = parent(p);
   increment counter(p);
return codeword c;
```

A step-by-step example for string *aafcebd* is shown in Figure 11.8.

1. Initially, the tree includes only the 0-node with all the source letters (*a*, *b*, *c*, *d*, *e*, *f*). After the first input letter, *a*, only the codeword for the position occupied by *a* in the 0-node is output. Because it is the first position, one 1 is output followed by a 0. The last letter in the 0-node is placed in the first position, and a separate node is created for the letter *a*. The node, with the frequency count set to 1, becomes a child of another new node that is also the parent of the 0-node.
2. After the second input letter, also an *a*, 1 is output, which is the Huffman codeword for the leaf that includes *a*. The frequency count of *a* is incremented to 2, which violates the sibling property, but because the leader of the block is the parent of node p (that is, of node *a*), no swap takes place; only p is updated to point to its parent, and then p's frequency count is incremented.
3. The third input letter, *f*, is a letter output for the first time; thus, the Huffman codeword for the 0-node, 0, is generated first, followed by the number of 1s corresponding to the position occupied by *f* in the 0-node, followed by 0: 10. The letter *e* is put in place of letter *f* in the 0-node, a new leaf for *f* is created, and a new node becomes the parent of the 0-node and the leaf just created. Node p, which is the parent of leaf *f*, does not violate the sibling property, so p is updated, p = *parent*(p), thereby becoming the root that is incremented.
4. The fourth input letter is *c*, which appears for the first time in the input. The Huffman codeword for the 0-node is generated followed by three 1s and a 0 because *c* is the third letter in the 0-node. After that, *d* is put in place of *c* in the 0-node, and *c* is put in a newly created leaf; p is updated twice allowing for incrementing counters of two nodes, left child of the root and the root itself.
5. The letter *c* is the next input letter; thus, first the Huffman codeword for its leaf is given, 001. Next, because the sibling property is violated, the node p (that is, the leaf *c*), is swapped with the leader *f* of $block_1$ that still includes this leaf. Then, p = *parent*(p), and the new parent p of the *c* node is incremented, which leads to another violation of the sibling property and to an exchange of node p with the leader of

blocks, namely, with the codeword. Next, the encoding node is marked. In the example in Figure 11.7, the process is repeated until the process is finished.

6. The next input letter is read. If it is a leaf in the tree, then the Huffman codeword for the leaf is generated and the counter of the node is incremented. If the node, which is the leaf, violates the sibling property, it is swapped with the leader, and it is marked. Now, if the parent node's counter is incremented, and it is also the root, the tree transformation is complete for this input letter. The remaining steps can be traced in Figure 11.8.

It is left to the reader to make appropriate modifications to this pseudocode to obtain a `DGKDynamicHuffmanDecode` algorithm.

It is possible to design a Huffman coding that does not require any initial knowledge of the set of symbols (see Gallager [1978] and Cormack and Horspool [1984]). The Huffman tree is initialized to a special escape character. For a new symbol to be sent, it is preceded by the escape character (or its current codeword in the tree), and followed by the symbol itself. The receiver can now know this symbol so that it is a codeword or not later on, it can be properly decoded. The symbol is inserted in the tree by making the old node with the lowest frequency count (call it x) have two children, one pertaining to the symbol previously in x and one to the new symbol.

Adaptive Huffman coding surpasses simple Huffman coding in two respects: It requires only one pass through the input, and it adds only an alphabet to the output. Both are measured in bytes, and are important. They can be applied to many types of files, not archive files. In particular, they can compress executable files, not archive files. The trade-off with executable files, however, is that they generally use byte-aligned machine instructions (called *opcodes*), and the variety of these opcodes is more uniform than in text files. Therefore, the Huffman trees are large, the codewords are relatively long, and the resulting files are not much smaller than the original. Encoding is usually very fast.

11.3 Shannon-Fano Code

Another variation of Huffman coding is the Shannon-Fano code. Its steps are similar to those described in Section 11.2. Shannon and Weaver's description of it is as follows:

```
Order the set of symbols according to the proposed probabilities;
Shannon-Fano Code (sequence S):
    if S contains two symbols
        assign 0 to the codeword of one symbol and 1 to the other symbol;
    else if S contains more than two symbols
        divide S into two subsequences S1 and S2 such that the total
            difference between probabilities of each subsequence is
        assign the codewords for each symbol in S1 by appending 0 to the beginning
            of each symbol in S1;
        ShannonFano (S1);
        ShannonFano (S2);
```

FIGURE 11.9 Description of Shannon-Fano algorithm applied to five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09.

A	0	.39
B	1	.21
C	2	.19
D	3	.12
E	4	.09

Figure 11.9 contains a message alphabet $\{A, B, C, D, E\}$ and the probabilities $P(A) = .39$, $P(B) = .21$, $P(C) = .19$, $P(D) = .12$, and $P(E) = .09$. Because $P(A) + P(B) = .60$ and $P(A) + P(B) + P(C) = .79$, the first two letters are assigned to S_0 and the third to S_1. Because $P(A) + P(B) + P(C) = .79$ and $P(D) + P(E) = .21$, the first three letters are assigned to S_0 and the last two to S_1. Because $P(A) = .39$ and $P(B) = .21$, the first letter is assigned to S_0 and the second to S_1. Because $P(C) = .19$ and $P(D) + P(E) = .21$, the third letter is assigned to S_0 and the last two to S_1. Because $P(D) = .12$ and $P(E) = .09$, the fourth letter is assigned to S_0 and the last to S_1. The codewords for A and B are formed by attaching 0 and 1, respectively, to the codeword for S_0. The codewords for C and D are formed by attaching 0 and 1, respectively, to the codeword for S_1. The codewords for E is formed by attaching 1 to the codeword for S_1. Because S_1 has two elements, the codewords for C and D are extended by adding 0 and 1, respectively. Similarly, the codeword for E is extended by adding 1 to the codeword for S_1. Because S_2 has only one element, its codeword remains unchanged. The codewords are summarized in Figure 11.9 and are uniquely decipherable.

The average length of the codewords generated by the Shannon-Fano method for the five letters A, B, C, D, and E probabilities .39, .21, .19, .12, and .09 is

$$L_{SF} = .39(2) + .21(2) + .19(2) + .12(3) + .09(3) = 2.27$$

which is the same as the average codeword length generated by the Huffman algorithm. Generally, however, it is not true that the Shannon-Fano method produces an average codeword length as small as the Huffman algorithm. Because of its greater simplicity, the Shannon-Fano method is sometimes used in practice.

Both the Huffman and Shannon-Fano algorithms are concerned with encoding a message, a string of symbols with different probabilities of occurrence, by assigning variable-length codewords to the symbols. The symbols with the highest probabilities are assigned the shortest codewords. This reduces the size of the encoded message. However, a disadvantage of both algorithms is that they require knowledge of the probabilities of the symbols in the message. If the probabilities are not known, they must be computed by scanning the message before encoding it. Also, the codewords must be stored with the message, which increases the size of the encoded message. The Lempel-Ziv algorithm, which is described in the next section, does not require knowledge of the probabilities of the symbols in the message before encoding it. Also, the codewords do not have to be stored with the message. The Lempel-Ziv algorithm is a self-correcting algorithm that can be used for data compression.

11.4 Run-Length Encoding

A run is defined as a sequence of identical characters. For example, the string s = "aaabbbbbcc" has three runs: a run of three 'a's followed by runs of two 'b's and of one 'a.' The run length encoding technique takes advantage of the presence of runs and represents them in an abbreviated, compressed form.

If runs are on the same characters as in the string s = "aaabbbbbcc," then instead of transmitting this string, information about runs can be transferred. Each run is coded by the pair (n, ch), where ch is a character and n is the integer representing the number of consecutive characters ch in the run. The string s is coded as (3,'a')(5,'b')(2,'c'). However, a problem arises if one of the characters being transferred is a digit, as in ..... 11111111111111, which is represented as (11)(5)(1). For eleven 1s, one 1 and five 1s. Therefore, for each run, instead of the number n, a character can be used whose ASCII value is n. For example, the run of 15 consecutive letters 'a' is represented as (-a,'-') has ASCII code 15), and the run of 14 1s is coded as (-n,'1') (-n has ASCII code 14).

This technique is only efficient when at least two character runs are transmitted, because for one character runs, the codeword is twice as long as the character. Therefore, the technique should be applied only to runs of at least two characters. This requires using a marker indicating that what is being transmitted is either a run in an abbreviated form or a literal character. Three characters are needed to represent a run: a compression marker cm, a literal character ch, and a counter n, which make up a triple (cm, ch, n). The problem of choosing the compression marker is especially delicate since it should not be confused with a literal character being transmitted. For a regular text file is transmitted, then the character '~' can be chosen. If there is no restriction on the characters transmitted, then wherever the compression marker itself occurs in the input file, we transmit the compression marker twice. The decoder discards one such marker upon receiving two of them in a row and retains just one as part of the data being received. For example, to print just one backslash. Since for each literal marker two of them must be sent, an infrequently used marker should be chosen. In addition, runs of markers are not sent in compressed form.

Since compressing runs results in a sequence of three characters, this technique should be applied to runs of at least four characters. The maximum length of a run that can be represented by the triple (cm, ch, n) is 255 for 8-bit ASCII if the number n represents the number of characters in the run. But because only runs of two or more characters are encoded, n can represent the number of actual characters in the run minus 1. For example, if n = 1, then there are two characters in the run. In this case, the longest run representable by one triple has 256 characters.

Run length encoding is only moderately efficient for text files in which only the blank character has a tendency to be repeated. In this case, a predecessor of this technique can be applied, null suppression, which compresses only runs of blanks and eliminates the need to identify the character being compressed. As a result, pairs (cm, n) are used for runs of three or more blanks. This simple technique is used in the IBM 3780 BISYNC protocol, where the graphic print line is between 30 and 50 c.

Run-length encoding is very useful when applied to files which are known to have many runs of at least four characters. One example is relational databases. All records in the same relational database file have to be of equal length. Records consist of a collection of fields, which may be—and most frequently are—longer than the information stored in them. Therefore, they have to be padded with some characters, thereby creating a large collection of runs whose only purpose is to fill up unused space in each field of every record.

Another candidate for compression using run length encoding is fax images, which are composed of combinations of black and white pixels. For low resolution, there are about 1.5 million pixels per page. Thus, transmission of one page at 28,800 bps requires about 1 minute. Clearly, some compression method is necessary.

A serious drawback of run length encoding is that it relies entirely on the occurrences of runs. In particular, this method taken by itself is unable to recognize the high frequency of the occurrence of certain symbols which call for short codewords. For example, AAAABBBB can be compressed, since it is composed of two runs, but ABABABAB cannot, although both messages are made up of the same letters. On the other hand, ABABABAB is compressed by Huffman encoding into the same number of codewords as AAAABBBB without taking into consideration the presence of runs. Therefore, it seems appropriate to combine both methods as in this chapter's case study.

11.5 Ziv-Lempel Code

The problem with some of the methods discussed thus far is that they require some knowledge about the data before encoding takes place. A "pure form" of the Huffman encoder has to know the frequencies of symbol occurrences beforehand, and codewords are assigned to the symbols. Some versions of the adaptive Huffman encoding can circumvent this limitation, not by relying on previous knowledge of the source characteristics, but by building this knowledge in the course of data transmission. Such a method is called a universal coding scheme, and Ziv-Lempel code is an example of a universal data compression code.

In a version of the Ziv-Lempel method called LZ77, a buffer of symbols is maintained. The first P position is held for A_1 most recently encoded symbols from the input, and the remaining L positions contain the L symbols about to be encoded. In each iteration, search up from one of the first A_1 positions in the buffer is conducted for a substring matching a prefix of a string that is in the second portion of the buffer. If such a match is found, a codeword is transmitted. The codeword is a triple composed of the position where the match was found, the length of match, and the first nonmatching symbol. Then the entire contents of the buffer is shifted to the left by the length of match plus one. Simultaneously, we shift out symbols from the input and shift in the same number of new symbols. This process fills the first A_1 positions with copies of the first symbol at the input.

As an example, consider the case when $A_1 = 6$, and the input is the string "abcbcbcb

Figure 11.10 Encoding the string "sababacabacabaabagagag

Strings are always at least one character long. After reading a new character, the concatenation of string and character is checked in the table. A new character is read if the concatenation is in the table. Otherwise, the codeword for string is output, the concatenation is stored in the table, and string is initialized to character. Figure 11.7 shows a trace of the execution of this algorithm applied to the input "ababababababa...". The legend contains the generated codewords, the strings included in the table, and their codewords. Codewords are represented by a number and a character.

A second shortcoming of efficiency is the encoding part of the table. Typically, one needs to locate codewords, but the last fragments of entries are exported in this table so that an efficient search procedure has to be used. A second concern is the size of the table, which grows particularly at the beginning of the compression. The problem of size is addressed by storing in the table codewords for the prefix and the last character of strings. For example, if "ba" is assigned the codeword 7, then "baa" can be stored in the table as the pair (7,"a"). Instead of the character "a", it is actually sufficient to store a pointer to the position where "a" is stored. In this way, all table entries have the same length. The problem of searching is addressed by using hash functions.

For decoding, the same table is created by inputting the encoded message and the exception is treated. Each codeword is decoded to a corresponding prefix and last character retrieved from the table. Since the prefix is also encoded, it is split into its last character and the table lookup for the entire string is started. This scheme is somewhat problematic which may be circumvented with an explicit stack. This is necessary since the decoding process applied to particular inputs yields a string in the reverse order. The decoding procedure can be summarized as follows:

```
TWOencompress();
    initialize table for the index;
    read priorcodeword and output its character (uncompressed so far);
    while (codewords are still left)
        read codeword;
        if (codeword is not in the table) // special case: codeword = tableindex
            enter in table string(priorcodeword) + firstchar(string(priorcodeword));
            output string(priorcodeword) + firstchar(string(priorcodeword));
        else enter in table string(priorcodeword) + firstchar(string(codeword));
            output string(codeword);
        priorcodeword = codeword;
```

This relatively simple algorithm has to consider a special case, when a codeword is not processed but a corresponding entry in the table. This situation arises when the string, being decoded, contains a substring "xbx" where "x" is a single character and "xb" is already in the table.

All of the discussed compression algorithms are widely used. UNIX tool `compress` compression program: `pack` uses the Huffman algorithm, `compact` is based on the adaptive Huffman method, and `compress` uses LZW coding. According to system literature, `pack` compresses text files by 25–40%, `compact` by 40%, and `compress` by 50–60%. The rate of compression is better for Ziv Lempel coding. It is also used

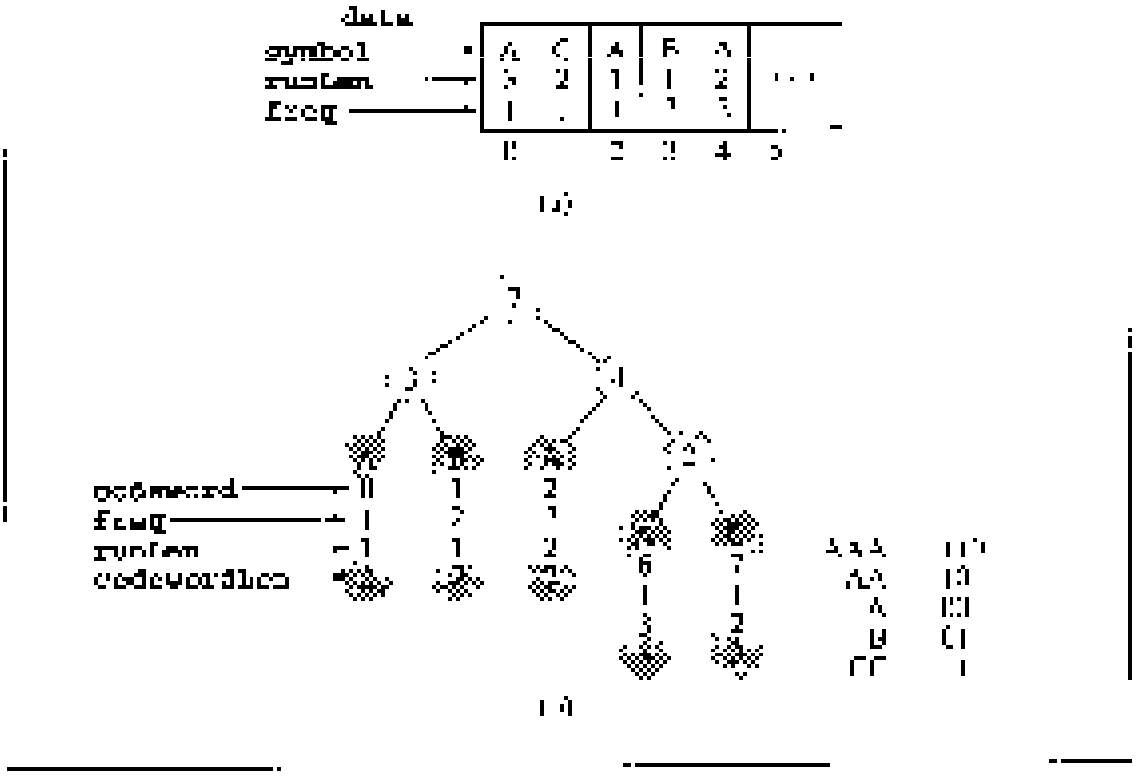
EXAMPLE 11.11 LZW applied to the string "abababababa"

encoder input	encoder output	table index (codeword)	table full string	abbreviated string
		1	a	a
		2	b	b
		3	c	c
a		4	ab	ab
b	1	5	ba	ba
ab	4	6	aba	aba
a	1	7	abab	abab
ba	5	8	bab	bab
a	1	9	baba	baba
ba	5	10	baba	baba
a	1			

11.6 Case Study: Huffman Method with Run-Length Encoding

As indicated in the discussion of run-length encoding, this method is suitable only for files which are known to guarantee in advance that at least one symbol does not occur. If no compression is achieved. The Huffman algorithm can also be used in conjunction with any run-encoding scheme that produces symbols. This method can be applied to single symbols, such as characters, but also to pairs of symbols, triples, or to any combination of variable-length sequences of symbols. Incorporating run-length encoding in the Huffman method works exceedingly well for files with many runs and moderately well for files with a small number of runs and a large number of different symbols.

FIGURE 11.12 (a) Contents of the array data after the message AAABAACCAABA has been processed. (b) Huffman tree generated from these data.



For files with no runs, this method reduced to plain Huffman encoding. In this approach, a file to be compressed is scanned first to determine all runs, isolated symbols, and the symbols long. Runs composed of the same symbol but of different lengths are treated as different "super-symbols" which are used to create a Huffman tree. For example, if the message to be compressed is AAABAACCAABA, then the super-symbols used in the Huffman tree are AAA, B, AA, CC, and A, rather than the symbols A, B and C. In this way, the number of codewords to be created grows from three (for the symbols A, B and C) to five for the super-symbols. The compression ratio becomes larger, for the codewords assigned to the runs are much shorter than in straight run-length encoding, or run-length encoding. This codeword is always three bytes long (24 bits). In Huffman code, it may be even one bit long.

Figure 11.12a illustrates the positions of the data in the sorted vector. Note that the sorted data are stored in the output file to be used by the decoder to reconstruct the same Huffman tree. Next, the encoder is allowed to create a new instance of HuffmanTree, which creates the tree of Huffman codewords using information collected in data. To do that, each node is a doubly linked list of single-node trees similar to the one in Figure 11.3 is created first. Then, repeatedly, the two trees with the lowest frequencies are combined to create a new tree, which eventually results in the Huffman tree as in Figure 11.12b.

After the tree has been created, the positions of the nodes, in particular the leaves, can be determined whereby the codewords of all symbols in the leaves can be generated. Each node in this tree has seven data members, but only five of them are shown, just for leaves. The codewords are stored as numbers which represent binary sequences of 0s and 1s. For example, the codeword for QC is 7: 111 in binary. However, these numbers are always the same length, and 7 is stored as 3 bits set to 1 preceded by 29 bits set to 0: 0 . . . 0111. It is, therefore, unclear how many bits out of 32 are included in the sequence representing the codeword for a certain symbol. Is it 111, 0111, 00111, or some other sequence? The codeword field for single As is 0. Is the codeword for AE, 00, 000, or some other 0s? To avoid ambiguity, the codewordLen field stores the number of bits included in the codeword for a given symbol. Since codewordLen for A is 1 and codeword is 0, then the codeword sequence representing A is 00.

After the Huffman tree is generated and the leaves are filled with relevant information, the process of coding information in the input file can be initiated. Because searching for particular symbols directly in the tree is too time-consuming, an array of pairs[] of linked lists corresponding to each ASCII symbol is created. The nodes of the linked lists are simple leaves of the tree linked through right pointers, and each list has as many nodes as the number of different run lengths of a given symbol in the input file. It gives immediate access to a particular linked list, but some linked lists may be long, if there are many run lengths of a given symbol.

Next, the file is scanned for the second time to find each super-symbol and its corresponding codeword in the Huffman tree and to transmit it to the output file. As the sequences are retrieved from the tree, they are tightly packed into a four-byte numerical variable pack. The first encountered super-symbol in the input file is AA4 with codeword 110, which is stored in pack so that pack contains the sequence 0 . . . 0110. After B is retrieved from the file, its code, 01, is attached to the end of pack. As a result, the contents of pack have to be shifted to the left by two positions to make room for 01, and then 01 is stored in it using the bitwise or operation |. Now, pack contains the string 0x0110 01. After pack is built up with codewords, it is output as a sequence of four bytes to the output file.

Particular care has to be taken to put exactly 32 bits in pack. When there are fewer available positions in pack than the number of bits in a codeword, only a portion of the codeword is put in pack. Then pack is output and the remaining portion of the codeword is put in pack before any other symbol is encoded. For example, if pack currently contains 001 . . . 1001; pack can take only two more bits. Since the codeword 1101 is four bits long, the contents of pack are shifted to the left by two positions: 0 . . . 100110 01 and the first two bits of the codeword, 11, are put at the end of pack after which pack's contents are 1 . . . 100111. Next, pack is output as four bytes (characters), and then the remaining two bits of the codeword, 01, are put into pack, which now contains 0 . . . 001.

Another problem is with the last codewords. The encoder fills the output file with bytes (in this case, with chunks of four bytes, each containing eight bits). What happens if there are no symbols left, but there is still data in pack? The decoder has to know that some bits at the end of file should not be decoded. If they are, some spurious characters will be added to the decoded file. In this implementation, the problem is solved by transmitting at the beginning of the encoded file the number of characters

node descriptor. Each node descriptor consists of a pointer to the node's left child, a pointer to the node's right child, and a pointer to the node's parent. The message AAABAACB AABAA is encoded as the sequence of codewords 11010110111010101, and the contents of pack are 0000000000000000110101111010100. If the encoding process is finished, the contents are padded on the left by the number of unused bits, already packed in order 1100111111100000000000000000000 and is output as a sequence of four bytes 11001101, 11111000, 00000000, and 00000000 or, in more readable decimal notation, as 205, 248, 0, and 0. The last 16 bits do not represent any codewords, and if they are included, they are decoded as eight As, whose codeword is 00. To prevent this, the output file includes the number of encoded characters, namely, 13 (A, A, A, B, A, A, C, A, A, B, A, A, A). The output file also includes the number of all symbols in the Huffman tree. For this example, it is the number 5, since the five different symbols can be found in the input (A and B in the Huffman tree A(6), B(3), C(1), and a). Therefore, the structure of the output file is as follows: the number of super-symbols, data size, number of characters, contents of pack for pads, pad lengths, and the symbols and codewords of all super-symbols, all in the output file.

Decoder is much shorter than encoder because it uses information supplied by the encoder in the header of the encoded message. Decoder recreates first the array data[] in inputTreeFrequencies(), then reconstructs the Huffman tree with the same createHuffmanTree() and createCodewords() decoder methods finally in decode() scans the tree in the order determined by the sequence of bits in the compressed file to recover the encrypted symbols.

As expected, this implementation gives particularly good results for files or files with a small repertoire of characters. The compression ratio for bib files is 50% (in terms of percentages), for text files, 57%, and for executable files, merely 73%.

Figure 11.13 contains the complete code for the encoder.

FIGURE 11.13 Implementation of Huffman method with run-length encoding.

```
//////////////////////////////////////////////////////////////////////////
// huffmanEncoding.cpp ////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <algorithm>

using namespace std;

class huffmanNode {
public:
    char symbol;
    unsigned long codeword, freq;
    unsigned int number, codewordLen;
```

Figure 11.13 (continued)

```
    HuffmanNode *left, *right;
    HuffmanNode() {
        left = right = 0;
    }
    HuffmanNode(char s, unsigned long f, unsigned int c,
        HuffmanNode *lt = 0, HuffmanNode *rt = 0) {
        symbol = s; freq = f; codeLen = c; left = lt; right = rt;
    }
};

struct ListNode {
public:
    HuffmanNode *tree;
    ListNode *next, *prev;
    ListNode() {
        next = prev = 0;
    }
    ListNode(ListNode *p, ListNode *n) {
        prev = p; next = n;
    }
};

class DataRec {
public:
    char symbol;
    unsigned int runLen;
    unsigned long freq;
    DataRec() {
    }
    bool operator==(const DataRec& dr) const { // used by find()
        return symbol == dr.symbol && runLen == dr.runLen;
    }
    bool operator<(const DataRec& dr) const { // used by sort()
        return freq < dr.freq;
    }
};

class HuffmanCoding {
public:
    HuffmanCoding() : mask(0xff), bytes(4), bits(8), ASCII(256) {
```

FIGURE 11.13 (continued)

```
        chars = new HuffmanNode[asCii-1];
    }
    void compress(chars.ifstreams);
    void decompress(chars,ifstreams);
private:
    const unsigned int bytes, bits, asCii;
    unsigned int dataSize;
    const unsigned long mask;
    unsigned long charCnt;
    ifstream fout;
    HuffmanNode *huffTree(), **chars;
    vector<bool_vector> data;
    void error(char *s) {
        cerr << s << endl; exit(1);
    }
    void output(unsigned long pack);
    void generateData(ifstream&);
    void outputRegularBits(ifstream&);
    void readBytesHuff(unsigned int&,ifstream&);
    void readBytesHuff(unsigned long&,ifstream&);
    void inputRegularBits(ifstream&);
    void createHuffTree();
    void createCodes(HuffmanNode*,unsigned long,int);
    void createCharPointerArrayOfLists(HuffNode**);
    void encode(ifstream&);
    void decode(ifstream&);
};

void HuffmanCoding::output(unsigned long pack) {
    char *s = new char[bytes];
    for (int i = bytes - 1; i >= 0; i--) {
        s[i] = pack & mask;
        pack >>= bits;
    }
    for (i = 0; i < bytes; i++)
        fout.put(s[i]);
}

void HuffmanCoding::generateData(ifstream& in) {
    char ch, ch2;
    Bytebuf r;
```

FIGURE 11.13 (continued)

```
    vector<DataRec>::iterator i;
    n.freq = 1;
    for (fIn.get(ch); !fIn.eof(); ch = ch2) {
        for (r.number = 1, fIn.get(ch2); !fIn.eof() && ch2 == ch;
 r.number++)
            fIn.get(ch2);
        r.symbol = ch;
        i = find(data.begin(),data.end(),r); if (i == data.end())
            data.push_back(r);
        else i->freq++;
    }
    sort(data.begin(),data.end());
}

void HuffmanCoding::output_frequencies(ifstream& fIn) {
    unsigned long temp4;
    char ch = data.size();
    unsigned int temp2 = data.size();
    temp2 >>= bits;
    fOut.put(char(temp2)).put(ch);
    fIn.clear();
    output((unsigned long)fIn.tellg());
    for (int i = 0; i < data.size(); i++) {
        fOut.put(data[i].symbol);
        ch = temp2 - data[i].number;
        temp2 >>= bits;
        fOut.put(char(temp2)).put(ch);
        temp4 = data[i].freq;
        output(temp4);
    }
}

void HuffmanCoding::read2bytes(unsigned int& num, ifstream& fin) {
    num = fIn.get();
    num <<= bits;
    num |= fIn.get();
}

void HuffmanCoding::read4bytes(unsigned long& num, ifstream& fin) {
    num = (unsigned long) fin.get();
    for (int i = 1; i < 4; i++) {
```

(continues)

Figure 11.13 (continued)

```
        num <<= bits;
        num |= (unsigned long) tIn.get();
    }
}

void ExtmanCoding::ReportFrequencies(ifstream& tIn) {
    DataRec r;
    read3ByteNum(dataSize,tIn);
    read4ByteNum(dataCnt,tIn);
    data.reserve(dataSize);
    for (int j = 0; !tIn.eof() && j < dataSize; j++) {
        r.symbol = tIn.get();
        read2ByteNum(r.ranLen,tIn);
        read4ByteNum(r.freq,tIn);
        data.push_back(r);
    }
}

void ExtmanCoding::createExtmanTree() {
    ListNode *p, *newNode, *head, *tail;
    unsigned long nextFreq;
    head = tail = new ListNode;              // initialize list pointers;
    head->tree = new
                 ExtmanNode(&data[0].symbol,data[0].freq,data[0].ranLen);
    for (int i = 1; i < dataSize(); i++) { // create the rest of the
                                           // list;
        tail->next = new ListNode(tail,0);
        tail = tail->next;
        tail->tree
            = new ExtmanNode(data[i].symbol,data[i].freq,data[i].ranLen);
    }
    while (head != tail) {                 // create one Extman tree;
        nextFreq = head->tree->freq + head->next->tree->freq; // two
                                           // lowest frequencies
        for (p = tail; p != 0 && p->tree->freq > nextFreq; p = p->prev);
        newNode = new ListNode(p,p->next);
        p->next = newNode;
        if (p == tail)
            tail = newNode;
        else newNode->next->prev = newNode;
```

Program 11.13 (continued)

```
        nextNode->item =
            new HuffmanNode('\0', newArray,0,head->item.freq+next->item.freq);
        head = head->next->next;
        delete head->prev->prev;
        delete head->prev;
        head->prev = 0;
    }
    EncTreeItem = head->item;
    delete head;
}

void HuffmanCoding::preorderSymbols(HuffmanNode *p, unsigned long
codeword, int level) {
    if (p->left == 0 && p->right == 0) {          // if p is a leaf,
        p->codeword    = codeword;                // store codeword
        p->codewordLen = level;                   // and its length,
    }
    else {                                        // otherwise add 0
        preorderSymbols(p->left,  codeword<<1,    level+1); // for left
                                                            // branch
        preorderSymbols(p->right,(codeword<<1)+1,level+1); // and 1 for
                                                            // right;
    }
}

void HuffmanCoding::transformTreeToArrayOfPointers(HuffmanNode *p) {
    if (p->left == 0 && p->right == 0) {          // if p is a leaf,
        p->right = chars[(unsigned char)p->symbol]; // initialize it in
        chars[(unsigned char)p->symbol] = p;      // a list associated
    }                                             // with symbol found in p;
    else {
        transformTreeToArrayOfPointers(p->left);
        transformTreeToArrayOfPointers(p->right);
    }
}

void HuffmanCoding::encode(ifstream &in) {
    unsigned long packCnt = 0, bold, maxPack = bytesToEncode, pack = 0;
    char ch, chx;
     int bitsLeft, runLength;
```

FIGURE 11.13 (continued)

```
    int ch = fIn.getch(); if(fIn.eof()) { [
        for (runLength = 1; fIn.getch(ch2); !fIn.eof();) && ch2 == ch)
            runLength++;
        fIn.getch(ch);
        for (HuffmanNode *p = chars[(unsigned char) ch];
            p != 0 && runLength != p->runLen; p = p->right)
        ;
        if (p == 0)
            error("A problem in encode()");
        if (p->codewordLen < maxPack - packCnt) {    // if enough room in
            pack = (pack << p->codewordLen) | p->codeword; // pack co
                                                    // store new
            packCnt += p->codewordLen;              // codeword, shift
        }                                           // the content to the
                                                    // left and attach
                                                    // new codeword;
        else {                                      // otherwise move
            bitsLeft = maxPack - packCnt;           // pack's content to
            pack <<= bitsLeft;                      // the left by the
            if (bitsLeft != p->codewordLen) {       // number of left
                hold = p->codeword;                 // spaces and if new
                hold >>= p->codewordLen - bitsLeft;// codeword is
                                                    // longer than room
                pack |= hold;                       // left, transfer
            }                                       // only as many bits as
                                                    // can be fitted in pack;
            else pack |= p->codeword;               // if new codeword
                                                    // exactly fits in
                                                    // pack, transfer it;
            output(pack);                           // output pack as
                                                    // four chars:
            if (bitsLeft != p->codewordLen) {       // transfer
                pack = p->codeword;                 // unprocessed bits
                packCnt = maxPack - (p->codewordLen - bitsLeft); // of
                packCnt = p->codewordLen - bitsLeft; // new codeword to
                                                    // pack;
            }
            else packCnt = 0;
        }
        ch = ch2;
```

FIGURE 11.13 (continued)

```
    }
    if (packCnt != 0) {
        pack <<= maxWork - packCnt; // transfer leftover codewords and
                                    // zero fill
        unPut_(pack);
    }
}

void HuffmanEncoding::compress(char *inFileName, ifstream& in) {
    char outFileName[30];
    strcpy(outFileName,inFileName);
    if (strchr(outFileName,'.'))                // if there is an extension
        strcpy(strchr(outFileName,'.')+1,"x");// overwrite it with 'x'
    else strcat(outFileName,".x");              // else add extension ".x";
    fout.open(outFileName,ios::out|ios::binary);
    getFreq(in);
    buildHuffTree();
    createCodewords(huffTreeRoot,0,0);
    for (int i = 0; i <= ASCII; i++)
        chars_ += 0;
    transferHuffTreeToArray(huffTreeRoot);
    fin.clear();        // clear eof especially if was set also:
    fin.seekg(0, ios::beg);
    encode(in);
    cout.precision(2);
    cout << "Compressed on data = " <<
            100.0*(float(fout.tellp())/fin.tellg()) << "%\n"
         << "Compressed on data without table = " <<
            100.0*(float(fout.tellp()-
                   (int)(sizeof(int)*(1+2*4)))/fin.tellg());
    fout.close();
}

void HuffmanEncoding::decode(ifstream &in) {
    unsigned long long code;
    char ch, bitCnt = 1, mask = 1;
    mask <<= bits - 1;          // changes 00000001 to 10000000
    for (chars = 0, in.get(ch); in.eof() && chars < charCnt; ) {
        for (HuffmanNode *p = huffTreeRoot; p != 0; {
```

Continued

Listing 11.13 (continued)

```
            if (p->left == 0 && p->right == 0) {
                for (int j = 0; j < p->runLen; j++)
                    fOut.put(p->symbol);
                charx += p->runLen;
                break;
            }
            else if (ch & mask) == 0)
                p = p->left;
            else p = p->right;
            if (bitCnt++ == bits) { // read next character from fIn
                fIn.get(ch);        // if all bits in ch are checked
                bitCnt = 1;
            }                       // otherwise move all bits in ch
            else ch <<= 1;          // to the left by one position
        }
    }
}

void HuffmanCoding::decompress(char *inFileName, ifstream& fIn) {
    char outFileName[30];
    strcpy(outFileName,inFileName);
    if (strchr(outFileName,'.'))              // if there is an extension
        strcpy(strchr(outFileName,'.')+1,"dec");// overwrite it with 'z'
    else strcat(outFileName,".dec");          // else add extension 'z'
    fOut.open(outFileName,ios::out|ios::binary);
    inputFrequencies(fIn);
    createHuffmanTree();
    createCodewords(HuffmanTree,0,0);
    decode(fIn);
    fOut.close();
}

//***************************** HuffmanEncoder.cpp ****************************

#include <iostream>
#include <fstream>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include "HuffmanCoding.h"
```

Figure 11.13 (continued)

```
void main(int argc, char* argv[]) {
    char fileName[30];
    HuffmanCoding Huffman;
    if (argc != 2) {
        cout << "Enter a file name: ";
        cin  >> fileName;
    }
    else strcpy(fileName,argv[1]);
    ifstream fIn(fileName,ios::binary);
    if (!fIn.fail()) {
        cerr << "Cannot open " << fileName << endl;
        return;
    }
    Huffman.decompress(fileName,fIn);
    fIn.close();
    }
```

11.7 EXERCISES

1. For which probabilities $\{p_i\}$ of n symbols is the average length maximal? When is it minimal?
2. Find l_{ave} for the letters X, Y, and Z and their probabilities .05, .05, and .9 and compare it to l_{est} computed for single letters and pairs of letters, as in Figure 11.6. Does l_{est} satisfactorily approximate l_{ave}? How can we remedy the problem?
3. Assess the complexity of all the implementations of the Huffman algorithm suggested in this chapter.
4. What are the lengths of the Huffman codewords of the most probable messages with respect to each option?
5. In the adaptive Huffman algorithm, first the codeword for an unprocessed symbol is issued and then the conversion table is updated. Could the table be updated first and then the new codeword for the symbol be issued? Why or why not?

6. [In Exercise 5 we saw that the Lempel-Ziv algorithm will sometimes encode a string using a codeword that has not been added to the dictionary yet. This can happen when the encoder comes across a string of the form $cwcwx$, where c is a single character, w is a (possibly empty) substring, and x is a string that starts with c. Suppose that cw is in the dictionary but cwc is not. The encoder will output the codeword for cw and then, when it comes to cwc, will output a codeword for a string it has not yet added to the dictionary. Explain how the decoder can still correctly decode.]

7. [What problem arises if the Lempel-Ziv algorithm runs out of codewords and a larger codeword size is needed? Describe three different approaches to dealing with this problem.]

8. [Explain why the greedy approach to parsing LZ77 factorizations (i.e., always choosing the longest match at each step) does not always result in an optimal factorization. Does it ever result in a suboptimal factorization?]

9. [In Figure 11.18, $i_1 = i_2 = i_3 = 3$. Describe an input sequence for which $i_1 = i_2 = i_3 = 3$ simplify the implementation of the LZ77 algorithm?]

10. [In which situations does the LZ77 algorithm produce a 0?]

11. [Describe the process of decoding a string encoded by the LZ77 algorithm. Assume that the encoder has already been described.]

12. [Compare the LZ77 algorithm with the LZW algorithm. Which algorithm produces smaller encodings?]

11.8 PROGRAMMING ASSIGNMENTS

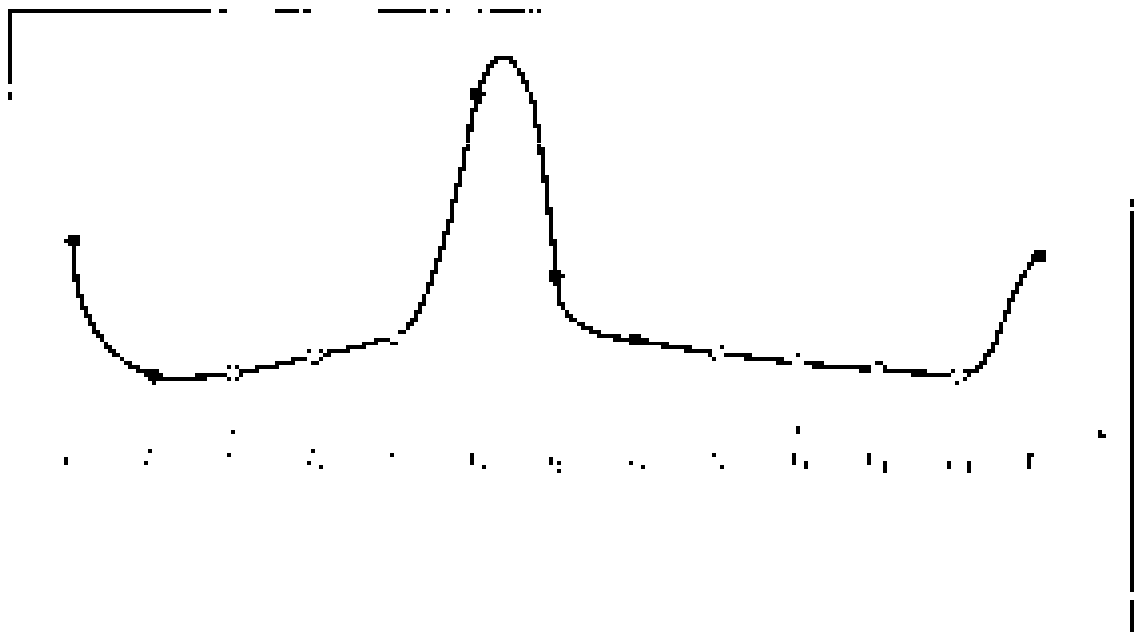
1. [Implement the LZW algorithm. Your program should consist of two parts: a function that encodes a string of characters into a sequence of integers, and a function that decodes a sequence of integers produced by the encoding function back into the original string of characters. Your encoding function should write its output to a file, and your decoding function should read its input from the file written by the encoding function.]

2. [Write an encoder and decoder for the Lempel-Ziv algorithm.]

3. [Write an encoder and a decoder for a Lempel-Ziv algorithm that uses variable-length codewords. The encoder should begin with a dictionary containing all strings of length 1. Whenever the encoder finds a string w in the dictionary such that wx is not in the dictionary, it should output the codeword for w and then add wx to the dictionary. Initially, the codeword for each string will be 1 bit long. Whenever the number of strings in the dictionary exceeds the number of codewords of the current length, the codeword length increases by 1 bit. For example, the first 2 codewords will be 1 bit long, the next 2 codewords will be 2 bits long, the next 4 codewords will be 3 bits long, and so on. Your encoder should write its output to a file, and your decoder should read its input from the file written by the encoder.]

4. [Several of the algorithms described in this chapter can be improved by using a variable-length encoding for the codewords. For example, the encoder for the LZW algorithm described in this section outputs a sequence of 12-bit codewords. Initially, these codewords are the integers from 0 to 255, which are used to encode the 256 single-character strings. Each time a new string is added to the dictionary, it is assigned the next available codeword. Thus, the first string added to the dictionary is assigned codeword 256, the next string is assigned codeword 257, and so on. The problem with this approach is that it is wasteful to use 12 bits to encode a character or a string that occurs frequently. For example, if the input is a text file containing only the 26 uppercase letters, then only 5 bits are needed to encode each character. Thus, when encoding this type of input, the first codeword that is more than 5 bits long should be codeword 32. Similarly, when codeword 64 is added to the dictionary, the codewords should be increased to 7 bits, and when codeword 128 is added to the dictionary, the codewords should be increased to 8 bits. Implement this variable-length encoding idea. Your encoder should write its output to a file, and your decoder should read its input from the file written by the encoder.]

FIGURE 11.14 A function representing spike frequency



...as per our discussion, we'll choose the code for 1 to be a single bit, while the code for 0 will be two bits. Now we can encode the word as 1 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 0 0

Lynch, Thomas J., Data Compression: Techniques and Applications, Lifetime Learning Publications, Belmont, CA, 1985.

Robinson, Tony, "Results of a Prototype Television Bandwidth Compression Scheme," Proc. IEEE, Vol. 55, No. 3, March 1967, pp. 356–364.

Ruth, Gregg A., and Peter J. Fleming, "Image Segmentation as an Approach to Data Compression," Communications of the ACM, Vol. 31, No. 8, August 1988, pp. 960–975.

Schalkoff, Robert J., Digital Image Processing and Computer Vision, John Wiley & Sons, New York, 1989, Chapter 8.

Huffman Coding

Gallager, Robert G., "Variations on a Theme by Huffman," IEEE Transactions on Information Theory, Vol. IT-24, No. 6, November 1978, pp. 668–674.

Knuth, Donald E., "Dynamic Huffman Coding," Journal of Algorithms, Vol. 6, No. 2, June 1985, pp. 163–180.

Vitter, Jeffrey S., "Algorithm 673: Dynamic Huffman Coding," ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989, pp. 158–167.

Run-Length Encoding

Capon, John, "A Probabilistic Model for Run-Length Coding of Pictures," IRE Transactions on Information Theory, Vol. IT-5, No. 4, December 1959, pp. 157–163.

Arithmetic Coding

Langdon, Glen G., Jr., "An Introduction to Arithmetic Coding," IBM Journal of Research and Development, Vol. 28, No. 2, March 1984, pp. 135–149.

Rissanen, Jorma, and Glen G. Langdon, Jr., "Arithmetic Coding," IBM Journal of Research and Development, Vol. 23, No. 2, March 1979, pp. 149–162.

Witten, Ian H., Radford M. Neal, and John G. Cleary, "Arithmetic Coding for Data Compression," Communications of the ACM, Vol. 30, No. 6, June 1987, pp. 520–540.

Dictionary Techniques

Ziv, Jacob, and Abraham Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, Vol. IT-23, No. 3, May 1977, pp. 337–343.

Ziv, Jacob, and Abraham Lempel, "Compression of Individual Sequences via Variable-Rate Coding," IEEE Transactions on Information Theory, Vol. IT-24, No. 5, September 1978, pp. 530–536.

Welch, Terry A., "A Technique for High-Performance Data Compression," Computer, Vol. 17, No. 6, June 1984, pp. 8–19.

Memory Management

The preceding chapters rarely looked behind the scenes to the how programs are actually executed and how variables of different types are stored. The reason is that this book emphasizes data structures rather than the inner workings of the computer. This latter belongs more to a book about operating systems or assembly language programming than to a discussion of data structures.

In a few instances, such a reference was unavoidable, namely when discussing recursion in Chapter 5. There recursion was explained in terms of the run time stack and how a computer actually works. We also alluded to this level when discussing dynamic memory allocation. It is hard to discuss dynamic memory allocation without some awareness of the structure of computer memory and the realization that, without one, pointers may contain invalid memory locations. In addition, the user should be careful to avoid exhausting the computer's memory resources. Managing memory is often the responsibility of the programmer, and memory leaks can be dragged with unreachable locations which were not dealt with during the program's execution by careful. The most efficient and appropriate data structure may not be apparent from a problem's description.

This section is the operation of a computer's memory from which portions of memory are eventually allocated at the request of a program. This heap has no logical relationship to the specific data structure called a heap or binomial heap. In languages such as FORTRAN, COBOL, or BASIC, the compiler determines how much memory is required to run a program. In languages that allow dynamic memory allocation, the amount of memory required cannot always be determined prior to the program run. To that end, the heap is used. Each program requests memory by asking the operating system for a block of memory. When the program is finished with the memory, it returns the memory to the heap. Thus, the heap is the pool of memory from which memory is allocated from the program to which memory is returned. Also, in most languages, once memory has been explicitly released by the

programmer through a language primitive or function call (such as `free` in C and `dispose` in Pascal), or the memory might be automatically reclaimed by the operating system. Automatic storage reclamation is a boon to many programmers because it relieves them of the burden of writing memory-management routines. It is usually a part of functional and logic programming languages and most object-oriented languages. It may also be useful for some procedural languages, such as spreadsheets and report generators.

The main issue in managing memory is keeping track of the memory blocks to be used by programs. If memory can be allocated from anywhere in memory, then all of memory is potentially available for use by the next program that requests memory. Memory is usually divided into several partitions, one for each program. The partition for the operating system may be in low memory, the transient program area may be in high memory, and user programs may be in between. The operating system keeps track of memary partitions with data structures that record which memory addresses are assigned to which programs. When a program requests more memory, the memory manager searches for available memory not assigned to any program. Typically, the memory manager can then assign additional memory addresses to the program and record the assignment in its data structures. When memory is returned, the memory manager updates its data structures to indicate that the memory is available.

Memory can be allocated from anywhere in memory, but programs usually require memory to be contiguous. Memory that is contiguous is a single block of memory without any gaps. Memory can be allocated to a program as a single block or as a collection of smaller blocks. Memory can be allocated to a program when the program is loaded, or memory can be allocated to a program as it executes. Memory can be allocated in fixed-size blocks or in variable-size blocks. Memory can be allocated from a single free memory pool or from multiple memory pools. Memory can be allocated from the operating system or from a program's own memory space. Memory can be allocated automatically or explicitly. Memory can be allocated by the compiler or by the programmer. Memory can be allocated statically or dynamically. Memory can be allocated globally or locally. Memory can be allocated contiguously or non-contiguously. Memory can be allocated in a single step or in multiple steps. Memory can be allocated in a single pass or in multiple passes. Memory can be allocated in a single thread or in multiple threads. Memory can be allocated in a single process or in multiple processes. Memory can be allocated in a single machine or in multiple machines. Memory can be allocated in a single network or in multiple networks. Memory can be allocated in a single system or in multiple systems. Memory can be allocated in a single environment or in multiple environments. Memory can be allocated in a single context or in multiple contexts. Memory can be allocated in a single scope or in multiple scopes. Memory can be allocated in a single lifetime or in multiple lifetimes. Memory can be allocated in a single duration or in multiple durations. Memory can be allocated in a single extent or in multiple extents. Memory can be allocated in a single range or in multiple ranges. Memory can be allocated in a single set or in multiple sets. Memory can be allocated in a single collection or in multiple collections. Memory can be allocated in a single group or in multiple groups. Memory can be allocated in a single class or in multiple classes. Memory can be allocated in a single type or in multiple types. Memory can be allocated in a single kind or in multiple kinds. Memory can be allocated in a single sort or in multiple sorts. Memory can be allocated in a single order or in multiple orders. Memory can be allocated in a single sequence or in multiple sequences. Memory can be allocated in a single series or in multiple series. Memory can be allocated in a single chain or in multiple chains. Memory can be allocated in a single link or in multiple links. Memory can be allocated in a single node or in multiple nodes. Memory can be allocated in a single tree or in multiple trees. Memory can be allocated in a single graph or in multiple graphs. Memory can be allocated in a single network or in multiple networks. Memory can be allocated in a single system or in multiple systems. Memory can be allocated in a single environment or in multiple environments. Memory can be allocated in a single context or in multiple contexts. Memory can be allocated in a single scope or in multiple scopes. Memory can be allocated in a single lifetime or in multiple lifetimes. Memory can be allocated in a single duration or in multiple durations. Memory can be allocated in a single extent or in multiple extents. Memory can be allocated in a single range or in multiple ranges. Memory can be allocated in a single set or in multiple sets. Memory can be allocated in a single collection or in multiple collections. Memory can be allocated in a single group or in multiple groups. Memory can be allocated in a single class or in multiple classes. Memory can be allocated in a single type or in multiple types. Memory can be allocated in a single kind or in multiple kinds. Memory can be allocated in a single sort or in multiple sorts. Memory can be allocated in a single order or in multiple orders. Memory can be allocated in a single sequence or in multiple sequences. Memory can be allocated in a single series or in multiple series. Memory can be allocated in a single chain or in multiple chains. Memory can be allocated in a single link or in multiple links. Memory can be allocated in a single node or in multiple nodes. Memory can be allocated in a single tree or in multiple trees. Memory can be allocated in a single graph or in multiple graphs.

12.1 The Sequential-Fit Methods

A simple organization of memory could require a linked list of all memory blocks which is updated after a block is either requested or returned. The blocks on such linked lists can be organized in a variety of ways, according to the block sizes or the block addresses. Whenever a block is requested, a decision has to be made concerning which block to allocate and how to treat the portion of the block exceeding the requested size.

For reasons of efficiency, doubly linked lists of blocks are maintained, with links residing in the blocks. Each available block of memory uses a portion of itself for two links. Also, both available and reserved blocks have two fields to indicate their status (available or reserved) and their size.

In the sequential-fit methods, all available memory blocks are linked together, and the list is searched to find a block whose size is larger than or the same as the

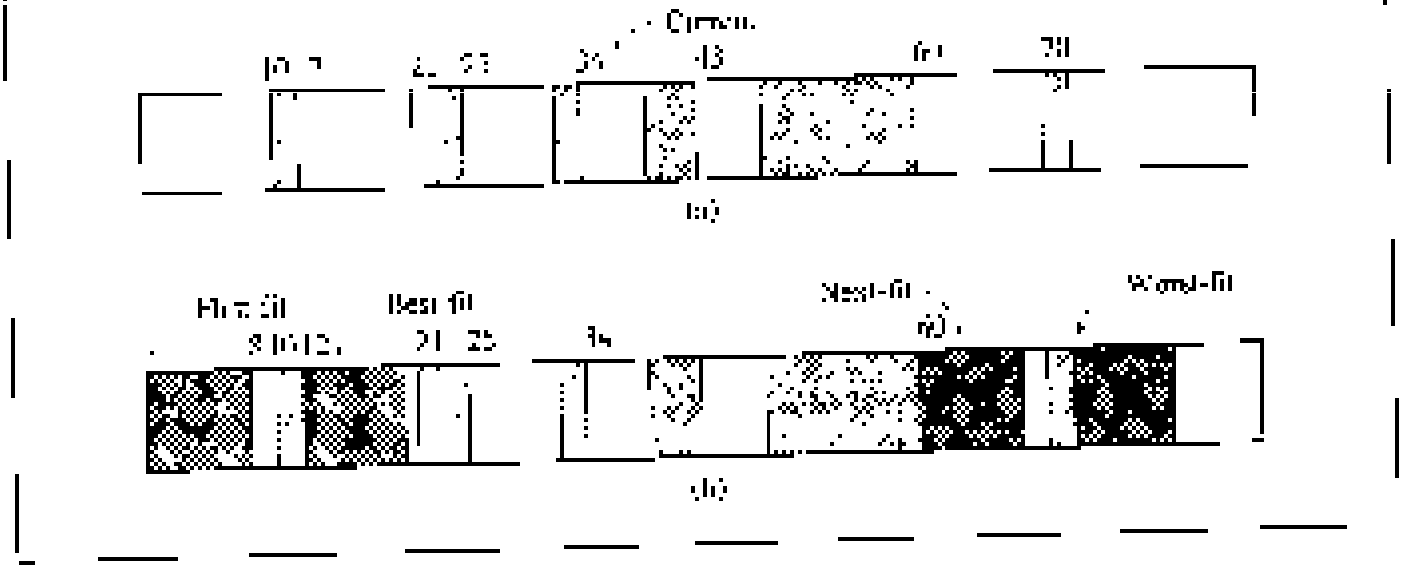


FIGURE 12.1 Memory allocation using sequential-fit methods.

requested size. A simple policy for handling requests of blocks of memory is to coalesce them with neighboring blocks and reflect this fact by properly adjusting the links in the linked list.

The order of searching the list for such a block determines the division of these methods into several categories. The first-fit algorithm allocates the first block of memory large enough to meet the request. The best-fit algorithm allocates a block which is closest in size to the request. The worst-fit method finds the largest block on the list and then allocates a portion equal to the request size. The remaining part is large enough to be used in later requests. The next-fit method allocates the next available block that is sufficiently large.

Figure 12.1a depicts a memory configuration after several requests and releases of memory blocks. Figure 12.1b illustrates which portion of memory would be allocated by which sequential-fit method to satisfy a request for 83B of memory.

The most efficient method is the first-fit procedure. The next-fit method is of comparable speed but causes more excessive memory fragmentation because it scans the list to blocks starting from the current position and reaches the end of the list much earlier than the first-fit method. But the best-fit algorithm is even worse in this respect since it searches for the closest match with respect to size. The pieces of blocks remaining after returning the requested size are small and practically unusable. The worst-fit algorithm attempts to prevent this type of fragmentation by avoiding, at least, creating the creation of small blocks.

The way in which blocks are organized on the list determines how fast the search for an available block succeeds or fails. For example, to implement the best-fit and the worst-fit methods, the blocks should be arranged by size. For other methods, the address order may be adequate.

12.2 The Nonsequential-Fit Methods

The sequential fit methods being what they are, they may become inefficient for large memory. In the case of large memory, a nonsequential search is desirable. One strategy divides memory into an arbitrary number of lists, each list holding blocks of the same size (Ross 1967). Larger blocks are split into smaller blocks to satisfy requests, and new lists may be created. Since the number of such lists can become large, they can be organized as a tree.

Another approach is based on the observation that the number of sizes requested by a program is limited, although the sizes may differ from one program to another. Therefore, the lists of blocks of different sizes can be kept short if it can be determined which sizes are the most popular. This leads to an *adaptive exact-fit* technique that dynamically creates and adjusts storage block lists which fit the requests exactly (Oldehoeft and Allan 1985).

In adaptive exact-fit, a size-list of block lists of a particular size returned to the memory pool during the last T allocations is maintained. A block b is added to a particular block list if this block list holds blocks of b's size and b has been returned by the program. When a request comes for a block of k bytes, a block from its block list is dispatched to meet the request. Otherwise, a more time-consuming search for a block in memory is triggered using one of the sequential-fit methods.

The exact fit method disposes of some block lists if no request comes for a block from this list in the last T allocations. In this way, lists of infrequently used block sizes are not maintained, and the list of block lists is kept small to allow a sequential search of this list. Because it is not a sequential search of the memory, the exact fit method is not considered a sequential-fit method.

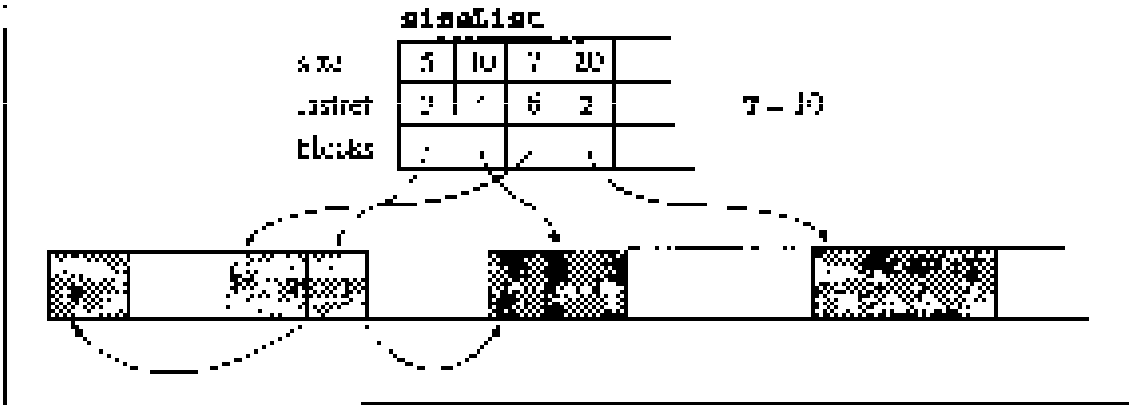
Figure 12.3 contains an example of a size-list and a heap created using the adaptive exact fit method. The memory is fragmented, but if a request comes for a block of size k, the allocation can be done immediately, since the size list has an entry for size k; thus, memory does not have to be searched. A simple algorithm for allocating blocks is as follows:

```
L = 0;
allocate (reqSize):
    L++;
    if a block list bl with negative blocks now allocated:
        listArray[bl] = -1;
        b = head of linked list;
        if b was the only block accessible from bl
            detach bl from sizeList;
    else b = search-memory-for-block-of(reqSize);
    dispose of all block lists on sizeList for which L - lastUsed[bl] > T;
    return b;
```

A procedure for returning blocks is symmetrical.

This algorithm might have to be tuned to the specific memory usage. The algorithm must be expanded to deal with this problem successfully. One solution is to vary

Figure 12.2 An example: configuration of a size-list and heap created by the adaptive-exact fit method.



fragmentation in computer memory after a certain number of allocations and deallocations. A noncompacting approach may consist in liquidating the size-list and building it anew after some predetermined period. The authors of this method claim that fragmentation problems "failed to materialize," but that can be attributed to the configurations of their users. Such problems certainly materialize in sequential-fit methods and in another nonsequential fit strategy to be discussed in the next section.

12.2.1 Buddy Systems

Nonsequential memory management methods known as *buddy systems* do not just assign memory in sequential slices, but divide it into two buddies which are merged whenever possible. In the buddy system, two buddies are never free. A block can have either a buddy used by the program or none.

The classic buddy system is the *binary buddy system* [Knowlton 1965]. The binary buddy system assumes that storage consists of 2^m locations for some integer m with addresses $0, \ldots, 2^m - 1$, and that these locations can be organized into blocks whose lengths can only be powers of 2. There is also an array `avail[]` such that, for each $i = 0, \ldots, m$, `avail[i]` is the head of a doubly linked list of blocks of the same size 2^i.

The name of this method is derived from the fact that every block of memory (except the entire memory) is coupled with a buddy of the same size which participates with the block in reserving and returning chunks of memory. The buddy of a block of length 2^i is determined by complementing bit i in the address of this block. This is strictly related to the lengths of blocks, which can only be powers of 2. In particular, all blocks of size 2^i have 0's in the i rightmost positions and differ only in the remaining bits. For example, if memory has only eight locations, then the possible addresses of blocks of size one are {001, 011, 010, 011, 100, 101, 110, 111}, addresses of blocks of size two are {000, 010, 100, 110}, of size four {000, 100}, and of size eight {000}. Note that in the second set of addresses the last bit is 0 and the addresses refer to blocks of size 2. The addresses in the third set have zeros ending us, since the size of the blocks is 4. Now, for the second set, there are two pairs of blocks and their buddies: {(000, 010), (100,

1100[]; for the third set, there is only one such, 1000 1000. Hence, the difference between the address of a block of size 2^i and the address of its buddy is only in bit $i+1$.

If a request arrives to allocate a new memory block of size k, then the buddy system returns a free memory block whose size is greater than or equal to k. Since there are many candidates for such blocks, the list of such blocks is checked in `avail[]` whose size is the smallest among all the sizes in the list of blocks can be found in location `avail[m]`. If the list is empty, then the next list of blocks is checked in position `m+1`; then in position `m+2`, and so on. The search continues until a nonempty list is found (or the end of `avail[]` is reached), and then a block is selected from it.

The skeleton for memory allocation in a binary buddy system is as follows:

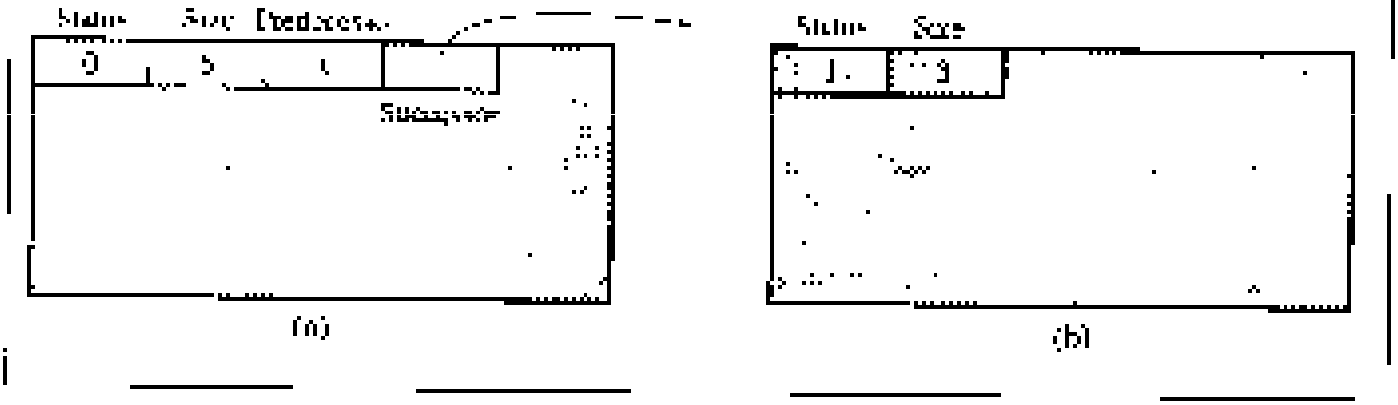
```
sizeofmemory = 2^m for some m;
avail[i] = - for i = 0, . . . , m-1;
avail[i] = (location of) memory;
reserve(regSize)
   roundedSize = lg(regSize)
   availSize = min(roundedSize, . . . , m) for which avail[availSize] > -1
   if no such availSize exist
      failure;
   block = avail[availSize];
   detach block from list avail[availSize];
   while (roundedSize < availSize)   // while an available block
      availSize--;                   // is too large - split it;
      block = left half of block;
      insert buddy of block in list avail[availSize];
   return block;
```

Each free block of the buddy system should include four fields indicating its status, its size, and its two neighbors in the list. On the other hand, reserved blocks include only a status field and a size field. Figure 12.3a illustrates the structure of a free block in the buddy system. The block is marked as free with the status field set to 0. The size is specified as 2^5 locations. No predecessor is specified, so this block is pointed to by `avail[5]`. The size of its successor is also 2^5 locations. Figure 12.3b illustrates a reserved block whose status field is set to 1.

Figure 12.4 contains an example of reserving three blocks, assuming that the memory in use is of size $2^7 = 128$ locations. First, the entire memory is free (Figure 12.4a). Then, 18 locations are requested, so `roundedSize` = lg(18) = 5. But `avail Size` = 7, so the memory is split into two buddies, each of size 2^6. The second buddy is marked as available by setting the status field and including it in the list, `avail[6]` (Figure 12.4b). `availSize` is still greater than `roundedSize`, so another iteration of the `while` loop of `reserve()` is executed. The first block is split into two and the second buddy is included in the list `avail[5]` (Figure 12.4c). The first buddy is marked as reserved and returned to the caller of `reserve()` for use. Note that only a portion of the returned block is really needed. However, the entire block is marked as reserved.

Next, a block of 14 locations is requested; now, `roundedSize` = lg(14) = 4, `availSize` = 5, and the block pointed to by `avail[5]` is claimed (Figure 12.4d).

FIGURE 12.3 Block structure in the binary buddy system.



This block is too large since roundedBlockSize < size, so the block is divided into two buddies. The first buddy is marked as reserved and returned, and the second is included in a list (Figure 12.6e). Finally, a block of 16 locations is requested. After two iterations of the while loop of getBlock, the configuration pictured in Figure 12.6g emerges: there are two available blocks of 16 locations, and both are linked up to the list avail[4].

To be sure, blocks of memory are not only claimed, but they are returned, too. They have to be included in the pool of available blocks. Before they are included, the status of each block's buddy is checked. If the buddy is available, the block is combined with its buddy to create a block twice as large as before the combination. If the buddy of the new block is available, it is also combined with its buddy, resulting in a still larger block of memory. This process continues until the entire memory is combined into one block or a buddy is not available. This coalescing creates blocks of available memory as large as possible. The algorithm of including a block in the pool of available blocks is as follows:

```
includeBlock(block)
   blockSize = size(block);
   buddy = address(block) with(2 ** blockSize+1 bit is complement);
   while status(buddy) is 0                    // buddy has not
         and size(buddy) == blockSize          // been claimed
         and blockSize <= (size of memory)     // buddy exists;
      detach buddy from list avail[blockSize];
      block = block plus buddy;  // coalesce block and its buddy;
      address(block) to 0;
      blockSize++;
      buddy = address(new enlarged block) with(2 ** blockSize+1 bit is complement);
   include block in list avail[blockSize];
```

Figure 12.5 illustrates this process. A block previously claimed is now released (Figure 12.5a) and, because the buddy of this block is free, it is combined with the

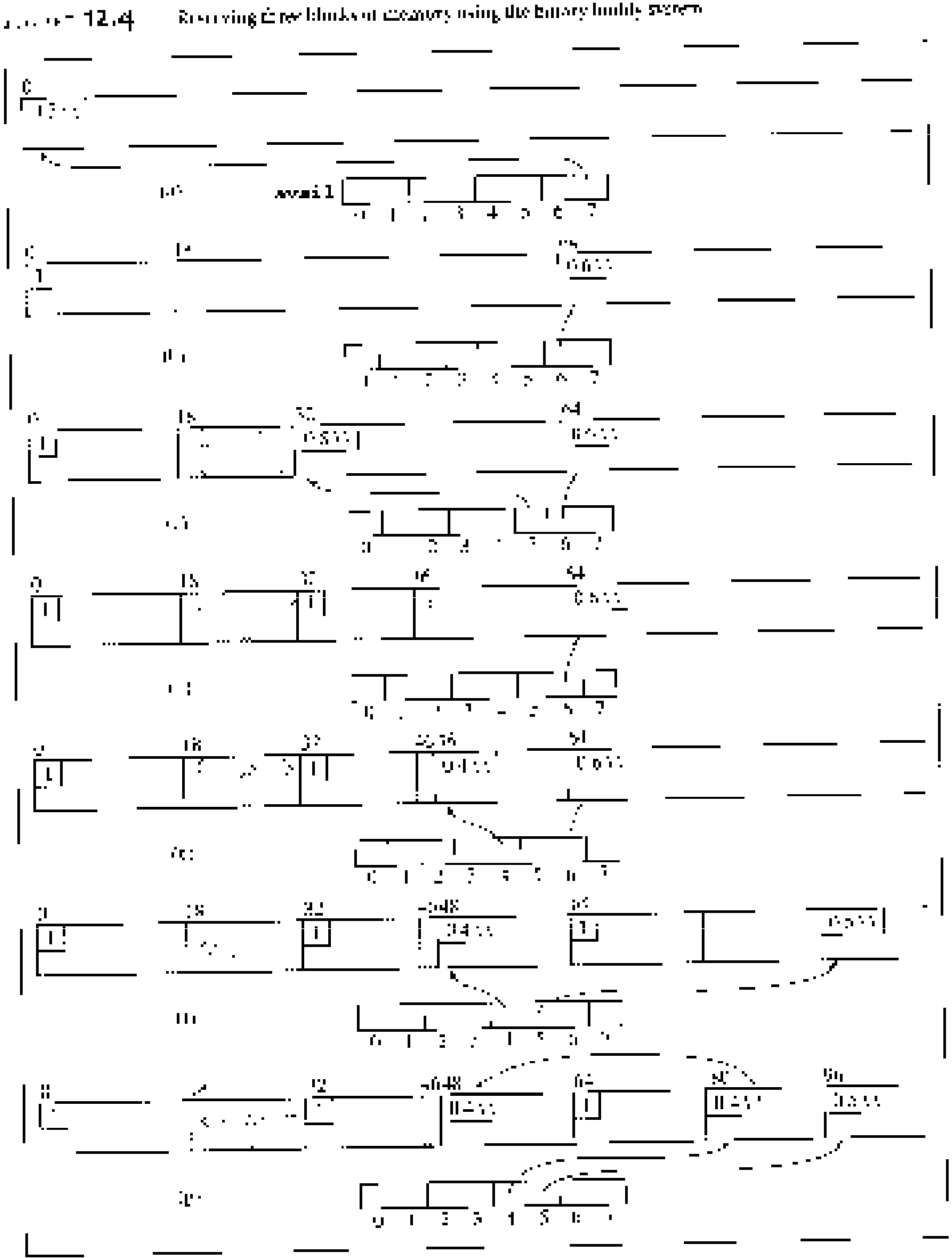
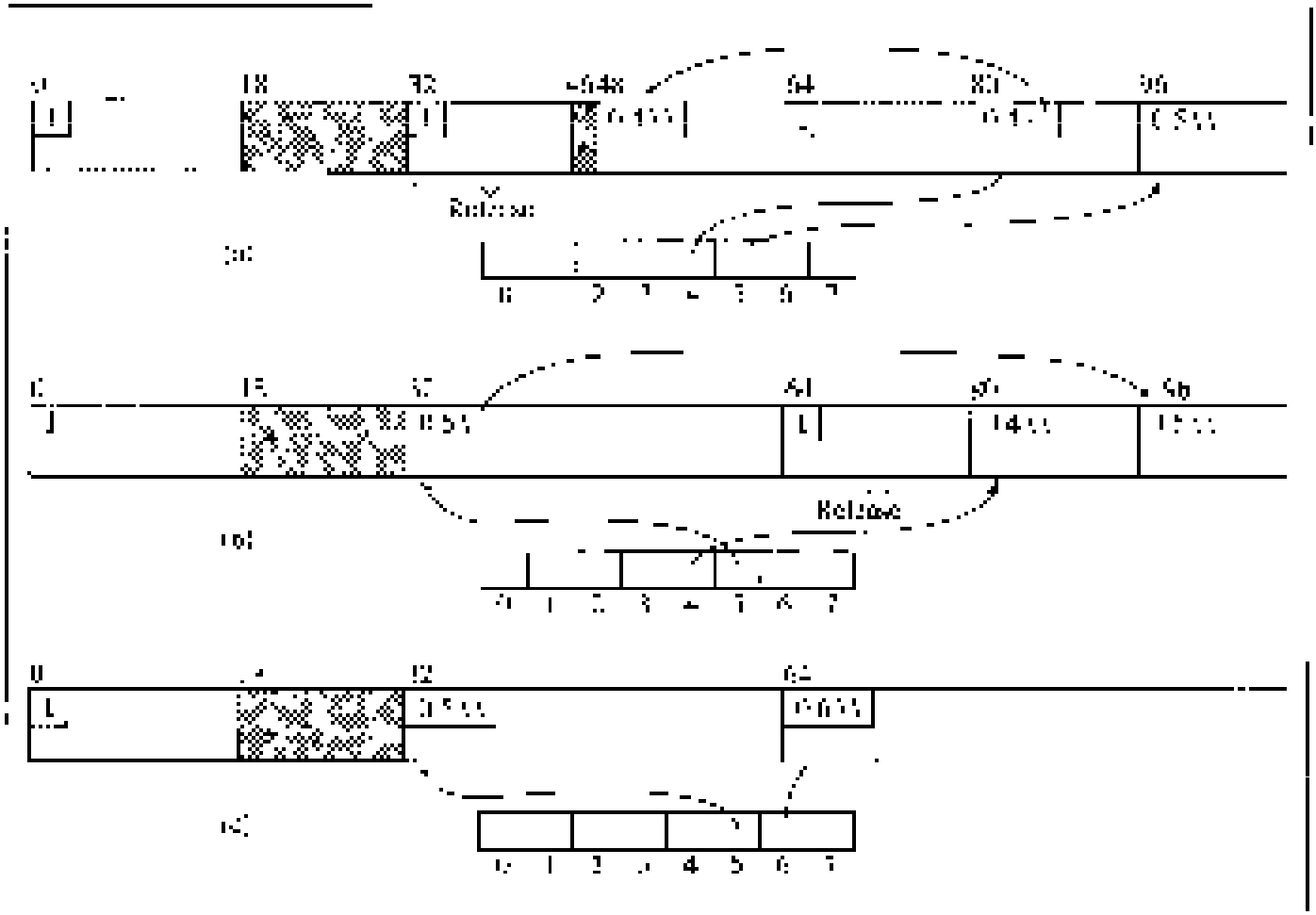


FIGURE 12.5 (a) Returning a block to the pool of blocks (b) resulting in a coalescing of a block with its buddy (c) Returning another block leads to two coalescings



blocks resulting in double-sized blocks, which is included in the list of size 1 (see Figure 12.5b). Releasing another block allows the memory manager to combine the block with its buddy and the resulting block with its buddy (Figure 12.5c). Note that the first portion of the leftmost block (marked with the darker screen) did not participate in this coalescing process and is still considered occupied. Also, the two rightmost blocks in Figure 12.5c, although adjacent, were not combined because they are not buddies. Buddies in the binary buddy method have to be of the same size.

The binary buddy system, although extremely efficient in terms of speed, may be inefficient in terms of space. Figure 12.4d shows that the leftmost block is assigned a size of 48 locations, but only 32 of them are in use, since the user really needs 18 + 14 locations. This means that almost half of these two blocks is wasted. The waste is even worse if the number of locations requested is equal to only more than a power of 2. In the extreme, approximately 50% of memory is wasted. This problem with internal fragmentation that results from trying to round all requests to the nearest larger power of 2.

Also, there may be a problem with external fragmentation as well. A request may be refused although the pool of empty blocks is sufficient to satisfy it, but separated in

the configuration of memory in Figure 12.4(a), a request for 50 locations is refused because there is no block available with a size of 64 locations or more. A request for 33 locations is treated similarly and for the same reason, although there are 33 available consecutive locations. But one of these locations belongs to another block, which prevents it from being used.

These problems are brought about by the fact that the binary buddy system uses a simple subdivision of blocks into two even parts, which results in the division of memory not sufficiently tuned to incoming requests. The sequence of block sizes possible in this system is $2^0, 2^1, 2^2, 2^3, \ldots, 2^m$. An improvement on the binary buddy system can be obtained if the sequence is rendered by the recurrence equation

$$s_i = \begin{cases} 1 & \text{if } i = 0 \\ s_{i-1} + s_{i-2} & \text{otherwise} \end{cases}$$

which can be considered a particular case of a more general equation:

$$s_i = \begin{cases} s_1 & \text{if } i = 1, \\ \vdots & \vdots \\ s_k & \text{if } i = k \\ \vdots & \\ s_{i-1} + s_{i-k} & \text{otherwise} \end{cases}$$

If $k = 1$, then this equation reduces to the equation for the binary buddy system. If $k = 2$, then the above formula is a very familiar equation for a Fibonacci sequence:

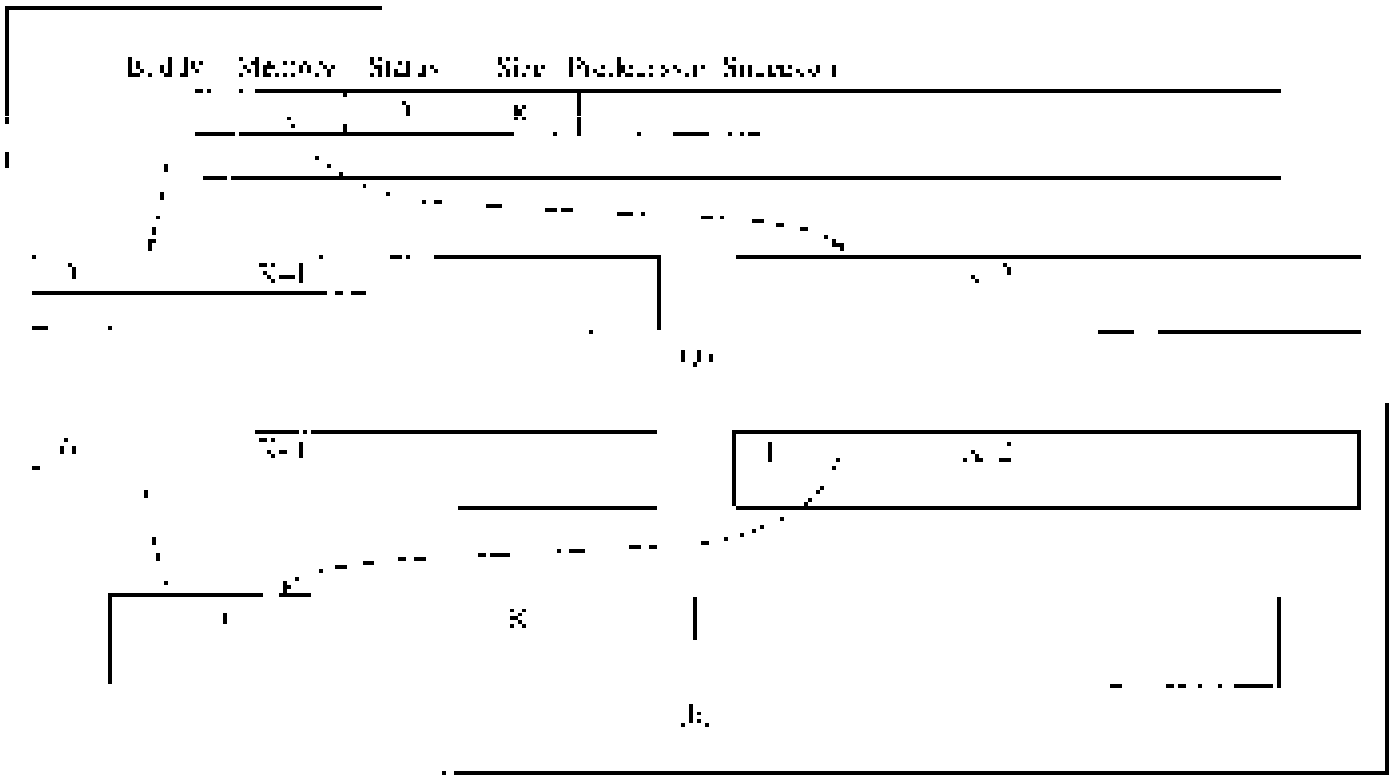
$$s_i = \begin{cases} 1 & \text{if } i = 0, 1 \\ s_{i-1} + s_{i-2} & \text{otherwise} \end{cases}$$

This leads to the Fibonacci buddy system, developed by Daniel S. Hirschberg. The cases s_0 and s_1 are the values for s_0 and s_1. If $k > 2$, then we refer to the series of the generalized Fibonacci system (Hirschberg 1973).

The problem with the Fibonacci buddy system is that finding a buddy of a block is not always simple. In the binary buddy system, the information stored in the size field of the block is sufficient to compute the address of the buddy. If the size holds the number n, then the address of the buddy is found by complementing the n^{th} bit in the address of the block. This works regardless of whether the block has a right buddy or a left buddy. The reason for this simplicity is that only powers of 2 for the sizes of all blocks are used, and each block and its buddy are of the same size.

In the Fibonacci system, this approach is complicated, yet it is necessary to know whether a returned block has a right or a left buddy in order to coalesce the two. Not surprisingly, finding the buddy of a block may be rather demanding in terms of time or space. To this end, Hirschberg used a table which could have nearly 1000 entries if buffers of up to 17,711 locations are allowed. The method can be simplified if an improper flag is included in each block, but a binary (left/right) flag may be insufficient. If block b_1 marked as left is coalesced with its buddy block b_2, then the question is: How do you find the buddy of the resulting block, b_3? An elegant solution uses two binary flags instead of one. A buddy-bit is maintained as before, and I bit uses LRTY. If a block b_1 is split into blocks $b_{1,0}$ and $b_{1,1}$, then initially $b_{1,0}.LRTY = 0$, $b_{1,0}.buddy\text{-}bit = 1$, $b_{1,1}.LRTY = 1$, and $b_{1,1}.buddy\text{-}bit = b_1.buddy\text{-}bit$. If $b_{1,0}$ is merged with $b_{1,1}$, then $b_{1,0}.buddy\text{-}bit = b_{1,1}.buddy\text{-}bit$.

FIGURE 12.6 (a) Splitting a block of size 8K to create two buddies using the buddy bit and the memory bit. (b) Coalescing two buddies obtaining information stored in buddy and memory bits.



buddy[ ] (see Figure 12.6a). In the two-dimensional array, pred denotes predecessor. Memory-bit[b_{pq}] indicates whether the parent is split or not, and memory-bit[b_{pqr}] is a bit of information indicating the present status (or one of the predecessors) of b_{pq}'s parent. Note that memory-bit[ ] provides information similar to split[ ] used in the buddy system algorithm.

The algorithms for reserving blocks and for returning them are essentially the same as the algorithms used for the binary buddy system. An algorithm for reserving a block is as follows:

```
avail[i] = -1 for i = 0, 1, ..., m-1;
avail[i] = first block of size 2^i;

reserve(int request);
   available = the position of the first block on the linked list pointed to by
               pointer avail[available] > -1;
   if no such block exists then
      failure;
   status = available block;
   remove the block pointed to by avail[available];
   while the current size is > request // while an available block is
                                       // too large  split it: choose
      the left buddy; available--; // smaller of the buddies if it's
```

```
        return block;
    }
}
```

Another scheme that uses multiple page sizes is the one employed by the Stanford THE multiprogramming system (Bron and Fenwick 1977). It is possible in the operating system to determine the amount of internal fragmentation (by allowing programs to terminate only at certain points rather than anywhere in their last page). Based on this information, the operating system can redefine the page size to reduce fragmentation. Frequently used programs may even be given smaller page sizes to reduce their fragmentation. Of course, frequent redefinition of page size requires changing all memory-management information kept in terms of the old page size. The THE system was designed to help solve this problem for a relatively small number of page sizes.

The Multics system (which ran on Honeywell 6180 computers) combined paging and segmentation (Daley and Dennis 1968). Recall that in this system, segments are multiples of the page size. Therefore, a segment will usually end in a partially empty page (on average, half a page). Multics therefore allows a segment to have pages of several sizes. The Multics operating system is coded in PL/I, a language that requires all segments to be a multiple of the page size. Therefore, the PL/I compiler must round segments up to the next page boundary. The amount of padding added to segments can be controlled by the programmer, so that she can reduce internal fragmentation if memory space is at a premium.

Another scheme that uses multiple page sizes is the one employed by the Stanford THE multiprogramming system (Bron and Fenwick 1977). It is possible in the operating system to determine the amount of internal fragmentation (by allowing programs to terminate only at certain points rather than anywhere in their last page). Based on this information, the operating system can redefine the page size to reduce fragmentation. Frequently used programs may even be given smaller page sizes to reduce their fragmentation. Of course, frequent redefinition of page size requires changing all memory-management information kept in terms of the old page size. The THE system was designed to help solve this problem for a relatively small number of page sizes.

The Multics system (which ran on Honeywell 6180 computers) combined paging and segmentation (Daley and Dennis 1968). Recall that in this system, segments are multiples of the page size. Therefore, a segment will usually end in a partially empty page (on average, half a page). Multics therefore allows a segment to have pages of several sizes. The Multics operating system is coded in PL/I, a language that requires all segments to be a multiple of the page size. Therefore, the PL/I compiler must round segments up to the next page boundary. The amount of padding added to segments can be controlled by the programmer, so that she can reduce internal fragmentation if memory space is at a premium.

The concept of a segment is useful to the programmer, but it is frequently unknown to the operating system. Paging, on the other hand, is used to provide as many (possibly small) pages as possible for efficient memory allocation. Combining these two ideas is the segmented-page architecture that is, for example, implemented in the Intel 386 and 486 processors. Segments are allocated in pages, but the memory-management unit can be programmed to recognize the segment boundaries. Thus, small segments can share a page, and a segment can be larger than a page. The memory-management unit, which is the interface between the CPU and the memory bus, is responsible for translating the segmented addresses generated by the CPU into paged addresses. Because the memory-management unit is controlled by the operating system, the operating system can define the segments and pages and can control the memory-management unit.

12.3 Garbage Collection

As mentioned at the beginning of this chapter, some languages have automatic storage reclamation in their environment so that an explicit return of unused memory cells must be done by the program. The program can allocate memory through the

function runs, but there is no need to return back to the operating system the allocated memory block if the block is not needed any longer. The block is simply abandoned, and it will be reclaimed by a method called a garbage collector that is automatically invoked to collect unused memory cells when the program is idle or when memory resources are exhausted.

The garbage collector views the heap as a collection of memory cells, or nodes, each cell composed of several fields. Depending on the garbage collector, the fields can be different. For example, in LISP a cell has two pointers, usually the car and cdr in LISP terminology (or head and tail in other cells, except for atomic cells that have no pointers. The cells include headers with such elements as an atom/nonatom flag and a mark/unmarked flag. Data that are included may be used in your another field of a cell or in the part of memory cells used for pointers in nonatomic cells. Moreover, if variable-sized data are used, the header includes the number of bytes in the data field. Each memory cell has pointer fields as well as possible. Pointers to all linked structures currently used by the program are stored in a root set, which contains all pointer variables. The garbage collector's task is to determine those parts of memory which are accessible from any of these pointers and parts which are not currently in use and can be returned to the free memory pool.

Garbage collection algorithms usually include two phases, which may be executed one after the other, or the two phases can be integrated:

1. The marking phase—in which all currently used cells
2. The reclamation phase—where all unmarked cells are returned to the memory pool; this phase can also include heap compaction.

12.3.1 Mark-and-Sweep

A classical method of collecting garbage is the mark-and-sweep technique, which clearly distinguishes the two phases. First, memory cells currently in use are marked by traversing each linked structure, and then the memory is swept to glean all unmarked cells that are then returned to the memory pool.

Marking

A simple marking procedure is easy to write like a tree traversal. The code is not optimal, but it is short, and it is not necessary to make changes to the node at the top of the stack.

```
mark(node)
    if node is not marked
        mark node;
        if node is not an atom
            mark(node.left node());
            mark(node.right node());
```

This procedure is called for each element of the pointer variables. The problem with this recursive algorithm is that it may cause a run-time stack overflow,

which is a very real prospect considering the fact that the list being marked can be very long. Therefore, an explicit stack can be used so that there is no need to store on the run-time stack the data necessary to properly resume execution after returning from recursive calls. Here is an example of an algorithm that uses an explicit stack:

```
markUsingAnExplicitStack (node):
    push(node);
    while stack is not empty
        node = pop();
        while node is not a marked location
            mark node;
            push(stack,node);
            node = findLink(node);
        // node is a marked atom
        mark node;
```

The problem of an overflow is not avoided altogether if the stack is an array, but the array may be made to be very large. If it is implemented as a linked list, it may be impossible to use, since the stack requires memory resources which have not been used up until the point at which the stack was required to perform this. There are two ways to avoid this problem: one is by using a stack of fixed size and invoking a garbage collection when the stack overflows; the other is by using the memory being collected itself.

A type of algorithm that requires no explicit stack was developed by Schorr and Waite [Schorr67] for use in systems that incorporate the stack into the list being processed. The technique belongs to the same category as the pointer reversal techniques discussed in Section 6.4.3. In the Schorr and Waite marking method, some links are temporarily reversed during the processing of the list to "remember" the path back to the node from which we originally started after marking all nodes accessible from a position in which the traversal has now terminated. When a marked node or an atom is encountered, the algorithm reverts to processing nodes. However, it is necessary to mark through the branch of the graph that is to be traversed, and the algorithm has to use a marker to indicate whether both branches of such a node or only the left path has been checked. To this end, the algorithm uses one additional bit called a tag bit. If the left of a cell is traversed, the tag bit remains zero so that, upon return to this cell, the path accessible from left will be followed, in which case the tag bit is set to one and reset to zero upon return. The summary of the algorithm is as follows:

```
reverse(link p1, p2, p3)
    tmp = p1;
    p3 = p1;
    p1 = p2;
    p2 = tmp;

SWmarking(root)
    prev = null;
    while (1)
        next node;
```

```
    if head(curr) is marked or atom
        if head(curr) is an unmarked atom
            mark head(curr);
        while snd(curr) is an unmarked atom
            if head(curr) is an unmarked atom
                mark head(curr);
            while prev is not null and tag(prev) is 1 // go back
                aux(prev) = tl;
                invertLink(curr, prev, tl(prev));
            if prev is not null
                invertLink(curr, prev, hd(prev));
            else finished;
        aux(curr) = 1;
        invertLink(prev, curr, tl(curr));
    else invertLink(prev, curr, hd(curr));
```

Figure 12.7 illustrates an example. Each part of this figure shows changes in the list after the indicated operations have been performed. Note that atom nodes do not require a tag bit. Figure 12.7a contains the list before marking. Each nonatomic node has two separate marking bits, a tag bit, and mark and aux fields. The marking and tag bits are initialized to 0. There is one more bit not shown in this figure—an interruption flag.

Here is a description of each iteration of the outside loop and the figure number which contains the structure of the list structure at that point.

Iteration 1: Execute `invertLink(prev, curr, hd(curr))` (Figure 12.7a).

Iteration 2: Execute another `invertLink(prev, curr, hd(curr))` (Figure 12.7c).

Iteration 3: Execute still another `invertLink(prev, curr, hd(curr))` (Figure 12.7d).

Iteration 4: Mark `hd(curr)`; and execute `invertLink(curr, prev, hd(prev))` (Figure 12.7e), execute another `invertLink(curr, prev, hd(prev))` (Figure 12.7f), set `tag(curr)` to 1, and execute `invertLink(prev, curr, tl(curr))` (Figure 12.7g).

Iteration 5: Mark `tl(curr)` to 1, set `tag(prev)` to 0, and execute `invertLink(curr, prev, tl(prev))` (Figure 12.7h). Execute `invertLink(curr, prev, hd(prev))` (Figure 12.7i), set `tag(curr)` to 1, and execute `invertLink(prev, curr, tl(curr))` (Figure 12.7j).

Iteration 6: Set `tag(prev)` to 0 and execute `invertLink(curr, prev, tl(prev))` (Figure 12.7k). The algorithm completes and `prev` becomes `null`.

Note that the algorithm has no problem with cycles in lists. `markingDFS` is slower than `markingWithStack`, since it continues to move its pointers through an already marked list. Hence, depending on a stack does not seem to be the best idea. Other approaches attempt to combine a stack with some form of overflow handling. A hybrid with `markingDFS` is to begin `markingWithStack` and switch to `markingDFS` if the stack becomes full. Other techniques attempt to intelligently select which interior nodes should be stored on the stack. For example, `markingWithStack()` might be modified to push onto the stack the nodes which have empty `aux` fields, under the assumption that they will be processed later. The `aux` field is then used to store the node's second child pointer, which is needed when the node is popped off the stack.

Figure 12.7 An example execution of the Schorr and Waite algorithm for marking used memory cells

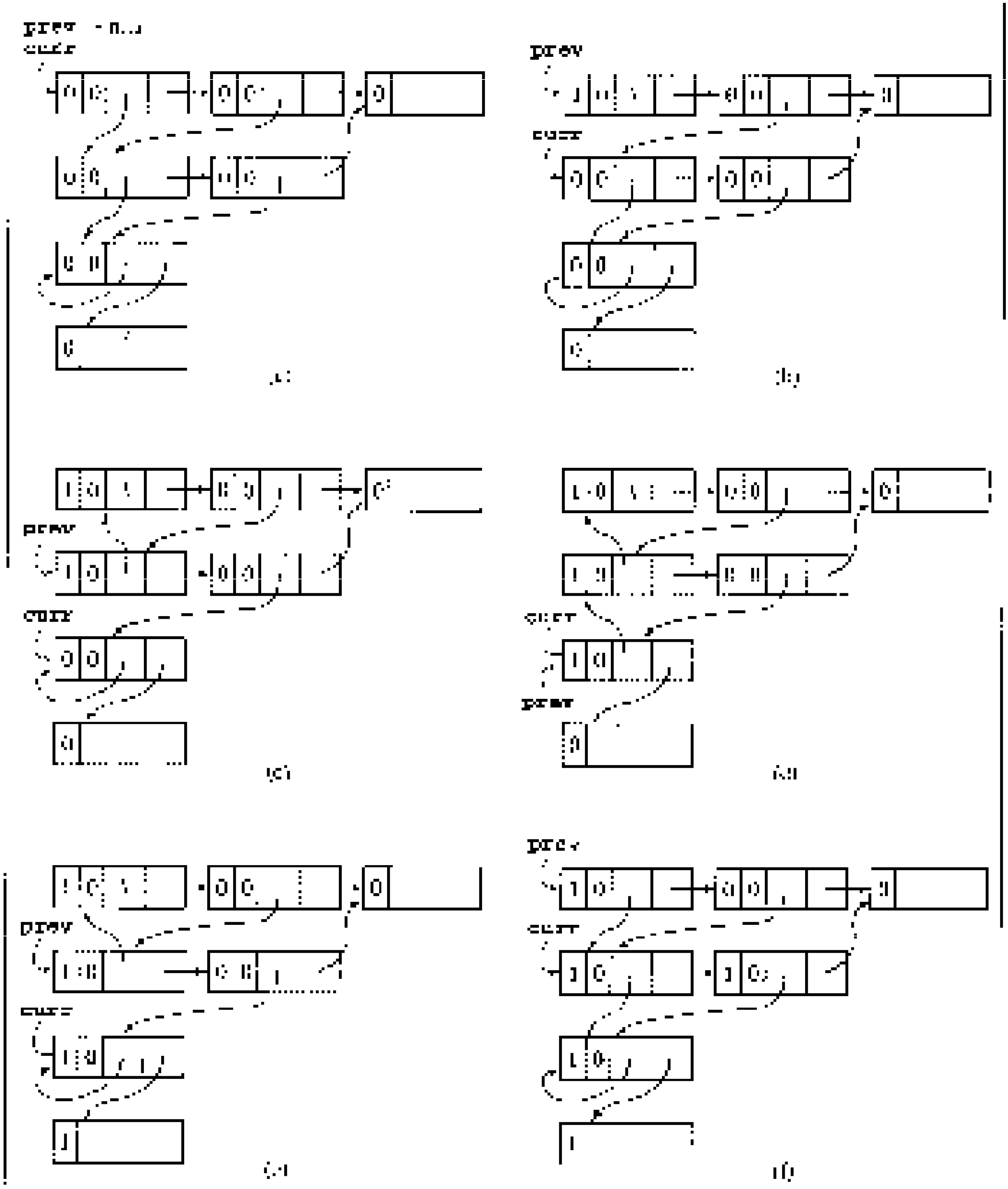
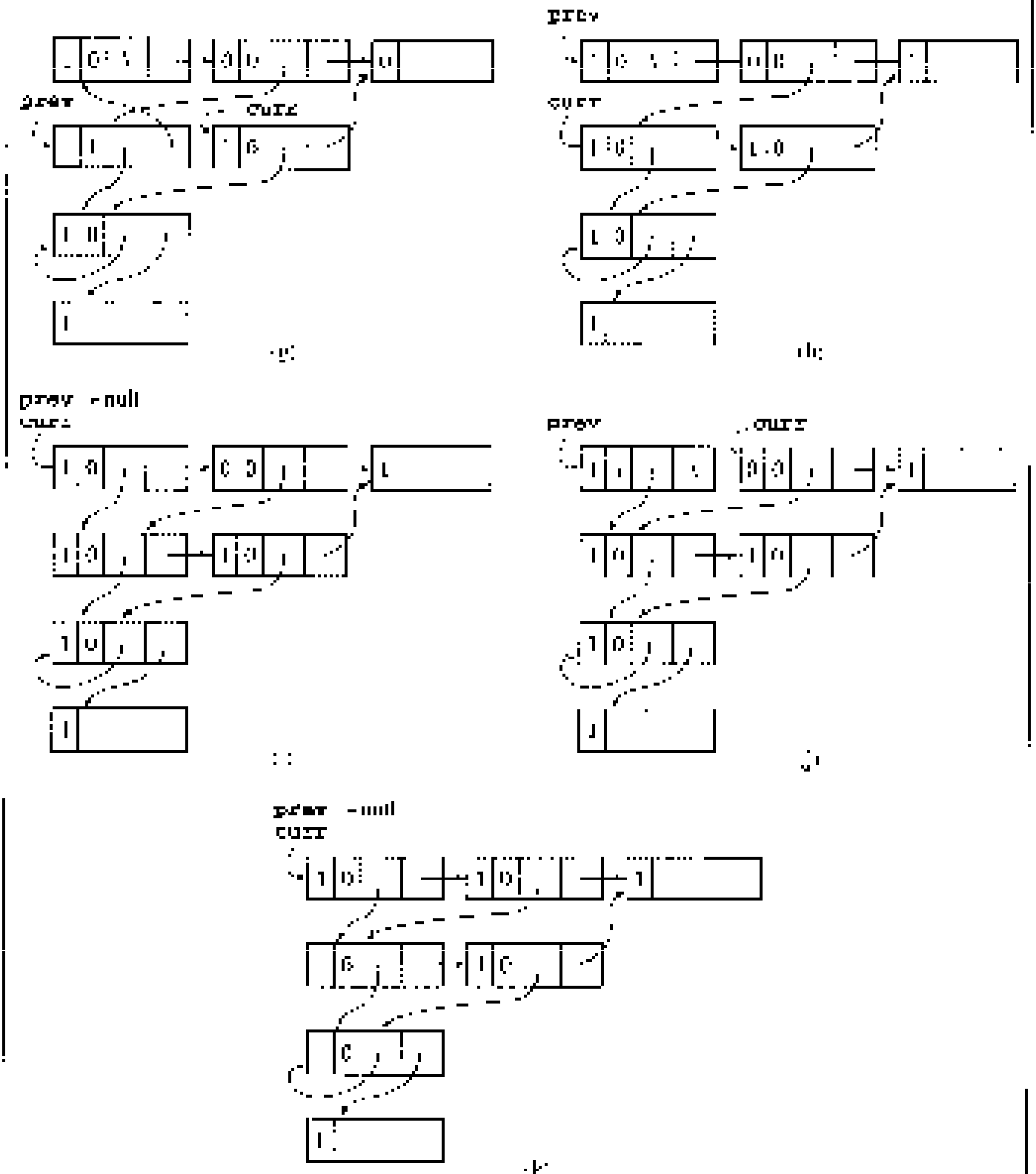


Figure 12.7 (continued)



The method described by Wegbreit requires no tag bits and uses a bit stack instead of a mark stack. One bit for each node on the trace path whose pointer fields both reference nonatoms. The trace path is the path from the current node to the root pointer. But as in the Schorr and White algorithm, link inversion is still in use. An improvement on this method is the Interlaced algorithm (Kurokawa 1981). As in Wegbreit's method, the Interlaced algorithm retains information about nodes which point to nonatoms on the stack. But the stack does not point to nodes, and so link inversion is unnecessary.

```
testmark(node)
   if node is nonatom
      mark node;
      while (1)
         if both(node, unmarked nonatom)
            if stack empty
               finished;
            else node = pop();
         else if only(node, unmarked nonatom is there)
            mark(node);
            node = nonatom;
         else if only(unmarked node) is not marked or is there
            mark(node);
            node = node;
         else if both(node, unmarked nonatom) and not marked or there
            mark(both(node, unmarked nonatom));
            push(node);
            node = both(node);
```

The reader is encouraged to apply this algorithm to the data in Figure 12.2. However, the vexing problem of stack overflow is far more complicated. Although the Deutsch algorithm claims to require approximately 3% recursion in most situations, some degenerate cases have been noted which require thousands of recursions. Therefore, the Deutsch algorithm must be extended to be robust. The basic idea of the resulting threaded marking algorithm is to delete from the stack nodes that are already marked or nodes whose nonatom path has already been traced. However, even this improved algorithm runs out of space occasionally, in which situation it gives up and advises a fatal stack overflow error (Sandewall 1969). Hence, the Schorr and White approach, with its recursive technique, is usually the most efficient through-flower.

Space Reclamation

After all the cells currently in use have been marked, the reclamation process returns all unmarked locations in the memory to the free list by going sequentially through the heap, cell by cell, starting from the highest address and moving toward zero, returning the unused cells to the free list. Upon completion of this process, all accessible cells are in ascending order in the free list. Thus, the next time the garbage collector is called, the memory that is used and cannot be accessed is returned to the free list. This algorithm is as follows:

```
Sweep()
    For each location from the base to the top
        If mark() associated with location
            Insert location in free list;
        Else Reset mark() associated with location;
```

The sweep algorithm involves a pass through the entire memory. It needs a pass required for marking and subsequent insertion of the used locations in the free list. Some implementations specially reserved for implementing the collection phase of the garbage collection algorithm.

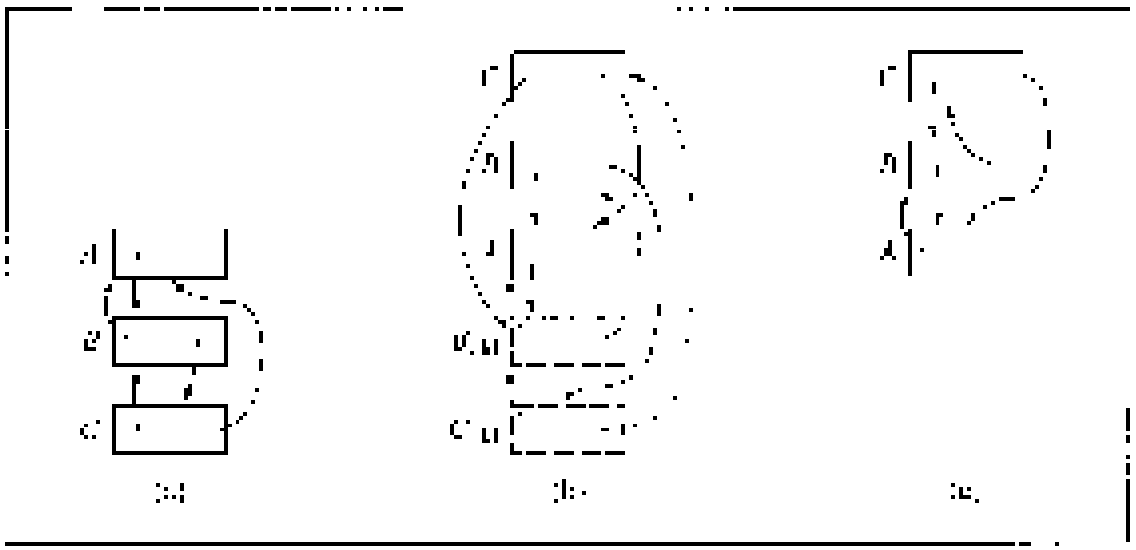
Compaction

After the mark-and-sweep process is complete, heap will be free of unused cells and the cells occupied by the program. This requires compaction. Call available locations contiguous cells, then there is a need to maintain the available locations. Also, garbage collection is used for relocating cells of equal size, such as pages. All available locations in sequence are highly desirable. Even when there is no necessity, garbage collection processes entire memory. In this way, responses to memory requests can be performed with a minimal number of accesses. And here is where mark-and-compacting is beneficial. It uses two pointers, one starting at the beginning of the heap and the other at the end of the heap, pointing toward each other. The heap and the stack are in opposite sides of memory and they grow toward each other. If occupied memory cells on the heap are available for the stack, then the stack has more room for expansion.

A simple two-pointer algorithm for heap compaction is as proposed earlier in the section for mark-and-sweep algorithm. Two pointers are used, one starting from opposite sides of memory. After the first memory finds an unmarked cell and the second finds a marked cell, the contents of the marked cell are moved to the unmarked cell, and its new location is recorded in the old location. This process continues until the pointers cross. Then, the entire heap part is scanned once to adjust the location of all pointers. If the pointers of the records of a reference have gone beyond the compacted area, then old locations are accessed to retrieve the new addresses. Here is the algorithm.

```
Compact()
    hi = (Location of heap);
    lo = (Location of heap);
    while (lo < hi) // Scan the entire heap
        while (lo < hi && cell pointed to by lo is marked)
            lo++;
        while (hi > lo && cell pointed to by hi is unmarked)
            hi--;
        Swap(cell at hi, cell at lo);
        Mark lo;
        Mark hi;
        Adjust lo--, hi++ // Because current lo and hi are
    lo = (Location of heap);
    while (lo <= hi) // Scan entire heap for compacted area;
```

Figure 12.8 An example of heap compaction



```
    free = next unmarked heap object in
      heap (free = null if no heap objects)
    while next unmarked heap object exists
      unmarked heap object = null (mark it)
  end sweep;
```

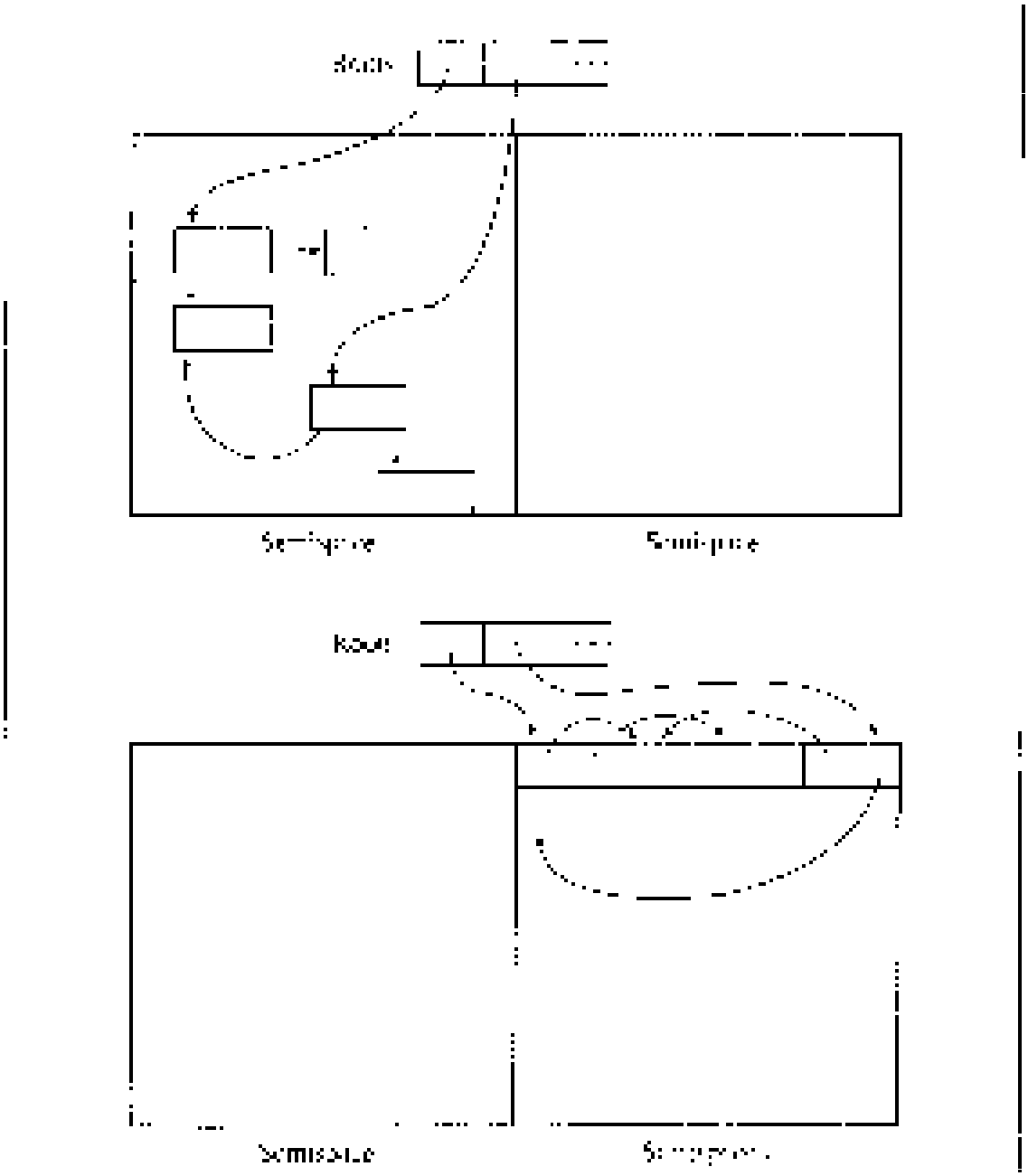
Figure 12.8 illustrates this process on the same two-word-wide memory. Four of cell A's fields point to other cells that should not be moved. Figure 12.8(a) illustrates the situation in the heap before compaction begins. In Figure 12.8(b), cells B and C have been moved into these spaces, and the pointer fields of the moved cells into existing, lower positions. Figure 12.8(c) illustrates the compacted portion of the heap after the marking bits and unused fields of all cells are updated, where necessary, to point to a position beyond the compacted area.

This simple algorithm is sufficient to illustrate the concepts, but it requires two passes through the heap for marked cells, a process that moves cells to new, lower locations, and one pass through the compacted area to update pointers, to make sure that heap pointers are valid. Our next method for memory compaction, the Lisp 2 algorithm, uses the same marking and sweeping, which updates pointers, but it requires only a single pass through the heap.

12.3.2 Copying Methods

Copying algorithms are also known as the semispace methods. Like the mark-and-sweep algorithm, they process only the accessible cells in the heap. Instead of marking them, they copy them as they are encountered. A second copy of the heap is maintained, and the copying operation is the only operation which gives the appearance of changing the heap. Cheney's algorithm [Cheney 1970] divides the heap into two contiguous areas of which only one is used for allocating storage at any time. When the active area is exhausted, the program is halted, and the copying operation begins. The copying operation traverses the active area, copying accessible cells into the second area. Pointers are updated to point to the new locations of the copied cells. When the copying operation is complete, the roles of the two areas are reversed: the area into which cells were copied becomes the active area, and the program resumes execution (see Figure 12.9).

FIGURE 12.9 (a) A situation in Cheney's algorithm before copying the contents of cells in use from semispace$_1$ to semispace$_2$, and (b) the situation right after copying. All used cells are packed contiguously.



Lists can be copied using breadth-first traversal (see Cheney [1970]). Lists were, in many cases, with no cross pointers. The algorithm would be the same as the breadth-first traversal discussed in Section 5.2.3. However, lists can have cycles and double pointers. We can point to cells we have already copied. Such a situation would produce multiple copies of the same cell. In the former case it would fall into an infinite loop. The problem can be easily resolved as follows: By retaining a forwarding address in the cell being copied. This allows the copying procedure to refer to a cell after it has

already been copied. Dijkstra's algorithm requires no extra memory space, but it does require a bit of overhead for each memory block. The basic idea is that each block has a color associated with it, and the algorithm uses these colors to determine which blocks contain garbage that can be reclaimed. The algorithm proceeds as follows:

Note that the cost of garbage collection is directly proportional to the size of the heap (as opposed to the number of blocks on the heap). This is because the algorithm must visit every block in the heap to determine whether it is garbage. Thus, as the heap grows, the cost of garbage collection grows. For example, a program using 4MB of memory might spend 0.5 seconds performing garbage collection. The same program using 8MB of memory might require 1 second for garbage collection. This overhead can quickly become unacceptable as memory grows. Fortunately, more sophisticated garbage-collection algorithms have been developed that can greatly reduce the cost of garbage collection. These algorithms are discussed briefly in the following sections.

12.3.3 Incremental Garbage Collection

Garbage collection can be invoked automatically when the memory manager runs low on memory. This approach can result in a noticeable pause in program execution while the garbage collector finishes its work. Garbage collection can take several seconds, which may be unacceptable in many real-time systems. For example, it is unacceptable to pause a video game for several seconds while garbage collection occurs. Fortunately, it is often possible to perform garbage collection in small chunks over a long period of time, so that no single pause in program execution is noticeable. This technique is known as *incremental garbage collection*. The basic idea is that when the garbage collector runs, it performs a small amount of garbage collection and then allows the program to run for a while before it is needed again. Eventually, enough garbage will be collected to satisfy memory requests. Incremental garbage collection is more complicated than regular garbage collection because the memory manager must be able to track changes to memory that occur while the garbage collector is running. Otherwise, the garbage collector may erroneously reclaim memory that is in use by the program. Many algorithms have been developed for incremental garbage collection. Most of these algorithms are based on the concepts of reachability and color that were introduced in the preceding section. The following sections discuss two of the most common incremental garbage-collection algorithms.

Copying Methods in Incremental Garbage Collection

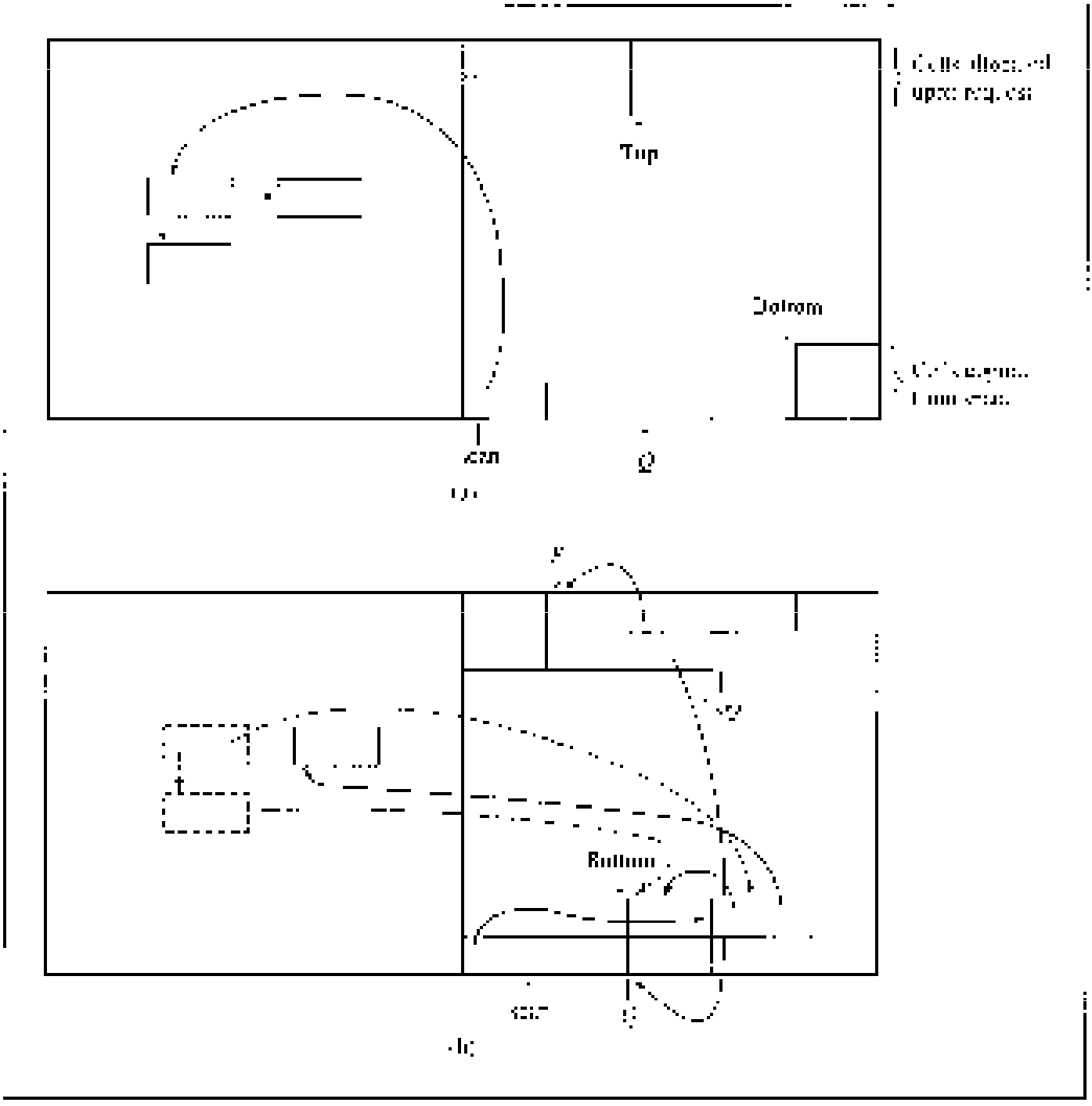
A well-known algorithm based on the semi-space copying technique is known as Baker's algorithm (Baker [1978]). As in stop-and-copy, Baker's algorithm divides memory into two spaces, called *from-space* and *to-space*, which are both the same size. The memory manager allocates memory only from from-space. When a memory request arrives and there is not enough room in from-space to satisfy the request, the garbage collector is invoked. The collector then begins copying live blocks from from-space to to-space. In this way, the collector gradually frees up memory in from-space. Unlike stop-and-copy, however, the collector does not copy all live blocks in from-space to to-space during a single invocation. Instead, the collector copies only enough blocks to satisfy the memory request that caused the collector to be invoked. After a small number of memory requests, from-space will fill up again, and the collector is invoked again. This time, the collector resumes copying live blocks where it left off the last time. Figure 12.10 illustrates the operation of Baker's algorithm.

In order to do this it maintains two pointers. The free pointer is one which points to the next free word that an object should be copied to (space if they all are in fromspace). Initially free is set to the beginning of tospace. The scan pointer is also initialized to the beginning of tospace, and the objects are put at the end of the queue. This queue is simply accessible by the second pointer, scan, which points to the beginning of the free space in tospace. The collector can process cells at the current cell during the same time. But it may wait until the next one. Figure 12.11 contains an example. If a request comes to allocate a cell whose head points to P and tail to Q (as in Figure 12.11(a)), and if Q is pointed to by some Pand Q existing in tospace, then a new cell is allocated in the upper part of tospace with both its pointers properly initialized. Assuming that $k = 2$, two cells are copied from the beginning of the queue pointed to by scan, and the task is processed when the next request comes. As in stop-and-copy, Baker's algorithm returns a forwarding address in the original cell so that references to it copy in tospace and in case later cons cells refer to this original.

Special care must be taken when the head and/or tail of a cell being allocated refer to a cell in fromspace that is either already copied or still in fromspace. Because the cells at the top of tospace are not processed, the collector returning a pointer to any cell found in fromspace cells leads to fatal consequences after fromspace becomes tospace, because the latter cells are now considered available and filled with new contents. The mutator could at any point use the pointer to the original and at a later point create a copy leading to inconsistencies. Hence, the mutator is preceded by a read barrier which precludes unfixing references to cells in fromspace. In the case of a reference to fromspace, we have to check whether this cell has a forwarding address. If it does not (it is still a cell in fromspace), then it is copied to tospace and the address of the new cell is returned. If it does, then the forwarding address is used in the allocation. Otherwise the cell referred to in the current allocation has to be copied before the actual allocation takes place. Two examples. If the head of a cell to be allocated is to point to P, a cell in fromspace which has already been copied as illustrated in Figure 12.11(a), P's new address is stored in its address space (Figure 12.11(b)). If the tail of the new cell is to point to Q, which is still unhooked in fromspace, Q is copied to tospace relative to its descendant, since $k = 2$, and only after the tail of the new cell is initialized to the copy of Q.

Baker's algorithm lends itself to various modifications and improvements. For example, to avoid constant conditional tests when allocating new cells, an indirection field is included in every cell. If a cell is in tospace, the indirection field points to itself; otherwise, it points to its copy in tospace (Brooks 1984). Tests are avoided, but indirection pointers have to be maintained for every cell created. Another way to solve the problem is by utilizing hardware facilities if available. For example, memory protection facilities can prevent the mutator's access to cells not processed by the collector. A page of the heap with unprocessed cells can be read protected (Ellis, Li, and Appel 1988). If the mutator attempts to access such a page, the access is trapped and an exception raised, forcing the collector to process the page so that the mutator can resume execution. But this method can undermine the incremental collection, since after the semispaces change roles, pages are mostly frequently protected, so that an entire page of the heap be processed. Some additional provisions have to be needed such as maintaining a scan of the entire page in case of a trap. On the other hand, if the heap is not accessed too frequently, this is not a problem.

Figure 12.10 A situation in memory (a) before and (b) after allocating a new block and interpointers pointing to cells found (c) to require updating in the Baker algorithm.



An interesting modification to Baker's algorithm is one based on the observation that most allocated cells are needed for a very short time after they are allocated, but a long time after that. This leads to a generational garbage collection algorithm, which divides all allocated cells into at least two generations and concentrates collection on the youngest generation, which generates most of the garbage. Such cells do not

FIGURE 12.11 Changes performed by the Baker algorithm when addresses P and Q refer to cells in from space (P to an already copied cell, Q to a cell still in from space).

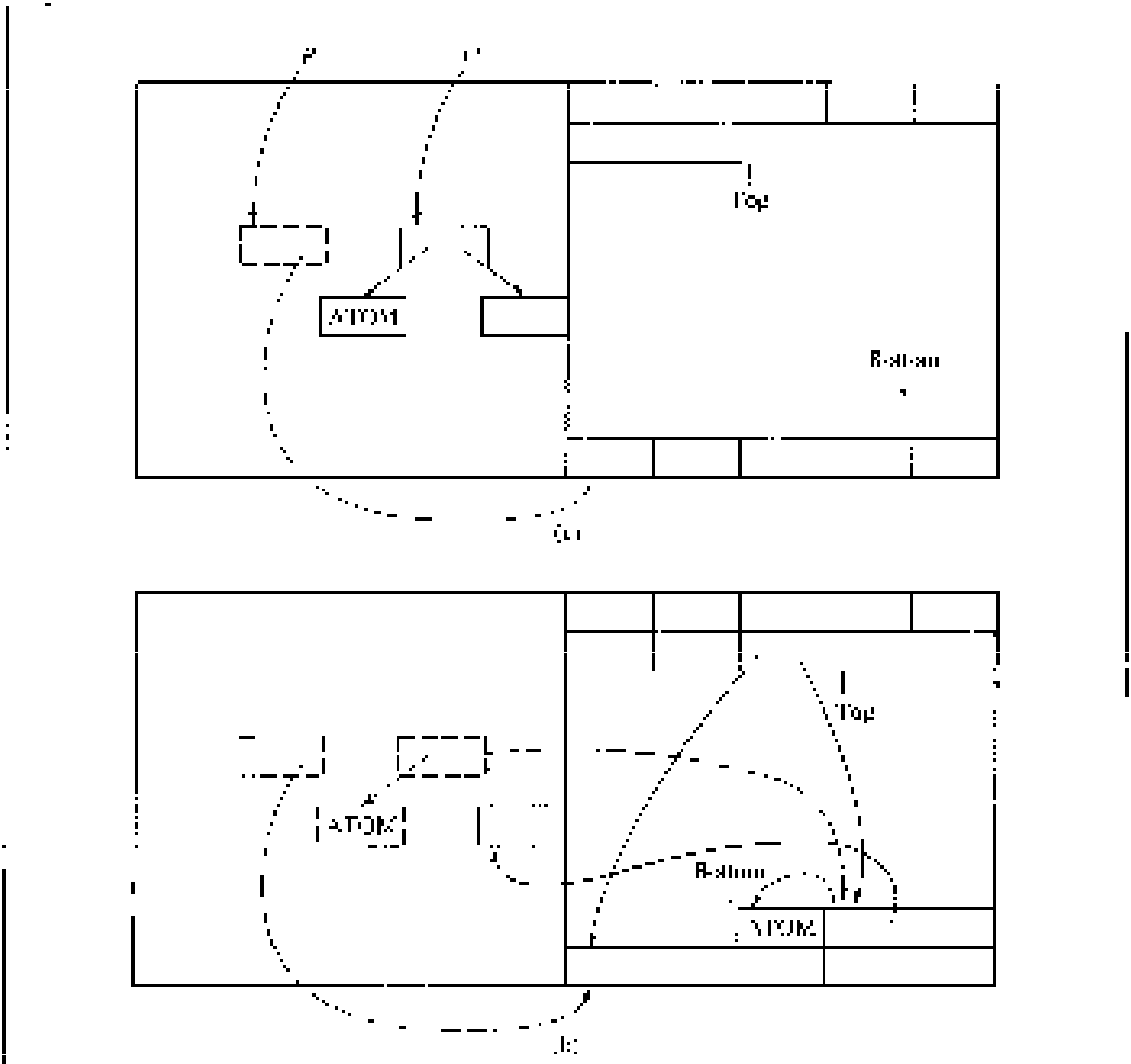
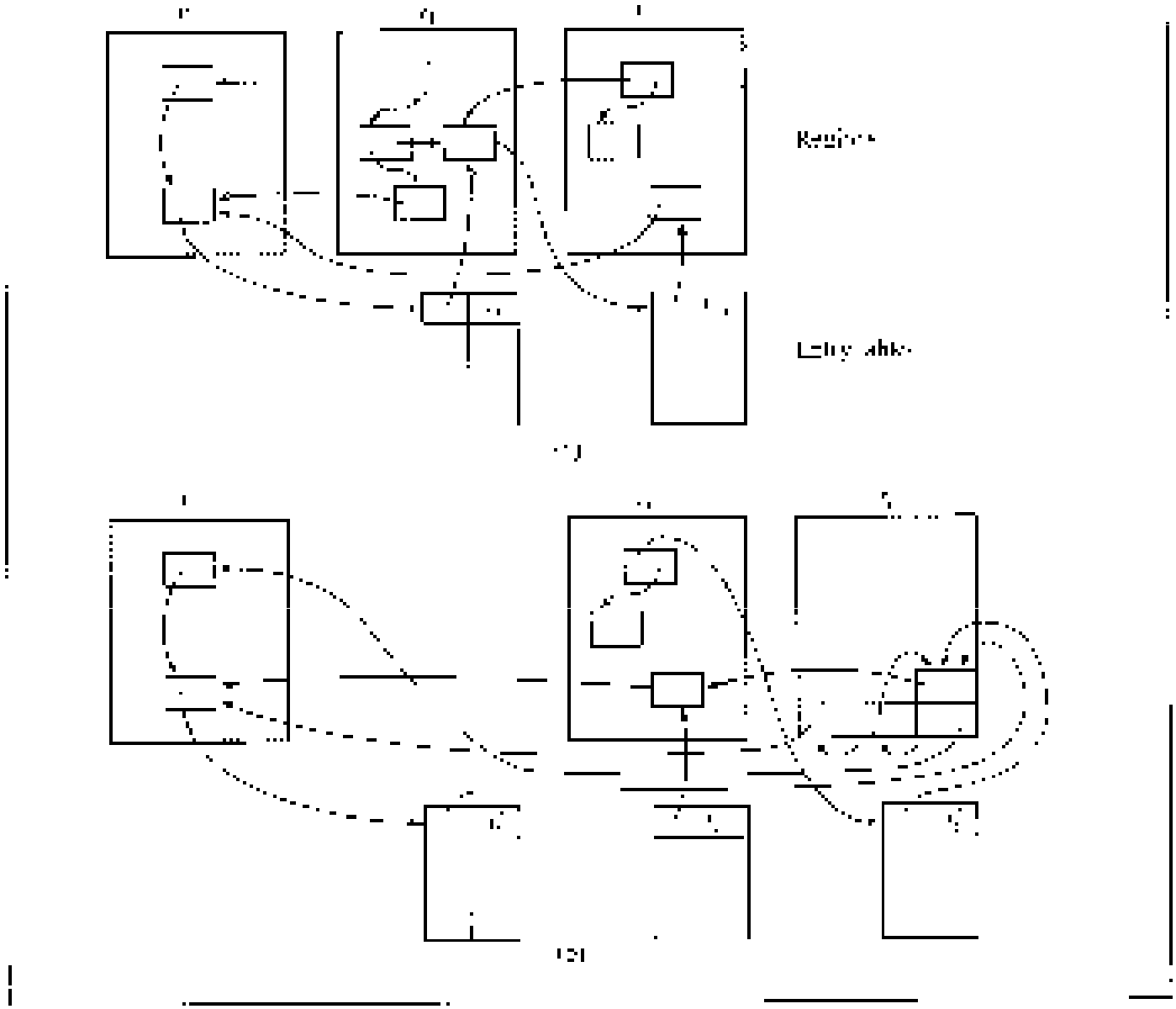


FIGURE 12.12 A situation in the operation of Lieberman and Hewitt's copying reachable cells from region r_1 to region r_2 in the Lieberman–Hewitt incremental generational garbage collector.



promote cells of an older generation. Some of these, however, can point forward to r_1 or r_2, when Lisp's rplaca is used. In this unusual situation, such forward references are automatically brought into entry tables of the target region. A pointer in a region r_3 that points to a cell in a region r_4 has a cell in the entry table of r_4 that points to it. If r_3 contains a pointer to the region r_1 that is full, all reachable cells in region r_1 are copied to region r_2, and all regions with cells younger than r_2 are visited to update pointers referring to cells which have been transferred to the new region. Regions with generations older than r_2 do not have to be visited. Presumably, only a few references in the entry table of r_2 are updated (see Figure 12.12). In

[illegible] problem of finding the entry tables that are used by garbage collection to determine which objects are accessible from outside the region. Any object pointed to by the entry table for region r_4 in Figure 12.12, for example, must be checked to see if the region itself is inaccessible.

Noncopying Methods

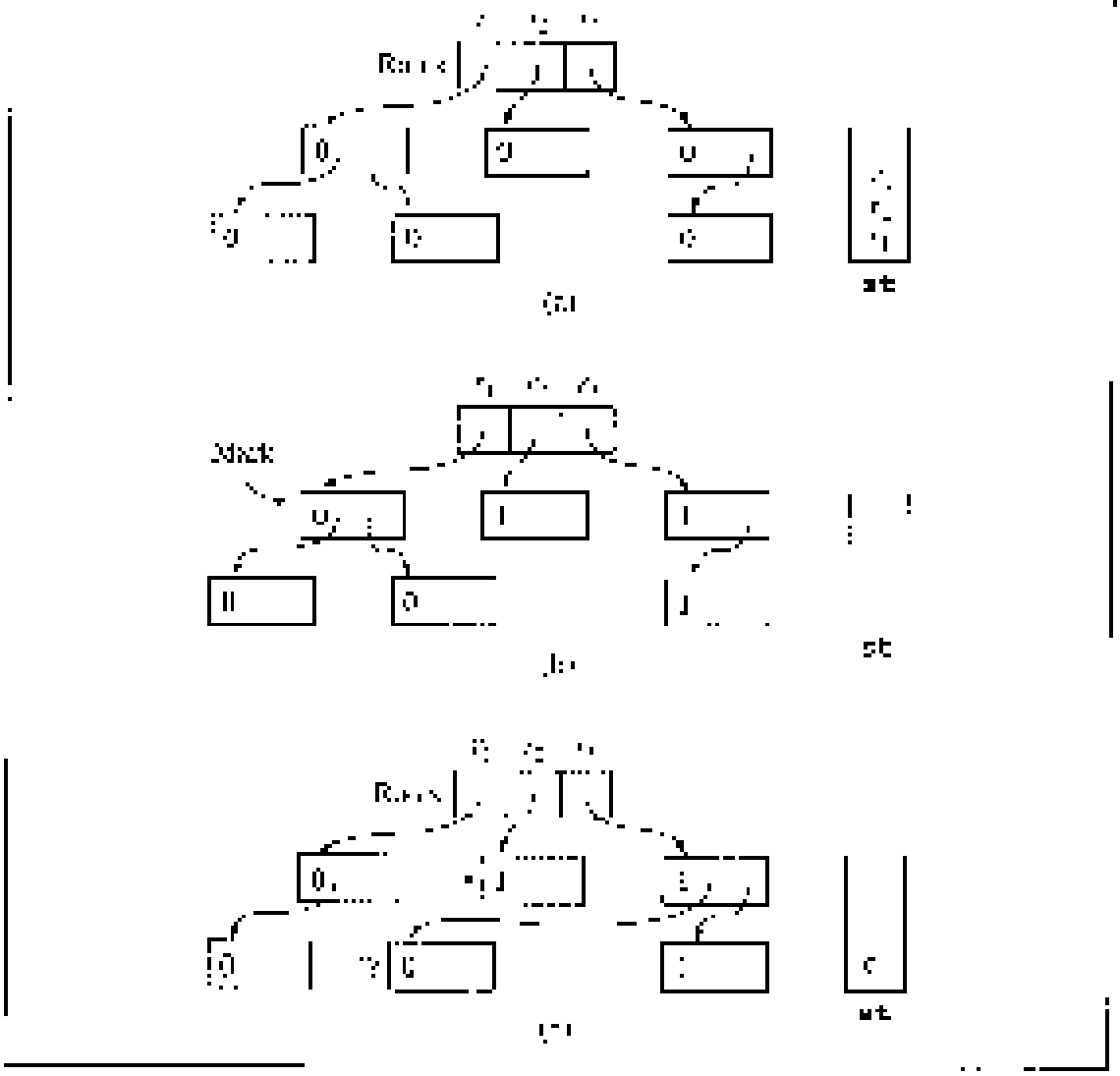
The [illegible] techniques based on copying. The problem is not so much with the amount of memory allocated to each object, but with their positions in memory, which have to be fixed. The application must not treat the addresses as integers otherwise, the program crashes. Therefore, some mechanism is needed to update the address of each object that has been moved. Typically, the [illegible] pointers to the copies. However, because of the expensive and [illegible] guess, the mark-and-sweep method is not [illegible] it simply [illegible] code. The simplicity of this method is [illegible] attempt to [illegible] it to real-time [illegible] results.

Mark-and-sweep collection [illegible] by [illegible] the [illegible] marking phase [illegible] the [illegible] by including [illegible] cell. The marking phase is [illegible] that used in the mark-and-copy algorithm except that it is [illegible] exclusively. The marking procedure [illegible] cells by [illegible] the [illegible] b_1 cells have been marked, the [illegible] the [illegible] b_2 is used during the sweeping phase to locate [illegible] cells that [illegible] be processed before [illegible] is turned over to the [illegible] by [illegible] to [illegible] whether it is in the middle of marking or sweeping. The [illegible] for marking or sweeping is always invoked after [illegible] is requested. [illegible] procedure [illegible] that [illegible] the root pointers and [illegible] in the following pseudocode

```
[illegible]
    if [illegible] for the marking phase
        mark(p) & [illegible]
    else if [illegible] in the sweeping phase
        sweep(p) & [illegible]
    else if [illegible] the marker phase has just been completed
        [illegible]
    p = [illegible]
    [illegible]
    mark(p) [illegible]
    mark(p) [illegible]
```

[illegible] [illegible] [illegible] [

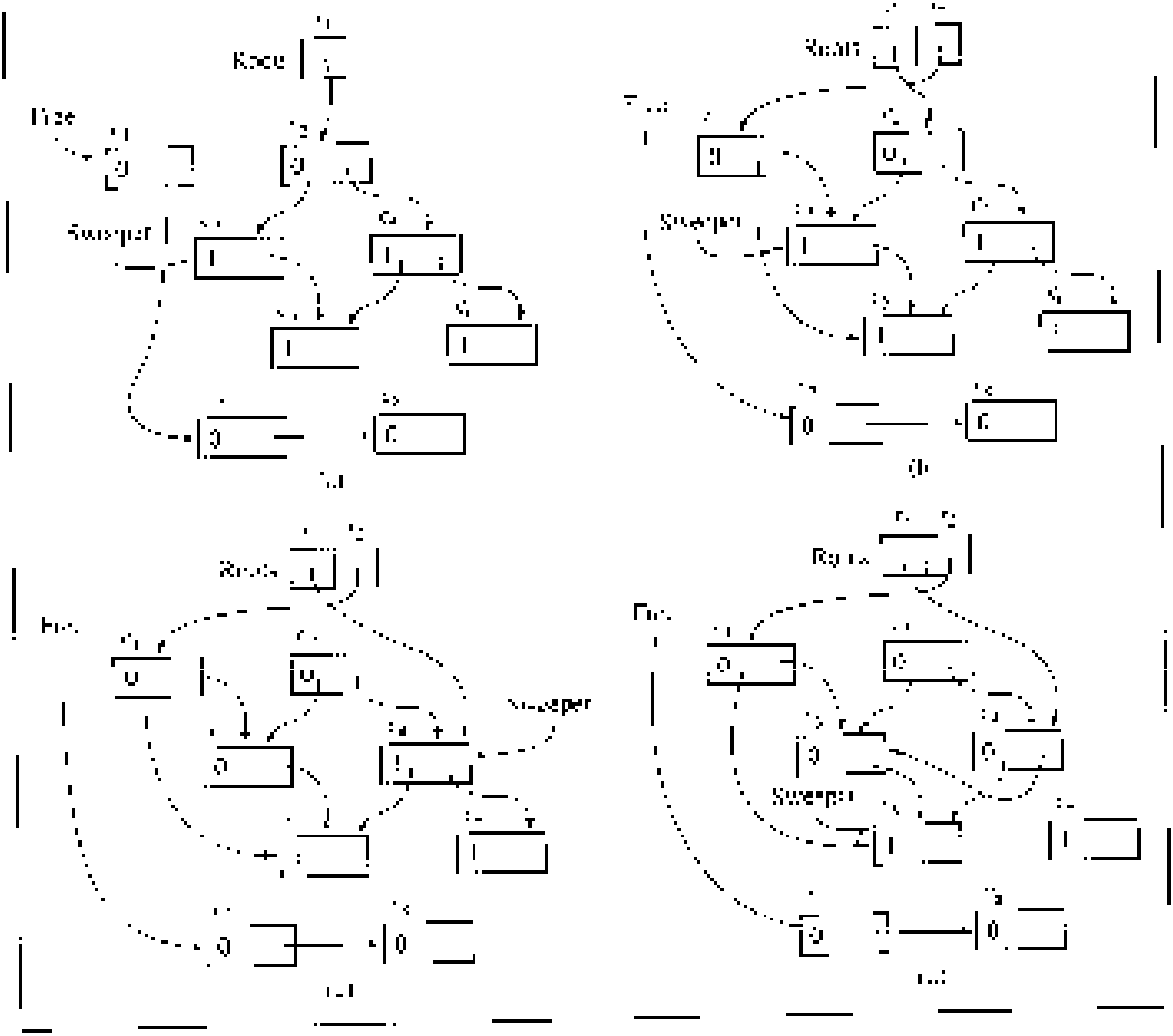
Figure 12.13 An inconsistency that occurs in Yuasa's snapshot-at-the-beginning concurrent garbage collector. A stack is not used to record cells that are encountered during the marking phase.



processed (Figure 12.13b), the mutator overwrites the pointer from x_2 to x_3 and from y_2 to x_3, and x_3 is assigned x_2 (Figure 12.13c). If the marking process is now restarted, it has no chance to mark x_3 since the entry for x_3 is assumed to have been processed. This leads to the collector reclaiming x_3 in error. To prevent that, the mutator which updates the pointer from x_2 to x_3 also pushes the old value of the field being updated on a mark stack used by the garbage collector. For example,

```
writeField(x, y)   // x.fields[y] = ...
    if collector is in the marking phase
        mark(x.fields[y]);
        stack.push(x.fields[y]);
    x.fields[y] = ...;
```

FIGURE 12.14 Memory changes during the sweeping phase using Yuasa's method



In the marking phase, the stack can be popped up to three times for each cell that is popped off its boundary cells are marked.

In the sweeping phase, memory is gone through the heap in address order, unmarked cells are linked to the free list, and marked cells are unmarked. For the sake of clarity, a new cell is allocated, it remains unmarked if it is taken from the heap that has already been swept. Otherwise, the next round of mark-up could lead to distorted results. Figure 12.14 illustrates an example. In (a) the pointer to the heap has already reached the end, and the mutator requests a new cell. We assume that the free list is empty, so the first cell is detached from the free list and made a new root (Figure 12.14b). But the newly allocated cell is unmarked because it precedes the sweeper on the heap.

free cells are used in a sweep phase, it becomes garbage, but is not swept until a subsequent sweeping phase. It can be reclaimed only if it is not referenced by any cell that is garbage. In other words, floating garbage is any garbage that is not collected on the current collection cycle.

12.4 Concluding Remarks

When assessing the efficiency of memory management algorithms and especially garbage collectors, we have a fundamental problem. Paul Wilson's comprehensive survey [Wilson 1992] is essential reading for anyone interested in this area, but it is long and detailed. Jones and Lins [1996] devote only a few pages to the issues of performance and evaluation, but their book is still a valuable companion to the study of automatic storage reclamation. It is those algorithms whose cost is sensitive to the size of the heap—mark-and-sweep collectors and most forms of copying collector—that are the most difficult to analyze. Their performance is typically characterized by a few parameters, such as the heap occupancy (the ratio of live memory to heap size). However, other metrics, such as the frequency of sweeping or fragmentation, are also important in the cost of copying algorithms. Jones and Lins [1996] also point out that the performance of the mark-and-sweep and the various copying techniques are very similar [Zorn 1989].

This example illustrates how difficult it is to make definitive statements about the performance of a garbage collector. The algorithm and its implementation are affected by characteristics of the underlying hardware, the operating system, the compiler, the application, and the heap occupancy. Available memory is another important factor. This is especially important for generational garbage collectors, whose efficiency relies on the assumption that most allocated cells will become garbage very quickly. On the other hand, a sweep-and-compact collector may perform better than copying techniques when the heap occupancy is high.

Asymptotic complexity analysis can be misleading. For example, a memory management algorithm with good asymptotic behavior may perform poorly in practice. Many of the algorithms we have described have been proposed in the context of functional languages, where it is impossible to create cyclic data structures. Their performance is usually described in terms of the number of cells in use, the number of cells allocated, and the number of pointers in the system. Unfortunately, these measures are not always relevant in practice. For example, in a system with virtual memory, the locality of reference can be more important than the number of cells in use. Hudak and Bloss [1985] have shown that a simple copying collector can outperform a sophisticated generational collector in a virtual memory environment.

Memory management algorithms are usually closely tied to the hardware on which they are implemented. Which algorithm is chosen for a specific garbage collection task is substantially affected by the availability of hardware support. Baker's treadmill algorithm [Baker 1992] is an example of a collector that was designed to take advantage of hardware support. Without hardware support, a naive implementation of the algorithm can be very inefficient. Unfortunately, hardware support is rarely available for garbage collection. Most commercial implementations of garbage collectors rely on software techniques. Even hardware support for write barriers is rare. Most commercial implementations of garbage collectors rely on software techniques. Even hardware support for write barriers is rare. Most commercial implementations of garbage collectors rely on software techniques.

tasks, where the responsiveness of the computer is the issue, additional overhead of the garbage collector is added. It may not necessarily claim in the case of nonincremental collectors, since at no time does a program have to wait an arbitrary length for the collector to finish its task. However the latency of interactive actions should be proportional to real-time constraints.

12.5 Case Study: An In-Place Garbage Collector

Copying algorithms for garbage collection are efficient in that they do not require processing unused cells. Cells that are not processed are considered garbage at the end of collection. However, these algorithms are inefficient in copying reachable cells from one semispace to another. An in-place garbage collector attempts to retain the advantage of copying algorithms without producing copies of the reachable cells (Baker 1992).

The in-place algorithm constantly maintains two doubly linked lists, `freeCells` and `accessibleCells`. The list `freeCells` initially contains all cells of `heap[]`, and a cell is moved from `freeCells` to the other list if a request comes to construct a list or construct a new atom. After `freeCells` becomes empty, the function `collect()` is invoked. This function first installs all pointers from `nonfreeCells` to an intermediate list `markedAccessibles` and sets their `marked` field to true. Then `collect()` detaches from `markDescendants` cell by cell to transfer such cell to another temporary list, `markedCells`. For each nonatom cell, `collect()` attaches to `markDescendants` unmarked head and tail pointers to be processed next. In the case of a list, they are attached to the beginning of `markDescendants`, thereby leading to depth-first traversal of list structures. For breadth-first traversal as in Cheney's algorithm, they have to be attached to the end of `markDescendants`, which requires another pointer to the end of the list. Note that, although a cell is transferred to `markedCells`, it is also marked to prevent infinite loops in case of cyclic structures and redundant processing in case of interconnected noncycling structures.

After the list `markDescendants` becomes empty, all reachable cells of `heap[]` have been processed and `collect()` is almost done. Before returning from `collect()`, all cells left in `accessibleCells` become members of `freeCells`, and all marked cells are put on `nonfreeCells` after clearing their `marked` fields to false. The user program can now resume.

To the user, the garbage collector is part of the program's environment and is executed in the background almost unnoticeable to the user. To exemplify the workings of a garbage collector, some elements of the program background are simulated in the case study, in particular the heap and the symbol table.

The heap is implemented as an array of objects with two flag fields, `atom` or `nonatom` and `marked` or `unmarked`, and two pointer fields which are really integer fields indicating positions in `heap[]` of the previous and next cell (if any). In accordance with this implementation, both permanent lists, `freeCells` and `accessibleCells`, and both temporary lists, `markDescendants` and `markedCells`, are simply integers indicating the index in `heap[]` of the first cell on a given list of cells.

The symbol table is implemented as an integer array roots[] of root pointers. No explicit variable names are used, only indexes to heap[] cells. For example, if roots is [5 2 4 0], then only four variables are currently in use by roots[0] through roots[3], and these variables are pointing to cells 5, 2, 4, and 0 in heap[]. The numbers [illegible] can be seen as [illegible] more palpable variable names, such as var, temp, [illegible] and [illegible].

The user program() is just a [illegible] simulator which does nothing but request allocations and deallocations of heap[]. These requirements are generated randomly and classified by the type of request: 30% are atom [illegible] allocations, 20% are list [illegible] allocations, 20% are head updates, 20% are tail updates, and the remaining 10% are deallocations. Deallocation is simulated by the function deallocate(), which decides whether a [illegible] generated root should be assigned empty (which represents the null reference) or a local block is exited, whereby all local variables are removed, which [illegible] means that it is only assigned to [illegible]. The [illegible] can be assigned differently [illegible] the number of assignments already made. This is just [illegible] changes in program(). Also, the size of heap[] and the size of roots[] can be modified.

The user program() randomly generates a number rand between 0 and 99 to indicate the operation to be performed. Then, variables are randomly chosen from roots[]. For example, if roots[] is [5 3 4 0], and g is 2, then the cell roots[g] = 4 of heap[] [illegible] by storing the value of [illegible]. If g = 4, this indicates that a new variable [illegible] has been created in position 4 of roots[], and position roots[4] is assigned the first cell from freeCells.

To see that this program does something, a simple function printList() is supplied which prints elements [illegible] and after each operation is [illegible] the contents of heap[] and of roots[]. Here is an example of an output generated by applying this program to a heap of six cells.

```
roots: 1 5 3
(0: -1 2 0 2 0 0) (1: 3 4 0 0 1 4) (2: 2 -1 0 [illegible] 2 2)
(3: 4 -1 1 [illegible] 3 0 5) (4: [illegible] 2 1 0 [illegible] 4) (5: [illegible] 1 0 3 [illegible])
freeCells: (0 [illegible] 0) (2 2 2)
nonFreeCells: (5 5 1) (1 [illegible] 4) (4 1 9 4) (3 [illegible])
```

This output represents the situation illustrated in Figure 12.15. Figure 12.15a shows the contents of [illegible] links previous and next used by lists freeCells and nonFreeCells, and head link [illegible] tail link [illegible] of nonatom cells. For atom cells, [illegible] value is stored [illegible] number of links [illegible] the number of crossing links. The same situation is presented in Figure 12.15b, where the cells are organized by [illegible] than the positions in heap.

Figure 12.15 [illegible]

FIGURE 13.15 An example of a situation on the heap:

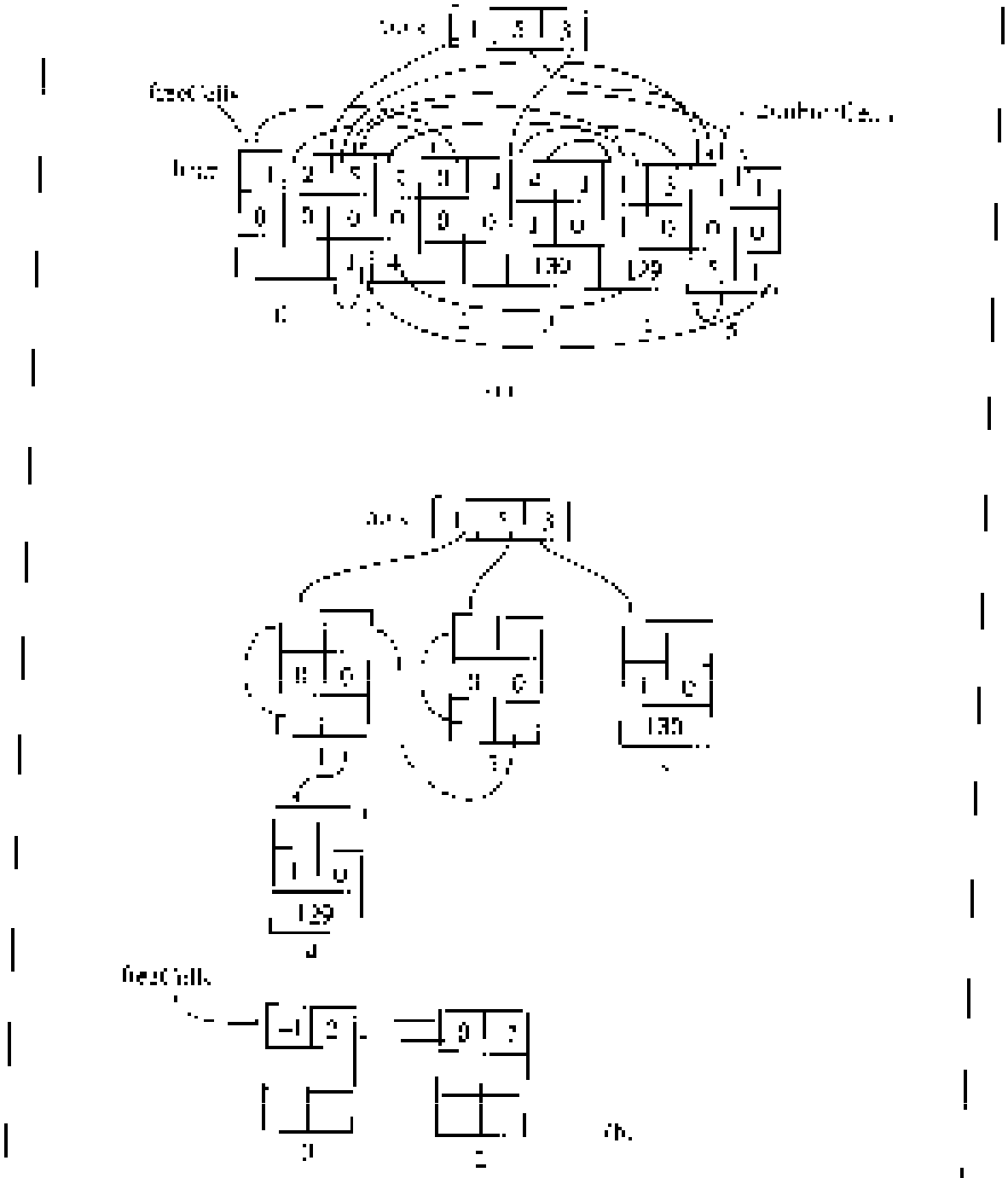


Figure 12.16 Implementation of an in-built garbage collector.

```
//-------------------------------------  Length  ---------------------------------

#ifndef HEAP_CLASS
#define HEAP_CLASS

class Cell {
public:
    bool check, marked;
    int prev, next;
    Cell() {
        prev = next = -1; nfcell.next[0] = nfcell.next[1] = -1;
    }
    union {
        int value;      // value for atom,
        int link[2];    // head and tail for non-atom;
    } info;
};

class Heap {
public:
    int nextfree;
    Heap();
    void mgcLinkHead(int p, int q) {          // Link p right to q;
        if (root[p] != empty && !atom[root[p]])
            Mem[root[p]].next = root[q];
    }
    void mgcLinkTail (int p, int q) {         // Link p right to q;
        if (root[p] != empty && !atom[root[p]])
            Mem[root[p]].prev = root[q];
    }
    void mgcLinkOption(int,int);
    void mgcLinkTo(int,int,int);
    void markFrom(int);
    void print(int,ostream&);
private:
    const int empty, OK, Read, Full, maxHeap, maxRoot;
    Cell *Mem;
    int *root, *protect, nextProtect;
    void Mark(int p) {
        return heap[p].info.link[mcell];
    }
```

Figure 12.16 (continued)

```
    int Tail(int p) {
        return heap[p].info.link[tail];
    }
    int value(int p) {
        return heap[p].info.value;
    }
    int prev(int p) {
        return heap[p].prev;
    }
    int next(int p) {
        return heap[p].next;
    }
    int & mark(int p) {
        return heap[p].mark;
    }
    int & marked(int p) {
        return heap[p].marked;
    }
    void insert(int,int);
    void detach(int,int);
    void transfer(int cell, int list1, int list2) {
        detach(cell,list1); insert(cell,list2);
    }
    void collect();
    allocate(int);
    friend ostream &operator<< (ostream&,Heap&);
};

Heap::Heap() : empty(1), SS(1), head(0), tail(1), maxHeap(5),
maxCell(20) {
    headCells = numberOfCells - empty;
    rootCnt = 0;
    heap = new Cell[maxHeap];
    roots = new int[maxCell];
    for (int i = maxCell-1; i >= 0; i--) {
        roots[i] = empty;
    for (i = maxHeap-1; i >= 0; i--) {
        insert(i,freeCells);
        mark(i) = false;
    }
}
```

Figure 12.16 (continued)

```
void Heap::deallocate(int cell, int last) {
    if (next[cell] != end())
        prev[next[cell]] = prev[cell];
    if (prev[cell] != end())
        next[prev[cell]] = next[cell];
    if (cell == last)                    // head of the list
        last = next[cell];
}

void Heap::makeAvailable(int cell, int last) {
    prev[cell] = empty;
    if (cell == last)          // heap is empty
        next[cell] = empty;
    else next[cell] = last;
    if (last != empty)
        prev[last] = cell;
    last = cell;
}

void Heap::collect() {
    int markedDescendants = empty, markedCells = empty;
    for (int p = 0; p < roots.size; p++) {
        if (roots[p] != end()) {
            traverse(roots[p],markedCells,markedDescendants);
            marked[roots[p]] = true;
        }
    }
    printf("%d descendants, %d cells\n",markedDescendants,markedCells);
    for (p = markedDescendants; p != empty; p = next[p]) {
        transfer(p,markedDescendants,markedCells);
        if (color[p] && !marked[head[p]]) {
            traverse(head[p],markedCells,markedDescendants);
            marked[head[p]] = true;
        }
        if (color[p] && !marked[tail[p]]) {
            traverse(tail[p],markedCells,markedDescendants);
            marked[tail[p]] = true;
        }
    }
    makeAvailable(
```

Figure 12.16 (continued)

```
    printList(markedCells,"markedCells");
    for (p = markedCells; p != empty; p = next(p))
        marked(p) = false;
    freeCells = nonFreeCells;
    nonFreeCells = markedCells;
}

Heap::allocate(int p) {
    if (p >= maxSize) {
        cout << "No space for new cons\n";
        return !OK;
    }
    if (freeCells == empty)
        collect();
    if (freeCells == empty) {
        cout << "No room in heap for new cell\n";
        return !OK;
    }
    if (p >= nextOut)
        nextOut = nextOut = p;
    roots[p] = freeCells;
    transfer(freeCells.first, nonFreeCells);
    return OK;
}

void Heap::allocate(int p, int val) // an instance of Lisp's setf;
    if (allocate(p) == OK) {
        atom(roots[p]) = true;
        value(roots[p]) = val;
    }
}

void Heap::allocate(int p, int q, int r) { // Lisp's cons;
    if (allocate(p) == OK) {
        atom(roots[p]) = false;
        head(roots[p]) = roots[q];
        tail(roots[p]) = roots[r];
    }
}
```

(continued)

Figure 12.16 (continued)

```
void Heap::operator delete(void p) {
    if (rootCnt > 0)
        if (frank[(int)p / 2] == 0)
            roots[p] = roots[--rootCnt]; // remove variable when exit my
                                         // a c code
        else roots[p] = empty; // set variable to null
}

void Heap::printAll(int last, char *name) {
    cout << name << ": ";
    for (int i = last; i != empty; i = next[i])
        cout << "[" << i << ' ' << Head[i] << ' ' << Tail[i] << "]";
    cout << endl;
}

ostream &operator<< (ostream &os, Heap &h) {
    cout << "roots: ";
    for (int i = 0; i < h.rootCnt; i++)
        cout << "r[" << i << "]=" << h.roots[i] << " ";
    cout << endl;
    for (int i = 0; i < h.maxCons; i++)
        cout << "[" << i << ": " << h.Head[i] << " " << h.next[i]
             << "]=cons(" << h.Head[i] << ", " << h.Tail[i] << ")";
    cout << endl;
    h.PrintList(h.freeCells,"freeCells");
    h.PrintList(h.nonfreeCells,"nonfreeCells");
    return os;
}

#endif

//-------------------------  collect.cpp  -------------------------

#include <iostream>
#include <cstdlib>
#include "heap.h"

Heap heap;

void program() {
    short int val = 23;
```

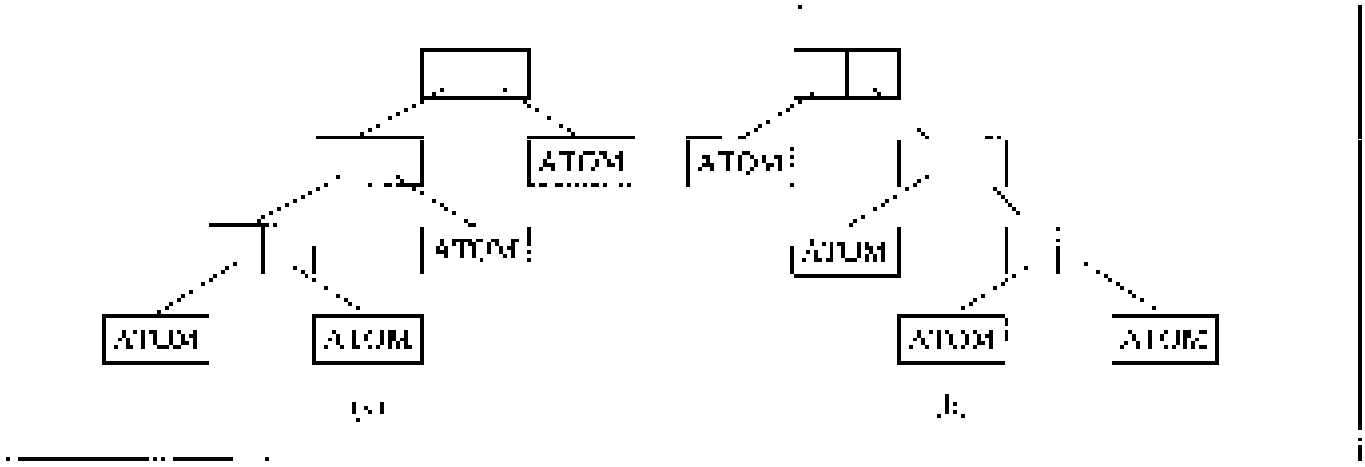
LISTING 12.16 *continued*

```
    int size, p, q, r;
    if (heapLocation == 1) { // call heapifyUsingRecursion(1);
        p = 0;
        r = 2;
    }
    else {
        r = randp1 % 10 + 1;
        p = randp1 % heap.numOfNodes(r); // guess by new num
        q = randp1 % heap.numOfNodes(r);
        r = randp1 % heap.numOfNodes(r);
    }
     if (r == 2)
        heap.el.setAsRoot(p, 99);
    else if (r == 4)
        heap.el.setAsLeaf(p, q, r);
    else if (r == 6)
        heap.el.setAsOneChild(p, q, r);
    else if (r == 8)
        heap.el.setAsTwoChild(p, q, r);
    else heap.el.remove(p);
    count += heap;
}

void main() {
    for (int i = 0; i < 50; i++)
        program();
}
```

12.6 EXERCISES

1. What happens if the best-fit method for swapping is implemented by block sort?
2. How does the "first heap to contain a given block" search method depend on the order of the sorted blocks? How can it be implemented if the blocks are not sorted?
3. The optimal-fitting heap algorithm is a block-allocation algorithm for searching a sample of blocks from the free list and the requested block. Use block swapping (Knuth 1973). What are the effects of the size of the sample on the performance of the algorithm? How does it compare to the first-fit and best-fit methods?

Figure 12.17 (a) Left degenerate tree (b) right degenerate list structures.



4. In what ways would a list-processing system be different if it had been implemented in segmented (rather than paged) memory? What are the additional costs and benefits?

5. What is the minimum number of bits that must be used in a tag field to allow the garbage collector to work correctly? Explain your answer.

6. Give an example of a pointer pointing to the middle of a word in the Dijkstra et al. tricolor marking algorithm.

7. Apply the Baker et al. algorithm to the left-degenerate and right-degenerate list structures in Figure 12.17. How many times will the pop() and push() procedures be executed in each case? Are there any differences? How would you optimize the algorithm for several list accesses of the same type?

8. In a reference counting garbage collection algorithm, a bit is usually associated with each memory word. This bit is set to 1 when the word is first allocated. When the reference count for the word is decremented to 0, the bit is reset to 0 and the word is returned to the free list. Explain why this bit is necessary. (*Hint:* Consider what happens when sweeping the heap.) Suggest a way to eliminate the bit. (*Hint:* Consider what happens when a cell is not pointed to by any other cell. Discuss the advantages and disadvantages of this garbage collection method.)

9. In Baker's algorithm, the copying performed by the collector must be invisible to the user program. Consider the following sequence: MOV R1,(R2) ; move the value pointed to by R2 into R1. Assume that R2 contains the memory address of a pointer to a list node in from space and R1 is to receive the address of the node to which the pointer points. Explain how the collector can ensure that the value placed in R1 is a pointer to the node in to space, rather than the node in from space. (*Hint:* Consider the case in which R2 contains the address of a pointer to the node in from space and the case in which R2 contains the address of a pointer to the node in to space.) How many copies of a node may be made per MOV request?

10. Explain the motivation of Baker's algorithm, which requires updating heap pointers in the case where the mutator accesses a trapped (Blelloch and Appel 1998). There is a problem with objects that may cross the page boundary. Suggest a solution to this problem.

12.7 Programming Assignments

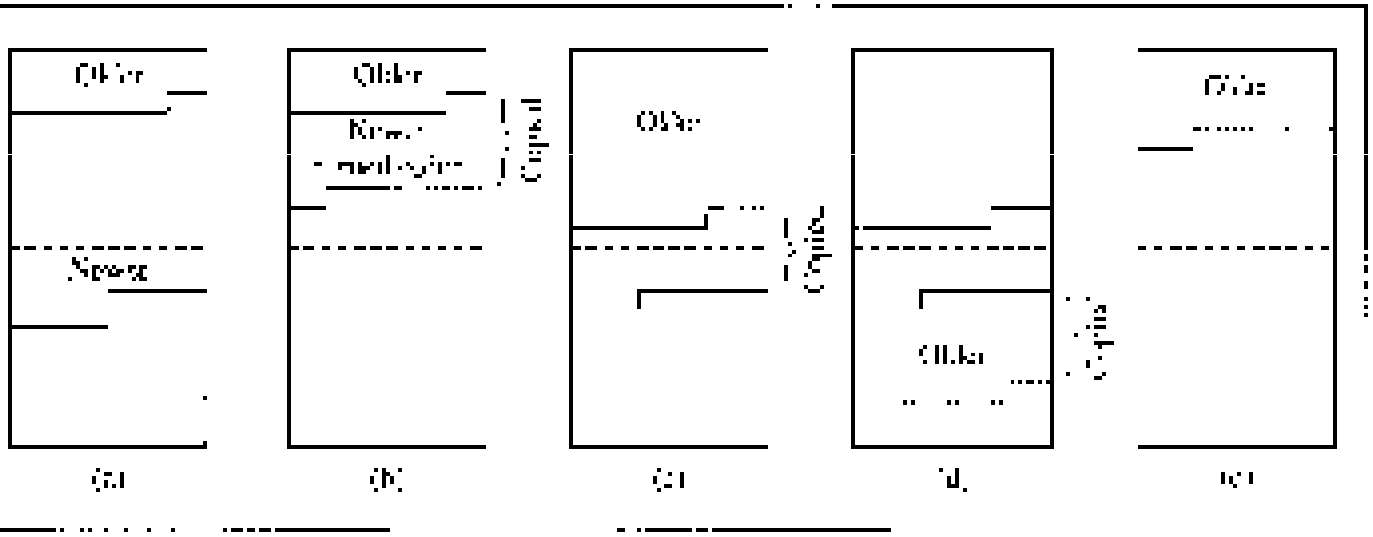
1. Implement the following memory allocation algorithm by writing a set of procedures for allocating memory from a large array. Your procedures should be called by a program that requests and releases memory by giving a number of bytes to be allocated or a pointer to a block to be released. Use a linked list of blocks of free memory. Each block contains a size field and a pointer to the next block. Initially, the list consists of one block containing all memory. Allocate memory from the first block that is large enough. Use the first part of the block to satisfy the request and the remainder to form a smaller free block. When memory is released, check to see if it is adjacent to one or two free blocks and, if so, combine them into a single larger block. Write a program to test your routines. First, allocate and release memory in random order. Next, write a program that allocates memory until it is exhausted. Measure the efficiency of your algorithm under various conditions. You might want to add a routine that prints a map of memory showing allocated and free areas.

2. In the binary buddy system, two pools of memory are maintained by the allocation routines. Both are the same size, which is a power of two. Initially, one pool is completely free and the other is completely allocated. Memory is allocated from the allocated pool until a request is made that cannot be satisfied. At this point, the free pool is divided in half, and one of the halves is divided in half, and so on, until enough free memory is available to satisfy the request. Each time a block is split, the two halves are known as buddies. When memory is freed, the buddy of the block being freed is checked to see if it is free. If so, the two buddies are combined to form a block twice as large. This process is repeated until the buddy of the block being freed is in use or the largest block is free. One way to implement the buddy system is to have a separate free list for each size block. Blocks can be split easily and buddies can be located quickly. Write a set of procedures to implement the buddy system. Compare its performance to that of the algorithm described in Exercise 1.

3. Implement a simple version of a generational garbage collector that uses only two regions (Appel 1989). The heap is divided into two even parts. The upper part holds cells which have been copied from the lower part. All cells in the lower part are free. The lower part is used for memory allocation and holds only newer cells (see Figure 12.18a). After it is full, the garbage collector cleans it by copying all reachable cells to the upper part (Figure 12.18b). Other work can then be done starting from the beginning of the lower part. After several such cycles, the upper part becomes full as well, and live cells being copied from the lower part can no longer be copied to the lower part (Figure 12.18c). In this case, the cleaning process of the upper part is begun by copying all reachable cells from the upper part to the lower part (Figure 12.18d), and then all reachable cells are copied to the beginning of the upper part (Figure 12.18e).

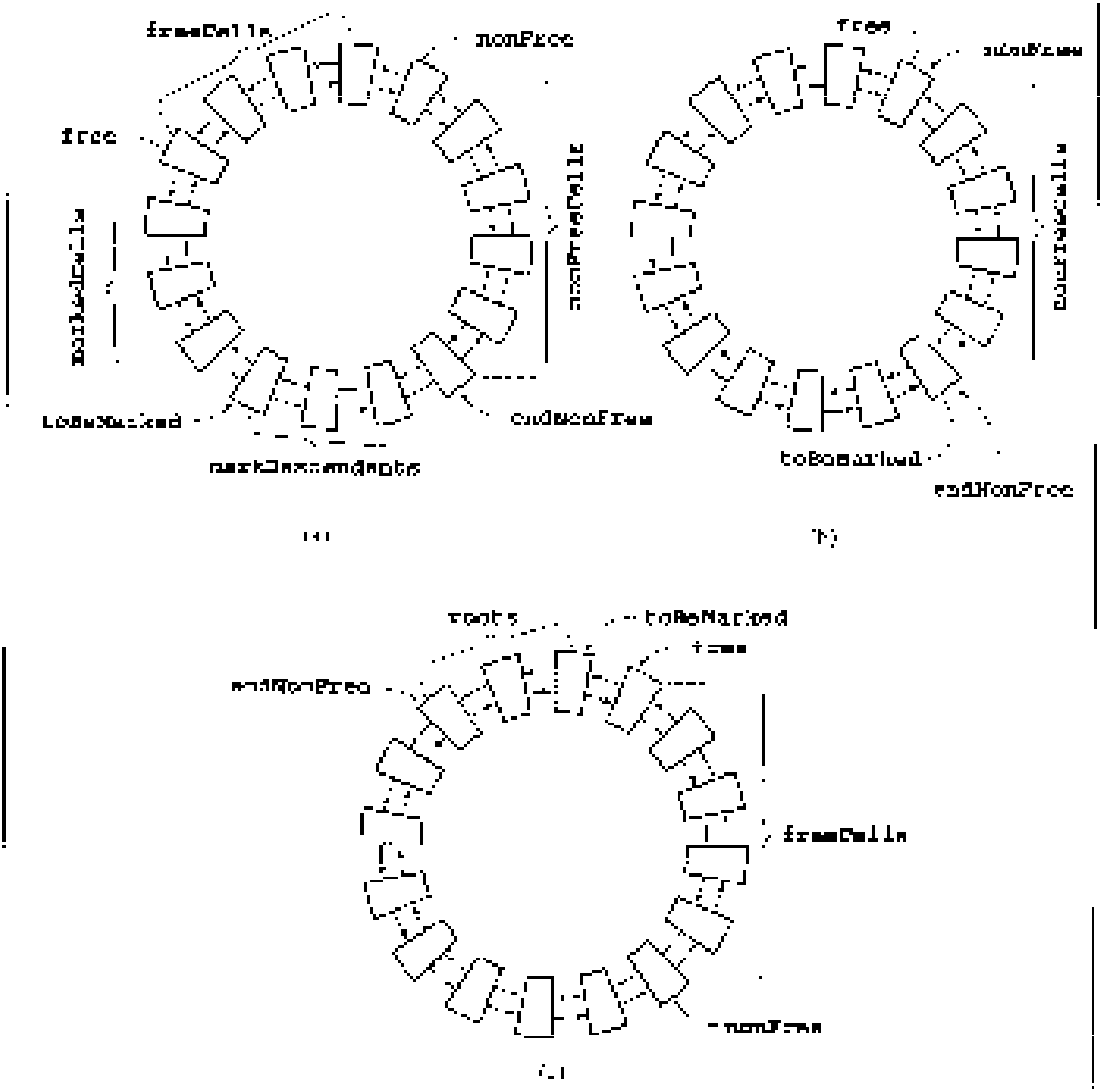
4. The case study presents an in-place incremental garbage collector. Modify the current one to become an incremental collector. In this case, pseudo-memory becomes active, which causes the function collect() to proceed a certain amount. To prevent the collector from encountering inconsistencies in structures possibly not completely processed by the mutator, the mutator should transfer any unmarked cells from the allocated list to the free list.

Figure 12.18 A heap with memory space for Appel's generational garbage collection.



Another very important modification is to maintain the promoted list, mentioned above. Here, instead of [illegible] we maintain [illegible] marked in the last collection. Then, instead of [illegible] we [illegible] marked [illegible] and [illegible] marked [illegible] [illegible] the list of live cells (Figure 12.19). In this way, we [illegible] and [illegible] the [illegible] by the [illegible]. Then, [illegible] we [illegible] [illegible] and [illegible] [illegible] through [illegible] (Figure 12.19). All [illegible] through [illegible] part of the [illegible] between [illegible] and [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible].

FIGURE 12.19 Heap treadmill.



Bibliography

Memory Management

Smith, Alan J., "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 473–530.

Stone, Harold S., *High-Performance Computer Architecture*, Reading, MA: Addison-Wesley, 1990, pp. 39–65.

Sequential-Fit Methods

Campbell, J. A., "Storage Organization and Garbage Collection," *Computer Journal*, Vol. 17, No. 4, November 1974, pp. 321–324.

Nonsequential-Fit Methods

Peterson, James L., and Norman, Theodore A., "Buddy Systems," *Communications of the ACM*, Vol. 20, No. 6, June 1977, pp. 421–431.

Ross, Douglas T., "The AED Free Storage Package," *Communications of the ACM*, Vol. 10, No. 8, August 1967, pp. 481–492.

Buddy Systems

Bromley, Allan G., "Comments on 'Storage Requirements of the Buddy-System Allocation,'" *Computer Journal*, Vol. 23, No. 1, February 1980, pp. 84–85.

Hinds, John A., and Greenberg, Bruce S., "An Implicit Boundary Tag Scheme for the Fibonacci Buddy System," *Communications of the ACM*, Vol. 18, No. 8, August 1975, pp. 451–452.

Hirschberg, Daniel S., "A Class of Dynamic Memory Allocation Algorithms," *Communications of the ACM*, Vol. 16, No. 10, October 1973, pp. 615–618.

Knowlton, Kenneth C., "A Fast Storage Allocator," *Communications of the ACM*, Vol. 8, No. 10, October 1965, pp. 623–625.

Page, Ivor, and Hagins, Jim, "Improving the Performance of Buddy Systems," *IEEE Transactions on Computers*, Vol. C-35, No. 5, May 1986, pp. 441–447.

Shen, Kenneth K., and Peterson, James L., "A Weighted Buddy Method for Dynamic Storage Allocation," *Communications of the ACM*, Vol. 17, No. 10, October 1974, pp. 558–562.

Garbage Collection

Appel, Andrew W., "Garbage Collection Can Be Faster Than Stack Allocation," *Information Processing Letters*, Vol. 25, No. 4, June 1987, pp. 275–279.

Appel, Andrew W., "Simple Generational Garbage Collection and Fast Allocation," *Software—Practice and Experience*, Vol. 19, No. 2, February 1989, pp. 171–183.

Baker, Henry G., "List Processing in Real Time on a Serial Computer," Communications of the ACM 21 (1978), 280–94.

Baker, Henry G., "Less-Instruction-Set Computer Architecture," ACM SIGPLAN Notices 27 (1992), 104–6, 108–10.

Brooks, Rodney A., "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, TX, 1984, 108–13.

Cheney, C. J., "A Nonrecursive List Compacting Algorithm," Communications of the ACM 13 (1970), 677–78.

Cohen, Jacques, "Garbage Collection of Linked Data Structures," Computing Surveys 13 (1981), 341–367.

Ellis, John R., Kai Li, and Andrew W. Appel, "Real-Time Concurrent Collection on Stock Multiprocessors," SRC Research Report 25 (1988), Nov. 1–11.

Fenichel, Robert R., and Jerome C. Yochelson, "A Lisp Garbage-Collector for Virtual-Memory Computer Systems," Communications of the ACM 12 (1969), 611–612.

Jones, Richard, and Rafael Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Chichester, 1996.

Kurokawa, Hiroshi, "A New Fast Garbage Collection Algorithm," Systems—Computers—Controls 1 (1970), 87–94.

Lieberman, Henry, and Carl Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," Communications of the ACM 26 (1983), 419–429.

Schorr, Herbert, and W. M. Waite, "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," Communications of the ACM 10 (1967), 501–506.

Wadler, Philip L., "Analysis of an Algorithm for Real Time Garbage Collection," Communications of the ACM 19 (1976), 491–500, 201, 1976, 120.

Weizenbaum, Joseph, "Symmetric List Processor," Communications of the ACM 6 (1963), 524–544.

Wilson, Paul R., "Uniprocessor Garbage Collection Techniques," Proceedings of the International Workshop on Memory Management, Springer-Verlag, 1992, —.

Yuasa, Taiichi, "Real-Time Garbage Collection on General-Purpose Machines," Journal of Systems and Software 11 (1990), 181–198.

Zorn, Benjamin, "Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection," Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, 1990, 87–98.

Computing Big-O

A.1 Harmonic Series

In some asymptotic analysis of algorithms, the expression H_n is used for the n^{th} harmonic number. Harmonic numbers are defined as the sum of the reciprocals of the first n positive integers, that is, $\sum_{i=1}^{n} \frac{1}{i}$. This is known as the harmonic series. The harmonic series can be approximated by the following formula:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon$$

where $0 < \varepsilon < \frac{1}{252n^6}$, and γ is Euler's constant (≈ 0.5772). This approximation can be very useful when working with algorithms, but it is rarely necessary in this form. Typically, we are only interested in the most significant term, so we can say that $H_n \in O(\ln n)$.

A.2 Approximation of the Function $\log(n!)$

The simplest approximation of $\log(n!)$ can be obtained by observing that each of the n terms in the product $n! = 1 \cdot 2 \cdot \cdots \cdot n$ is less than or equal to n. Thus, $n! \leq n^n$ for all $n \geq 1$, which implies that $\log(n!) \leq \log(n^n) = n \log n$. Therefore, $\log(n!)$ is an upper bound of $\log(n!)$, and $\log(n!) \in O(n \log n)$.

To obtain a lower bound for $\log(n!)$, the elements of the product $n!$ are grouped appropriately as

$$n! = 1 \cdot 2 \cdot \cdots \cdot \left\lceil \frac{n}{2} \right\rceil \cdot \cdots \cdot (n-1) \cdot n \geq \left\lceil \frac{n}{2} \right\rceil^{\lfloor n/2 \rfloor}$$

[illegible]

[illegible]

[illegible]

$$\text{[illegible]}$$

It follows that

$$\text{[illegible]}$$

or

$$\text{[illegible]}$$

This is transformed into

$$\text{[illegible]}$$

which is the same as (A.10). This completes the proof of the theorem.

It appears that one can randomly choose half of the elements to be the pivot. The reason is that the number of elements that are between the 25th and 75th percentiles is half the number of elements. Using this observation, we can assert that after a constant number of recursive calls, we are guaranteed to have a good pivot. Then we can conclude that the average number of comparisons used is $O(N)$. However, this analysis is left for the exercises.

A.3 Big-O for Average Case of Quicksort

In order to analyze the average case, we must make some assumption about the possible inputs of interest. Let us assume that we are sorting N distinct numbers, and that all $N!$ permutations of these numbers are equally likely. The following recursive relationship is then easy to verify:

$$\text{[illegible]}$$

The initial conditions $T(0) = T(1) = 0$ tell us that no comparisons are made when sorting zero or one element. For $N > 1$, we pick the pivot in one of N positions; all are equally likely. The first partitioning pass uses $N + 1$ comparisons. After the pivot is placed, if the left half has i elements, then the right half has $N - i - 1$ elements. Each of these sizes is also equally likely. Thus we obtain the formula

$$\begin{aligned}
T(N) &= \frac{1}{N}\left[\sum_{i=0}^{N-1} (T(i) + T(N-i-1))\right] + N + 1 \\
&= \frac{1}{N}\left[\sum_{i=0}^{N-1} T(i) + \sum_{i=0}^{N-1} T(N-i-1)\right] + N + 1 \\
&= \frac{2}{N}\sum_{i=0}^{N-1} T(i) + N + 1
\end{aligned}$$

or

$$\text{[illegible]}$$

To solve the recurrence, the summation operator is removed. First, a difference equation is subtracted from an equation obtained from it to get

$$\text{[illegible]}$$

resulting in

$$\text{[illegible]}$$

from which

$$\text{[illegible]}$$

This equation can be expanded, obtaining

$$\text{[illegible]}$$

from which

$$\text{[illegible]}$$

Note that H_{n-1} is a harmonic number. Using an approximation for this number (see Appendix A.1),

$$\text{[illegible]}$$

A.4 Average Path Length in a Random Binary Tree

In Chapter 6, an approximation is used for the average path length in a randomly created binary search tree. Assuming that

$$P_n(i) = \frac{(i-1)(P_{i-1}+1) + (n-i)(P_{n-i}+1)}{n}$$

this approximation is given by the recurrence relation

$$P_1 = 0$$

$$P_n = \frac{1}{n}\sum_{i=1}^{n} P_n(i) = \frac{1}{n^2}\sum_{i=1}^{n}\left((i-1)(P_{i-1}+1) + (n-i)(P_{n-i}+1)\right)$$

$$P_n = \frac{2}{n^2}\sum_{i=1}^{n-1} i(P_i+1) \qquad (1)$$

From this, we also have

$$P_{n-1} = \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i(P_i+1) \qquad (2)$$

After multiplying this equation by $\left(\frac{n-1}{n}\right)^2$ and subtracting the resulting equation from (1), we have

$$P_n = P_{n-1}\frac{(n-1)^2}{n^2} + \frac{2(n-1)(P_{n-1}+1)}{n^2} = \frac{n-1}{n^2}\left((n+1)P_{n-1} + 2\right)$$

After successive applications of this formula to each P_{n-i}, we have

$$P_n = \frac{n-1}{n^2}\left[(n+1)\frac{n-2}{(n-1)^2}\left[n\frac{n-3}{(n-2)^2}\left[(n-1)\frac{n-4}{(n-3)^2}\left[\cdots\frac{1}{2^2}(3P_1+2)\cdots\right]+2\right]+2\right]+2\right]$$

$$P_n = 2\left(\frac{n-1}{n^2} + \frac{(n+1)(n-2)}{(n-1)n^2} + \frac{(n+1)(n-3)}{n(n-1)(n-2)} + \frac{(n+1)(n-4)}{(n-1)(n-2)(n-3)} + \cdots + \frac{n+1}{2\cdot 3n}\right)$$

$$P_n = 2\left(\frac{n+1}{n}\right)\left(\sum_{i=1}^{n-1}\frac{n-i}{(n-i+1)(n-i+2)}\right) = 2\left(\frac{n+1}{n}\right)\left(\sum_{i=1}^{n-1}\left(\frac{2}{n-i+2} - \frac{1}{n-i+1}\right)\right)$$

$$P_n = 2\left(\frac{n+1}{n}\right)\frac{2}{n+1} + 2\left(\frac{n+1}{n}\right)\left(\sum_{i=1}^{n}\frac{1}{i} - 2\right) = 2\left(\frac{n+1}{n}\right)H_n - 4$$

So, P_n is $O(2 \ln n)$.

Algorithms in the Standard Template Library

B.1 STANDARD ALGORITHMS

To access these algorithms, the program has to include the statement

```
#include <algorithm>
```

In the two tables that follow, a phrase "elements in the range `[first, last)`" is an abbreviation of "elements indicated by iterators in the range `[first, last)`," or "elements reached by iterators from `first` up to, but not including, `last`."

Member Function	Operation
`iterator adjacent_find (first, last)`	Find the first pair of equal elements in the range `[first, last)` and return an iterator pointing to the element of the first pair. Return `last` if no duplicates found.
`iterator adjacent_find (first, last, f)`	Find the first pair of elements in the range `[first, last)` that satisfy the binary predicate `f` and return an iterator pointing to the position of the first element in the pair. Return `last` if no such pair is found.
`bool binary_search(first, last, value)`	Return `true` if binary search locates `value` in the range `[first, last)`. Assumes the elements are sorted.
`bool binary_search(first, last, value, f)`	Return `true` if binary search locates `value` in the range `[first, last)`. Assumes the elements are sorted. Function `f` is used to compare elements and `value`.
`iterator copy(first, last, result)`	Copy all the elements in the range `[first, last)` to `result` and return an iterator indicating the end of the range of copied elements.

`iterator copy_backward(first, last, result)`	Copies the elements in the range [first, last) into the range whose end is indicated by result and returns an iterator addressing the beginning of the range
`size_type count(first, last, value)`	Returns the number of the values in the range [first, last)
`size_type count_if(first, last, f())`	Returns the number of elements in the range for which the Boolean function f() returns true
`bool equal(first1, last1, first2)`	Compares the range [first1, last1) and the range starting at the same length and returns true if the corresponding elements in the two ranges are equal
`bool equal(first1, last1, first2, f())`	Compares the range [first1, last1) and the range starting at the same length which begins at the position indicated by the iterator first2 and returns true if the ranges contain similar elements where similarity is determined by a user-supplied Boolean function f(); returns false otherwise
`pair<iterator, iterator> equal_range(first, last, value)`	Returns a pair of iterators that delimit the subrange of the range [first, last) in the form of a pair type whose first member is the lower bound of value and whose second member is the upper bound of value, if such a subrange is found; if not found, returns pairs whose both iterators equal to first
`pair<iterator, iterator> equal_range(first, last, value, f())`	Returns a pair of iterators that delimit the subrange of the range [first, last) in the same manner as the equal_range(first, last, value) function, except that the Boolean function f() is used to determine whether the elements are equal to value
`void fill(first, last, value)`	Assigns value to all elements in the range [first, last)
`void fill_n(first, n, value)`	Assigns value to n elements in the range [first, first + n)
`iterator find(first, last, value)`	Returns an iterator to the first occurrence of value in the range [first, last); returns last if such an occurrence is not found
`iterator find_end(first1, last1, first2, last2)`	Finds the last occurrence in the range [first1, last1) that matches the subsequence [first2, last2), and returns an iterator addressing the beginning of the subrange of elements that are in the sequence [first1, last1); returns last1 if no such subrange is found
`iterator find_end(first1, last1, first2, last2, f())`	Finds the last occurrence in the range [first1, last1) that matches the subsequence [first2, last2) in the sense that the Boolean function f() returns true, and returns an iterator addressing the beginning of the subrange of elements that are in the sequence [first1, last1); returns last1 if no such subrange is found
`iterator find_first_of(first1, last1, first2, last2)`	Finds the first occurrence in the range [first1, last1) of any element that is in the sequence [first2, last2); returns last1 if no such element is found
`iterator find_first_of(first1, last1, first2, last2, f())`	Finds the first occurrence in the range [first1, last1) of any element that is in the sequence [first2, last2) in the sense that the Boolean function f() returns true; returns last1 if no such element is found
`iterator find_if(first, last, f())`	Returns an iterator to the first element in the range [first, last) for which the Boolean function f() returns true; returns last if such an element is not found

`function for_each(first, last, t);`	apply the function `f(t)` to all the elements in the range `[first, last)`, and return the function
`void generate(first, last, f);`	fill the elements in the range `[first, last)` with the values returned by successive calls to `f()`; the algorithm returns `void`
`void generate_n(first, n, f);`	fill the elements in the range `[first, first + n)` with the values returned by the function `f()`; `n` must be non-negative
`bool includes(first1, last1, first2, last2);`	return `true` if the elements in the sorted range `[first1, last1)` are included in the sorted range `[first2, last2)`; use the `<` operator for comparisons in ascending order
`bool includes(first1, last1, first2, last2, f);`	return `true` if the elements in the sorted range `[first1, last1)` are included in the sorted range `[first2, last2)`; use `f` for comparisons in ascending order (`f` returns `true` or `false`)
`void inplace_merge(first, middle, last);`	put the first two consecutive ranges `[first, middle)` and `[middle, last)` into `[first, last)`; use `<` for comparisons in ascending order
`void inplace_merge(first, middle, last, f);`	put the first two consecutive ranges `[first, middle)` and `[middle, last)` into `[first, last)`; use `f` for comparisons in ascending order
`void iter_swap(i1, i2);`	swap the elements `i1` and `i2`
`bool lexicographical_compare(first1, last1, first2, last2);`	return `true` if the range `[first1, last1)` is lexicographically less than the range `[first2, last2)`
`bool lexicographical_compare(first1, last1, first2, last2, f);`	return `true` if the range `[first1, last1)` is lexicographically less than the range `[first2, last2)`; use `f` for comparisons
`iterator lower_bound(first, last, value);`	return an iterator to the first element in the sorted range `[first, last)` that is greater than or equal to `value`; use `<` for comparisons in ascending order; return `last` if `value` is greater than all elements
`iterator lower_bound(first, last, value, f);`	return an iterator to the first element in the sorted range `[first, last)` that is greater than or equal to `value`; use `f` for comparisons in ascending order; return `last` if `value` is greater than all elements
`void make_heap(first, last);`	make the elements in the range `[first, last)` into a heap; use `<` for comparisons
`void make_heap(first, last, f);`	make the elements in the range `[first, last)` into a heap; use the function `f()` for comparisons
`const T& max(x, y)`	return the greater of the two values `x` and `y`
`const T& max(x, y, f);`	return the greater of the two values `x` and `y`; determine this by the function `f()`
`iterator min_element(first, last);`	return an iterator indicating the position of the smallest element in the range `[first, last)`
`iterator min_element(first, last, f);`	return an iterator indicating the position of the smallest element in the range `[first, last)`; use `f` for comparisons
`void merge(first1, last1, first2, last2, result);`	put the results of merging the ranges `[first1, last1)` and `[first2, last2)` into `result` in ascending order

`void assign(first1, last1, first2, last2, assign_f);`	Assigns each element in the range [first1, last1) to the corresponding element in the range [first2, last2) for which assign_f(*first1, *first2) returns true.
`const T& min(x, y);`	Returns the minimum of the elements x and y.
`const T& min(x, y, <);`	Returns the minimum of the elements x and y, determined by the comparison function <.
`iterator min_element(first, last);`	Returns an iterator pointing to the smallest element in the range [first, last).
`iterator min_element(first, last, <);`	Returns an iterator pointing to the position of the smallest element in the range [first, last), as determined by the function <.
`pair<iterator, iterator> mismatch(first1, last1, first2);`	Compares elements in the range [first1, last1) with those in the range [first2, first2 + (last1 - first1)), and returns a pair of iterators that point to the first positions in the ranges at which values differ.
`pair<iterator, iterator> mismatch(first1, last1, first2, ==);`	Compares elements in the range [first1, last1) with those in the range [first2, first2 + (last1 - first1)), and returns a pair of iterators that point to the first positions in the ranges at which values differ, as determined by the function ==.
`bool next_permutation(first, last);`	Generates the next permutation that is greater than the current one in the range [first, last), and returns true if such a permutation exists and false otherwise.
`bool next_permutation(first, last, <);`	Generates the next permutation that is greater than the current one in the range [first, last), as determined by the function <, and returns true if such a permutation exists and false otherwise.
`void nth_element(first, nth, last);`	Rearranges the elements in the range [first, last) so that the element pointed to by nth is the same as the element that would appear there if the range were sorted, and so that no element in the range [nth, last) is smaller than any element in the range [first, nth).
`void nth_element(first, nth, last, <);`	Rearranges the elements in the range [first, last) so that the element pointed to by nth is the same as the element that would appear there if the range were sorted according to the function <, and so that no element in the range [nth, last) is smaller than any element in the range [first, nth).
`void partial_sort(first, middle, last);`	Rearranges the range [first, last) so that the elements in the range [first, middle) are sorted in ascending order, and the remaining elements in the range [middle, last) are in an unspecified order.
`void partial_sort(first, middle, last, <);`	Rearranges the range [first, last) so that the elements in the range [first, middle) are sorted according to the function <, and the remaining elements in the range [middle, last) are in an unspecified order.
`iterator partial_sort_copy(first1, last1, first2, last2);`	Copies the range [first1, last1) to the range [first2, first2 + min(last1 - first1, last2 - first2)), and sorts the elements in the range [first2, last2) in ascending order. Returns an iterator pointing to the position one element beyond the last element sorted.

`ForwardIterator partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last);`	Copies the smaller of `result_last - result_first` elements from the range `[first, last)` to the range `[result_first, result_last)`, in ascending order determined by the operator `<` or the function object `comp`. Returns an iterator to the position one past the last copied element.
`bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);`	Returns `true` if all the elements in the range `[first, last)` that satisfy the predicate `pred` appear before all the elements that don't. Returns `true` if the range is empty.
`ForwardIterator partition(ForwardIterator first, ForwardIterator last, Predicate pred);`	Rearranges the elements in the range `[first, last)` so that all the elements for which the predicate `pred` returns `true` appear before all the elements for which it returns `false`. Returns an iterator `i` such that `pred(*j)` is `false` for every iterator `j` in the range `[i, last)`.
`void sort_heap(RandomAccessIterator first, RandomAccessIterator last);`	Sorts the elements in the heap `[first, last)` in ascending order. Assumes that the range `[first, last)` is a heap.
`void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`	Sorts the elements in the heap `[first, last)` in the order specified by `comp`. Assumes that the range `[first, last)` is a heap ordered by `comp`.
`bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);`	Transforms the range `[first, last)` into the next permutation, as defined by lexicographical ordering. Returns `false` if the new permutation is lexicographically smaller than the previous one, `true` otherwise.
`bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);`	Transforms the range `[first, last)` into the next permutation, as defined by lexicographical ordering using `comp`. Returns `false` if the new permutation is lexicographically smaller than the previous one, `true` otherwise.
`void pop_heap(RandomAccessIterator first, RandomAccessIterator last);`	Swaps the values in `*first` and `*(last - 1)` and then makes `[first, last - 1)` into a heap.
`void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`	Swaps the values in `*first` and `*(last - 1)` and then makes `[first, last - 1)` into a heap ordered by `comp`.
`void push_heap(RandomAccessIterator first, RandomAccessIterator last);`	Makes `[first, last)` into a heap. Assumes that `[first, last - 1)` is a heap.
`void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`	Makes `[first, last)` into a heap ordered by `comp`. Assumes that `[first, last - 1)` is a heap ordered by `comp`.
`void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);`	Randomly rearranges the elements in the range `[first, last)`. Assumes that the length of the range is less than the value of the random number generator.
`void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand);`	Randomly rearranges the elements in the range `[first, last)`. Assumes that the length of the range is less than the value returned by the random number generator `rand`.
`ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);`	Removes from the range `[first, last)` all the elements that are equal to `value`. Returns an iterator that indicates the end of the resulting range.
`OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, const T& value);`	Copies from the range `[first, last)` all the elements that are not equal to `value`. Returns an iterator that indicates the end of the resulting range.
`ForwardIterator remove_copy_if(ForwardIterator first, ForwardIterator last, OutputIterator result, Predicate pred);`	Copies from the range `[first, last)` all the elements for which the predicate `pred` returns `false`. Returns an iterator that indicates the end of the resulting range.
`ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, Predicate pred);`	Removes from the range `[first, last)` all the elements for which the predicate `pred` returns `true`. Returns an iterator that indicates the end of the new range.
`void replace(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);`	Replaces all occurrences of `old_value` in the range `[first, last)` with `new_value`.

`OutputIterator replace_copy(FirstIt, LastIt, result, oldValue, newValue)`	copy elements from the range [first, last) to the sequence beginning at result, and replace all occurrences of oldValue in the new range with newValue; return an iterator indicating the end of the new range
`OutputIterator replace_copy_if(FirstIt, LastIt, result, f(), value);`	copy elements from the range [first, last) to the sequence beginning at result, and replace in the new sequence elements for which the predicate f() is true with value; return an iterator indicating the end of the new range
`void replace_if(First, last, f(), value);`	replace all elements in the range [first, last) for which the predicate f() is true with value
`void reverse(BidirIt, BidirIt);`	reverse the order of elements in the range [first, last)
`OutputIterator reverse_copy(first, last, result);`	copy in reverse order the elements from the range [first, last) to the range that starts at result; return an iterator indicating the end of the copied sequence
`void rotate(First, middle, last);`	rotate the elements in the range [first, last), by middle – first positions
`OutputIterator rotate_copy(first, middle, last, result);`	rotate the elements in the range [first, last), as middle – first positions and place the new sequence in the range that starts at result
`Iterator search(First1, Last1, First2, Last2);`	search the range [first2, last2) for the range [first1, last1); return an iterator indicating the beginning of the subsequence, or last1 if the subsequence is not found; uses operator== to compare elements
`IterType search_n(First, last, n, value, f());`	search the range [first, last) for n consecutive elements equal to value and return an iterator indicating the beginning of the subsequence, or last if no such subsequence is found; uses f() to compare elements with value
`OutputIterator set_difference(First1, last1, First2, last2, result);`	starting at position result, place in ascending order the elements that belong to the range [first1, last1) but not to the range [first2, last2); return an iterator indicating the end of the resulting range
`OutputIterator set_difference(First1, last1, First2, last2, result, f());`	starting at position result, place in ascending order the elements in the range [first1, last1) that do not appear in the range [first2, last2); return an iterator indicating the end of the resulting range.

`iterator set_intersection(first1, last1, first2, last2, result);`	Copies the elements that are in both sorted ranges [first1, last1) and [first2, last2) to the range beginning at result, in sorted order. Returns an iterator to the end of the result range.
`iterator set_intersection(first1, last1, first2, last2, result, f);`	Same as the previous version, but uses the function f to compare elements. Returns an iterator to the end of the result range.
`iterator set_symmetric_difference(first1, last1, first2, last2, result)`	Copies the elements that are in one of the sorted ranges [first1, last1) and [first2, last2) but not in both to the result range, in sorted order. Returns an iterator to the end of the resulting range.
`iterator set_symmetric_difference(first1, last1, first2, last2, result, f);`	Same as the previous version, but uses the function f to compare elements. Returns an iterator to the end of the resulting range.
`iterator set_union(first1, last1, first2, last2, result);`	Copies the union of the sorted ranges [first1, last1) and [first2, last2) to the result range, in sorted order. Returns an iterator to the end of the resulting range.
`iterator set_union(first1, last1, first2, last2, result, f);`	Same as the previous version, but uses the function f to compare elements. Returns an iterator to the end of the resulting range.
`void sort(first, last);`	Sorts the elements in the range [first, last).
`void sort(first, last, f);`	Same as the previous version, but uses the function f to compare elements.
`void sort_heap(first, last);`	Sorts the elements in the heap [first, last).
`void sort_heap(first, last, f);`	Same as the previous version, but uses the function f to compare elements.
`iterator stable_partition(first, last, f);`	Reorders the elements in the range [first, last) so that all the elements for which the function f returns true precede the elements for which it returns false. Returns an iterator to the first element for which f returns false.
`void stable_sort(first, last)`	Sorts the elements in the range [first, last) in ascending order while preserving the relative order of equal elements.
`void stable_sort(first, last, f);`	Same as the previous version, but uses the function f to determine the relative order of elements.
`void swap(x, y)`	Swaps the values of x and y.
`iterator swap_ranges(first1, last1, first2);`	Swaps corresponding elements in the range [first1, last1) and the range [first2, first2+last1-first1), and returns the iterator first2+last1-first1.

`iterator transform(first, last, result, f);`	transforms the elements in the range [first, last) by calling the function object f and replaces the original elements with the transformed elements; returns an iterator indicating the end of the result range
`iterator transform(first1, last1, first2, result, f);`	applies the binary function object f to the corresponding elements in the ranges [first1, last1) and [first2, first2 + (last1 - first1)) and places the results in the range beginning at result; returns an iterator indicating the end of the result range
`iterator unique(first, last);`	removes all duplicates in the range [first, last) that is in ascending order; returns an iterator indicating the end of the possibly reduced range
`iterator unique(first, last, f);`	removes all elements from the range [first, last) that are ordered by the relation f; returns an iterator indicating the end of the possibly reduced range
`iterator unique_copy(first, last, result);`	copies the range [first, last) to the range beginning at result, except that in any group of equal elements only the first one is copied; returns an iterator to the end of the result range
`iterator unique_copy(first, last, result, f);`	copies the range [first, last) to the range beginning at result, except that in any group of elements for which the relation f holds for each pair of adjacent elements only the first one is copied; returns an iterator to the end of the result range
`iterator upper_bound(first, last, value);`	returns an iterator to the furthermost rightmost position in the range [first, last) that it is possible to insert value before without violating the ordering; returns last if value is larger than all the elements
`iterator upper_bound(first, last, value, f);`	returns an iterator to the furthermost rightmost position in the range [first, last) that it is ordered by the relation f before which value can be inserted without violating the ordering; returns last if value is larger than all the elements

A class to encapsulate binary operations on numbers

`#include <functional>`

Member Function	Operation
`T accumulate(first, last, value);`	returns value plus the sum of the values in the range indicated by the iterators first and last
`T accumulate(first, last, sum, op);`	returns the result of applying the two-argument operation op to value and to all the values in the range indicated by the iterators first and last

`iterator adjacent_difference(first, last, result);`	`x = *first; while (++first != last) { y = *first; *++result = y - x; x = y; } return ++result;`
`iterator adjacent_difference(first, last, result, op);`	`x = *first; while (++first != last) { y = *first; *++result = op(y, x); x = y; } return ++result;`
`T inner_product(first1, last1, first2, value);`	`while (first1 != last1) value = value + (*first1++ * *first2++); return value;`
`T inner_product(first1, last1, first2, value, op1, op2);`	`while (first1 != last1) value = op1(value, op2(*first1++, *first2++)); return value;`
`iterator partial_sum(first, last, result);`	`*result = *first; while (++first != last) { *(result+1) = *result + *first; ++result; } return ++result;`
`iterator partial_sum(first, last, result, op);`	`*result = *first; while (++first != last) { *(result+1) = op(*result, *first); ++result; } return ++result;`

Name Index

Abdi, Hervé, 152
Abdi-Nejad, Shirin, 204
Adams, R. J., 412
Aiken, L. S., 381, 412
Allen, B., 264, 302
Al-Fawzan, M., 355, 374
Appel, A. G., 207, 465, 470, 621, 624
Arabie, P., 213
Archambeau, N., 64
Ben-Akiva, M. E., 299
Bentler, P. M., 203, 404, 411, 412, 620, 624, 634
Bernstein, I. H., 213
Bock, R. D., 213
Bollen, K. A., 204
Bonifay, W. E., 204, 370
Borg, I., 421
Borhani, F., 314
Brandstätter, E., 84, 85, 86
Breckler, S. J., 204
Brown, R. L., 261, 306
Browne, M. W., 213
Bryant, F. B., 64, 65, 66
Bulmer, M., 204
Burnham, K. P., 262, 263
Cattell, R. B., 84, 85, 370
Chan, D., 374
Chen, F., 204
Cheung, G. W., 374
Chou, C.-P., 411
Cohen, J., 342
Comrey, A. L., 208, 209, 213
Cudeck, R., 30, 440
Curran, P. J., 411, 412, 621
Dalgleish, L. I., 204
Davison, M. L., 213, 342, 344
Dawis, R. V., 427, 428, 429, 430
Depaoli, S., 204
Diamond, J., 64
Dolan, C. V., 374
Donnellan, M. B., 204
Dunn, G., 204
Dunn, J. E., 342
Dziuban, C. D., 261, 264, 302, 413, 414
Eaton, M. L., 204
Edwards, M. C., 204
Embretson, S. E., 204
Enders, C. K., 64, 65, 66
Fabrigar, L. R., 204
Fan, X., 204
Finch, J. F., 411, 412
Floyd, F. J., 77, 229, 235, 239, 300, 442, 447
Fox, J., 299
Gerbing, D. W., 342
Gerlach, R. W., 261, 264, 302, 413, 414
Gessaroli, M. E., 204, 302
Gibbons, R. D., 204
Gifford, D. D., 204
Gilliam, S., 204
Gorsuch, R. L., 213
Graham, J. M., 64
Green, S. B., 204
Grice, J. W., 204
Gribbin, C., 204
Guadagnoli, E., 213
Gustafsson, J.-E., 374
Hakstian, A. R., 213
Hancock, G. R., 204
Harman, H. H., 213
Harrington, D., 204
Hatcher, L., 204
Hayashi, K., 204
Hayduk, L. A., 204
Hershberger, S. L., 204
Hetzel, L., 204
Hoelter, J. W., 204
Hogarty, K. Y., 213
Hong, S., 204
Horn, J. L., 213
Hoyle, R. H., 204
Hu, L.-T., 204, 411, 412
Hubbard, R., 342
Hubert, L., 213
Hull, D. M., 204
Humphreys, L. G., 213
Jackson, D. L., 204
Jaccard, J., 299
Jöreskog, K. G., 204, 411, 412
Kaiser, H. F., 213
Kano, Y., 204
Kelley, K., 204
Kenny, D. A., 204
Kim, J.-O., 213
Kline, R. B., 204
Knapp, T. R., 213
Kolenikov, S., 204
Kroonenberg, P. M., 204
Lance, C. E., 204
Lee, S.-Y., 204
Lei, P.-W., 204
Levy, R., 204
Little, T. D., 64, 65, 66
Liu, Y., 204
Loehlin, J. C., 204
Long, J. S., 204
Lubinski, D., 213
MacCallum, R. C., 204, 213, 411, 412, 440
Marcoulides, G. A., 204
Marsh, H. W., 204, 374
Maruyama, G., 204
Mashburn, A. J., 204
Mathieu, J. E., 204
McArdle, J. J., 204
McDonald, R. P., 204, 213, 411, 412, 440
McIntosh, A. R., 204
Meade, A. W., 204
Meehl, P. E., 342
Mehta, P. D., 204
Meredith, W., 204
Millsap, R. E., 204
Mooijaart, A., 204
Mulaik, S. A., 204
Muthén, B. O., 204
Muthén, L. K., 204
Nesselroade, J. R., 204
Nevitt, J., 204
Nunnally, J. C., 213
Olsson, U. H., 204
Pedhazur, E. J., 213
Peterson, C. B., 204, 205, 206, 440
Preacher, K. J., 204
Price, L. R., 204
Raiche, G., 204
Rajaratnam, N., 213
Rencher, A. C., 213
Revelle, W., 204
Rhemtulla, M., 204
Rigdon, E. E., 204
Rindskopf, D., 204
Rizopoulos, D., 204
Rogers, J. L., 204
Rosseel, Y., 204
Rovine, M. J., 204
Rozeboom, W. W., 204
Rubin, D. B., 64
Saris, W. E., 204
Satorra, A., 204
Schafer, J. L., 64
Schmitt, N., 204
Schumacker, R. E., 204
Schwarz, G., 204
Shapiro, A., 204
Shrout, P. E., 204
Silvia, E. S. M., 204
Smith, G. T., 204
Snook, S. C., 213
Sörbom, D., 204
Spearman, C., 213
Spector, P. E., 204
Stapleton, L. M., 204
Steiger, J. H., 204, 213, 411, 412, 440
Stevens, J. P., 213
Stine, R., 204
Sturgis, P., 204
Tanaka, J. S., 204
Tatsuoka, M. M., 213
Taylor, A. B., 204
Tellegen, A., 204
Thissen, D., 204
Thompson, B., 204, 213, 342
Thurstone, L. L., 213
Tucker, L. R., 213
Velicer, W. F., 204, 213
Vernon, P. E., 213
Vigneau, F., 204
Wall, M. M., 204
Wang, L., 204
Wegener, D. T., 204
Weng, L.-J., 204
West, S. G., 204
Widaman, K. F., 204, 374
Wilcox, R. R., 204
Wilkinson, L., 204
Williams, L. J., 204
Wothke, W., 204
Wright, A. G. C., 204
Yates, A., 213
Yuan, K.-H., 204
Yung, Y.-F., 204
Zhang, G., 204
Zhang, Z., 204
Zinbarg, R. E., 204
Zwick, W. R., 213

Higgins, [illegible]
Hoyle, [illegible]
Hill, [illegible]
Hoffman, [illegible]
Hudson, [illegible]
Hansen, [illegible]
Hermann, [illegible]
Jones, [illegible]
Johnson, [illegible]
Kelly, [illegible]
Kaplan, [illegible]
Kennedy, [illegible]
Klein, [illegible]
Lewis, [illegible]
Lopez, [illegible]
Lee, [illegible]
Meyer, [illegible]
Miller, [illegible]
Nelson, [illegible]
Nguyen, [illegible]
Olsen, [illegible]
Ortiz, [illegible]
Patel, [illegible]
Perez, [illegible]
Phillips, [illegible]
Quinn, [illegible]
Reed, [illegible]
Rivera, [illegible]
Roberts, [illegible]
Sanchez, [illegible]
Schmidt, [illegible]
Singh, [illegible]
Smith, [illegible]
Taylor, [illegible]
Thomas, [illegible]
Turner, [illegible]
Vargas, [illegible]
Walker, [illegible]
Wang, [illegible]
Weber, [illegible]
Young, [illegible]
Zhang, [illegible]

Abercrombie, A. [illegible]
Ackroyd, [illegible]
Adams, [illegible]
Addison, [illegible]
Allen, [illegible]

Anderson, [illegible]
Angell, [illegible]
Archer, [illegible]
Armitage, [illegible]
Armstrong, [illegible]
Ashcroft, [illegible]
Atkinson, [illegible]
Austin, [illegible]

Bailey, [illegible]
Baker, [illegible]
Barnes, [illegible]
Barrett, [illegible]
Bates, [illegible]
Bell, [illegible]
Bennett, [illegible]
Berry, [illegible]
Bishop, [illegible]
Blackwell, [illegible]
Blake, [illegible]
Bond, [illegible]
Boyle, [illegible]
Brady, [illegible]
Brooks, [illegible]
Brown, [illegible]
Bryant, [illegible]
Burke, [illegible]
Burton, [illegible]
Butler, [illegible]

Campbell, [illegible]
Carpenter, [illegible]
Carter, [illegible]
Chapman, [illegible]
Clark, [illegible]
Cole, [illegible]
Collins, [illegible]
Cook, [illegible]
Cooper, [illegible]
Cox, [illegible]
Crawford, [illegible]
Cunningham, [illegible]

Davies, [illegible]
Davis, [illegible]
Dawson, [illegible]
Day, [illegible]
Dixon, [illegible]
Duncan, [illegible]

Edwards, [illegible]
Elliott, [illegible]
Ellis, [illegible]
Evans, [illegible]

Fisher, [illegible]
Fletcher, [illegible]
Ford, [illegible]
Foster, [illegible]
Fox, [illegible]
Francis, [illegible]
Frost, [illegible]

Gardner, [illegible]
George, [illegible]
Gibson, [illegible]
Gill, [illegible]
Graham, [illegible]
Grant, [illegible]
Gray, [illegible]
Green, [illegible]
Griffiths, [illegible]

Hall, [illegible]
Hamilton, [illegible]
Harris, [illegible]
Harrison, [illegible]
Hart, [illegible]
Hawkins, [illegible]
Henderson, [illegible]
Hill, [illegible]
Holmes, [illegible]
Howard, [illegible]
Hughes, [illegible]
Hunt, [illegible]

Jackson, [illegible]
James, [illegible]
Jenkins, [illegible]
Johnson, [illegible]
Jones, [illegible]

Kelly, [illegible]
Kennedy, [illegible]
King, [illegible]
Knight, [illegible]

Lambert, [illegible]
Lawrence, [illegible]
Lee, [illegible]
Lewis, [illegible]
Long, [illegible]

Marshall, [illegible]
Martin, [illegible]
Mason, [illegible]
Matthews, [illegible]
Miller, [illegible]
Mills, [illegible]
Mitchell, [illegible]
Moore, [illegible]
Morgan, [illegible]
Morris, [illegible]
Murphy, [illegible]

Nelson, [illegible]
Newman, [illegible]
Nicholson, [illegible]

Owen, [illegible]

Palmer, [illegible]
Parker, [illegible]
Patterson, [illegible]
Payne, [illegible]
Perkins, [illegible]
Peters, [illegible]
Phillips, [illegible]
Porter, [illegible]
Powell, [illegible]
Price, [illegible]

Quinn, [illegible]

Reed, [illegible]
Reynolds, [illegible]
Richards, [illegible]
Roberts, [illegible]
Robertson, [illegible]
Robinson, [illegible]
Rogers, [illegible]
Ross, [illegible]
Russell, [illegible]

Sanders, [illegible]
Scott, [illegible]
Shaw, [illegible]
Simpson, [illegible]
Smith, [illegible]
Spencer, [illegible]
Stevens, [illegible]
Stewart, [illegible]
Sullivan, [illegible]

Taylor, [illegible]
Thomas, [illegible]
Thompson, [illegible]
Turner, [illegible]

Vaughan, [illegible]

Walker, [illegible]
Wallace, [illegible]
Walsh, [illegible]
Ward, [illegible]
Watson, [illegible]
Webb, [illegible]
Wells, [illegible]
West, [illegible]
White, [illegible]
Williams, [illegible]
Wilson, [illegible]
Wood, [illegible]
Wright, [illegible]

Young, [illegible]

Zimmerman, [illegible]

Subject Index

Abstract data types, 1
Accessibility, 203–204
Ackermann's function, 148
Activation record, 244–245
Adaptive recursive quicksort, 214
Adjacency list representation, 355–359
Administrator, 398
Adjacency, 347
Adjacency matrix, 357–374
Adjacent vertices, 372
Algorithm, 26
Alternating path, 415
Amortized complexity, 56–60, 265–266, 268–269
Anchor, 171–172, 176, 187
Arc, 216, 371
Articulation point, 397
Assignment problem, 421
Asymptotic notation, 47
Augmenting path, 414
Average path length, 233–235
AVL tree, 235 and 333–334

Bar Ilan, 365
Base type, 105
Bellman-Ford, 192, 194
Binary search, 232–239
Binary search tree, 329
Binary tree, 327
Bin-packing, 6
Bipartite graph, 407
Bipartite matching, 407–409
Biconnected component, 397
Big O notation, 34–39
Big omega notation, 38–39
Binary heap sort, 339–340
Binary search, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139,

Thin interpolation splines, 496–497
Thinning, 500–501
Three-dimensional graphics, 9, 11, 13–14, 16–17, 19–20, 22–23, 25–26, 28–29, 31–32, 34–35, 37–38, 40–41, 43–44, 46–47, 49–50, 52–53, 55–56, 58–59, 61–62, 64–65, 67–68, 70–71, 73–74, 76–77, 79–80, 82–83, 85–86, 88–89, 91–92, 94–95, 97–98, 100–101, 103–104, 106–107, 109–110, 112–113, 115–116, 118–119, 121–122, 124–125, 127–128, 130–131,

Assumed perfect hash function, 315
Minimum cost, 447
Minimum spanning tree, 561–566
Modular decomposition, 204–212
Monotone decision tree, 52
Monotone subsequence, 62–63
Multigraph, 87
Multiple pattern matching, 196
multiplication, 921
multiplier, 921
Multiplying, 922

Nearest-neighbor graph, 784
Negative cycle, 438
Network, 624
Newton's method, 888
Nonlinear programming, 624
(see also convex programming)
Normal vector, 826
NP-complete problem, 1021–1048
NP-hard problem, 1021–1048
Null pointer, 201

Object, 7
Oriented depth-first search forest, 200
Optimization, 314–340
optimal binary search tree, 331
Optimal substructure, 324
Optimization problem, 314
Ordered binary tree, 215
Ordering method, 100–103
Orientation, 815

P, 1021–1048
parenthesization, 338
partition function, 211–215, 224
Path, 87
Perfect hash function, 307–318
Permutation, 49
Perturbation, 829
Point, 781
Pointer, 21
Point quadtree, 296
Polar coordinates, 779–787
Polygon, 806
Polynomial, 904–907
Postfix notation, 219
Power series, 911
Powers of 2, 20
Prefix function, 1000–1001
Prefix notation, 340
Prefix property, 344
Prim–Dijkstra algorithm, 578–582, 587–588, 590
Primality testing, 1048–1049
Prime number, 1048
Principle of optimality, 324
Priority queue, 554
Probabilistic analysis, 52
Probability, 1048
Problem instance, 1021
Problem size, 1021
Product, 907
Projection, 815
Pseudocode, 17
Pseudo-random number, 1048
Public-key cryptography, 1048

Quadrant, 778
Quadratic equation, 888
Quadratic programming, 624
Quadtree, 288–296
Queue, 211, 213, 215, 216, 220, 221, 224, 225
Quick sort, 146–157, 159–165, 167–169
Quick sort, randomized, 162–165
Quick sort, 171–177, 179–185, 189, 191, 193, 196, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 221, 223, 225, 227, 229, 231, 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, 253, 255, 257, 259, 261, 263, 265, 267, 269, 271, 273, 275, 277,

Standard template library, 21–24
  algorithms, 635–647
  containers, 611–634
  function objects, 648–658
  generic programming, 597–610
  header files, 598–599
  iterators, 635–636
  maps, 639–641
  multimaps, 641–642
  multisets, 639–641
  overview, 595–596
  queues, 637–638
  sorting, 645–646
Standardization, 5–6
Statechart diagrams, 503
Statements
  assignment, 60–61
  break, 212
  compound, 60–61
  declaration, 60–61
  do-while, 208–209
  empty, 60–61
  expression, 60–61
  for, 209–211
  if, 202–204
  if-else, 202–204
  return, 60–61
  switch, 212–214
  while, 205–208
Static binding, 500
Static data members, 360–362
Static member functions, 362–364
Static variables, 362
Stepwise refinement, 283
Stream classes, 7
Streaming stack operations, 57
Streaming out, 446–447

Subscripting, 149–150
Subscripting operator, 149–150
Subtraction, 69–70
Superclass, 304–308
Syntax, 5
Syntax diagrams, 54–59
System testing, 284
System.out, 56

Template functions, 599–601
Template parameters, 599–601
Testing, 284–285
Text files, 434–435
Text processing, 434–435
this pointer, 365–366
Throw exceptions, 1019–1020
Throw specifications, 1021–1022
Tokens, 54–55
Top-down design, 283
Top-down testing, 284
Trailing return type, 1030–1031
Transfer of control, 202–214
Try blocks, 1018–1019
Type conversion, 71–72
Type parameters, 599–601

Unary operators, 69–70
Unified Modeling Language, 502–503
Unit testing, 284
Universal character names, 1032–1033
Unqualified lookup, 1035–1036
Use cases, 502
User-defined literals, 1034–1035
Using declarations, 1037–1038

Value parameters, 95–96
Variables, 58–61
Variadic templates, 1039–1040
Vectors, 150
Virtual base classes, 1041–1042

Widening conversions, 71–72
Wrapper classes for primitive types, 1043–1044
Wrapping, 69–70

Zero-initialization, 1045–1046
Zero-length strings, 1047–1048

Computer Programming/C++

An Accessible Approach to Data Structures

In *Data Structures and Algorithms in C++, Second Edition*, Adam Drozdek has successfully created a textbook that makes the subject accessible to students who are new to data structures, while providing an appealing scope of material for more advanced students.

This text:

- Emphasizes the connection between data structures and algorithms
- Carefully presents the difficult subject of recursion
- Offers a case study in every chapter (except Chapter 2)
- Introduces the Standard Template Library and integrates it in the case studies
- Provides programming assignments in all but the first chapter

Drozdek amply illustrates his discussion of data structures with figures and case studies, which strengthens student understanding of the power and utility of data structures.

www.brookscole.com

JOIN US ON THE INTERNET
www.thomsonlearning.com

VISIT BROOKS/COLE ON THE INTERNET

ISBN 0-534-37597-9