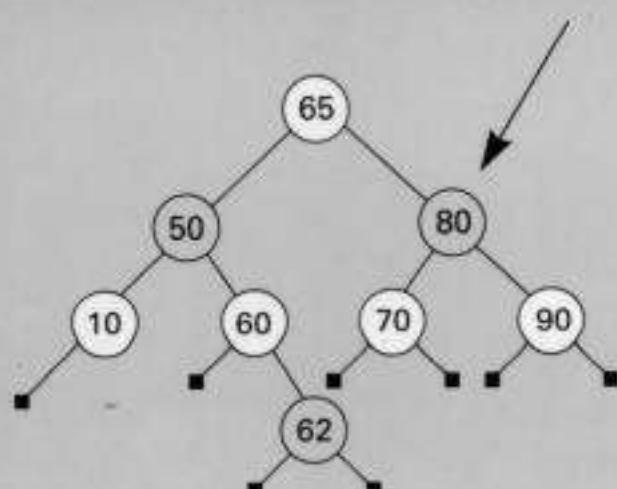
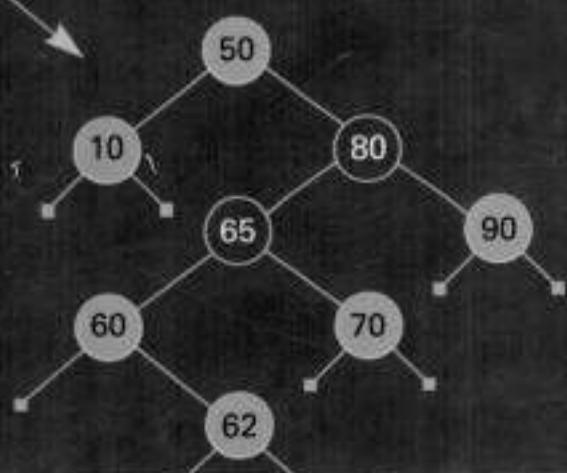
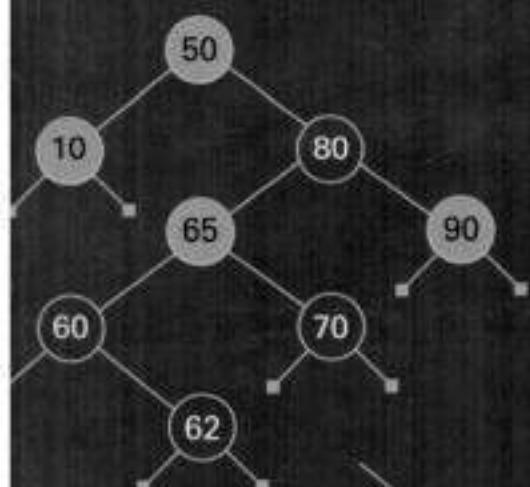


Data Structures, Algorithms and Applications IN

C++



Second Edition

SARTAJ SAHNI

Data Structures, Algorithms and Applications in

C++

Second Edition

SARTAJ SAHNI



Universities Press

Universities Press (India) Private Limited

Registered Office

3-5-819 Hyderguda, Hyderabad 500 029 (A.P.), India

Email: hyd2_upilco@sancharnet.in

Distributed by

Orient Longman Private Limited

Registered Office

3-6-752 Himayatnagar, Hyderabad 500 029 (A.P.), India

Other Offices

Bangalore, Bhopal, Bhubaneshwar, Chennai,
Ernakulam, Guwahati, Hyderabad, Jaipur, Kolkata,
Lucknow, Mumbai, New Delhi, Patna

© Silicon Press 2005

All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval systems, without permission in writing from the publisher.

First published in India by

Universities Press (India) Private Limited 2005

ISBN 81 7371 522 X

For sale in India, Pakistan, Nepal, Myanmar, Maldives, Bhutan, Bangladesh and Sri Lanka only.

Not for export to other countries.

Printed at

Orion Printers

Hyderabad 500 004

Published by

Universities Press (India) Private Limited

3-5-819 Hyderguda, Hyderabad 500 029 (A.P.), India

To my mother,
Santosh

My wife,
Neeta

and my children,
Agam, Neha, and Param

This One



5ZDB-13G-H7QA Copyrighted material

PREFACE

The study of data structures and algorithms is fundamental to computer science and engineering. A mastery of these areas is essential for us to develop computer programs that utilize computer resources in an effective manner. Consequently, all computer science and engineering curriculums include one or more courses devoted to these subjects. Typically, the first programming course introduces students to basic data structures (such as stacks and queues) and basic algorithms (such as those for sorting and matrix algebra). The second programming course covers more data structures and algorithms. The next one or two courses are usually dedicated to the study of data structures and algorithms.

The explosion of courses in the undergraduate computer science and engineering curriculums has forced many universities and colleges to consolidate material into fewer courses. At the University of Florida, for example, we offer a single one-semester undergraduate data structures and algorithms course. Students coming into this course have had a one-semester course in Java programming and another in discrete mathematics/structures.

Data Structures, Algorithms, and Applications in C++ has been developed for use in programs that cover this material in a unified course as well as in programs that spread out the study of data structures and algorithms over two or more courses. The book is divided into three parts. Part I, which consists of Chapters 1 through 4, is intended as a review of C++ programming concepts and of methods to analyze and measure the performance of programs. Students who are familiar with programming in C should be able to read Chapter 1 and bridge the gap between C and C++. Although Chapter 1 is not a primer on C++, it covers most of the C++ constructs with which students might have become rusty. These concepts include parameter passing, template functions, dynamic memory allocation, recursion, classes, inheritance, and throwing and catching exceptions. Chapters 2 and 3 are a review of methods to analyze the performance of a program—operation counts, step counts, and asymptotic notation (big oh, omega, theta, and little oh). Chapter 4 reviews methods to measure performance experimentally. This chapter also has a brief discussion of how cache affects measured run times. The applications considered in Chapter 2 explore fundamental problems typically studied in a beginning programming course—simple sort methods such as bubble, selection,

insertion, and rank (or count) sort; sequential search; polynomial evaluation using Horner's rule; and matrix operations such as matrix addition, transpose, and multiply. Chapter 3 examines binary search. Even though the primary purpose of Chapters 2 through 4 is to study performance analysis and measurement methods, these chapters also ensure that all students are familiar with a set of fundamental algorithms.

Chapters 5 through 16 form the second part of the book. These chapters provide an in-depth study of data structures. Chapters 5 and 6 form the backbone of this study by examining the array and pointer (or linked) methods of representing data. These two chapters develop C++ classes to represent the linear list data structure, using each representation method. We compare the different representation schemes with respect to their effectiveness in representing linear lists by presenting experimental data. The remaining chapters on data structures use the representation methods of Chapters 5 and 6 to arrive at representations for other data structures such as arrays and matrices (Chapter 7), stacks (Chapter 8), queues (Chapter 9), dictionaries (Chapters 10, 14, and 15), binary trees (Chapter 11), priority queues (Chapter 12), tournament trees (Chapter 15), and graphs (Chapter 16).

In our treatment of data structures, we have attempted to maintain compatibility with similar or identical structures that are available in the C++ Standard Templates Library (STL). For example, the linear list data structure that is the subject of Chapter 5 is modeled after the STL class `vector`. Throughout the book we make use of STL functions such as `copy`, `min`, and `max` so students becomes familiar with these functions.

The third part of this book, which comprises Chapters 17 through 21 (Chapters 20 and 21 are available from the Web site for this book), is a study of common algorithm-design methods. The methods we study are greedy (Chapter 17), divide and conquer (Chapter 18), dynamic programming (Chapter 19), backtracking (Chapter 20), and branch and bound (Chapter 21). Two lower-bound proofs (one for the minmax problem and the other for sorting) are provided in Section 18.4; approximation algorithms for machine scheduling (Section 12.6.2), bin packing (Section 13.5), and the 0/1 knapsack problem (Section 17.3.2) are also covered. NP-hard problems are introduced, informally, in Section 12.6.2.

A unique feature of this book is the emphasis on applications. Several real-world applications illustrate the use of each data structure and algorithm-design method developed in this book. Typically, the last section of each chapter is dedicated to applications of the data structure or design method studied earlier in the chapter. In many cases additional applications are also introduced early in the chapter. We have drawn applications from various areas—sorting (bubble, selection, insertion, rank, heap, merge, quick, bin, radix, and topological sort); matrix algebra (matrix addition, transpose, and multiplication); electronic design automation (finding the nets in a circuit, wire routing, component stack folding, switch-box routing, placement of signal boosters, crossing distribution, and backplane board ordering); compression and coding (LZW compression and Huffman coding); computational

geometry (convex hull and closest pair of points); simulation (machine shop simulation); image processing (component labeling); recreational mathematics (Towers of Hanoi, tiling a defective chessboard, and rat in a maze); scheduling (LPT schedules); optimization (bin packing, container loading, 0/1 knapsack, and matrix multiplication chains); statistics (histogramming, finding the minimum and maximum, and finding the k th smallest); and graph algorithms (spanning trees, components, shortest paths, max clique, bipartite graph covers, and traveling salesperson). Our treatment of these applications does not require prior knowledge of the application areas. The material covered in this book is self-contained and gives students a flavor for what these application areas entail.

By closely tying the applications to the more basic treatment of data structures and algorithm-design methods, we hope to give the student a greater appreciation of the subject. Further enrichment can be obtained by working through the more than 800 exercises in the book and from the associated Web site.

WEB SITE

The URL for the Web site for this book is

<http://www.cise.ufl.edu/~sahni/dsaac>

From this Web site you can obtain all the programs in the book together with sample data and generated output. The sample data are not intended to serve as a good test set for a given program; rather they are just something you can use to run the program and compare the output produced with the given output. Solutions to many of the exercises that appear in each chapter, codes for these solutions, sample tests and solutions to these tests, additional applications, and enhanced discussions of some of the material covered in the text also appear in the Web site.

HOW TO USE THIS BOOK

There are several ways in which this book may be used to teach the subject of data structures and/or algorithms. Instructors should make a decision based on the background of their students, the amount of emphasis instructors want to put on applications, and the number of semesters or quarters devoted to the subject. We give a few of the possible course outlines below. We recommend that the assignments require students to write and debug several programs, beginning with a collection of short programs and working up to larger programs as the course progresses. Students should read the text at a pace commensurate with classroom coverage of topics.

TWO-QUARTER SCHEDULE—QUARTER 1

One week of review. Data structures and algorithms sequence.

Week	Topic	Reading
1	Review of C++ and program performance.	Chapters 1–4. Assignment 1 given out.
2	Array-based representation.	Chapter 5. Assignment 1 due.
3	Linked representation.	Sections 6.1–6.4. Assignment 2 given out.
4	Bin sort and equivalence classes.	Sections 6.5.1 and 6.5.4. Assignment 2 due.
5	Arrays and matrices.	Chapter 7. Examination.
6	Stacks and queues.	Chapters 8 and 9. Assignment 3 given out.
7	Skip lists and hashing.	Chapter 10. Assignment 3 due.
8	Binary and other trees.	Sections 11.1–11.8. Assignment 4 given out.
9	Union-find application. Heaps and heap sort.	Sections 11.9.2, 12.1–12.4, and 12.6.1. Assignment 4 due.
10	Leftist trees, Huffman codes, and tournament trees.	Sections 12.5 and 12.6.3 and Chapter 13.

TWO-QUARTER SCHEDULE—QUARTER 2
Data structures and algorithms sequence.

Week	Topic	Reading
1	Binary search trees. Either AVL or red-black trees. Histogramming.	Chapters 14 and 15. Assignment 1 given out.
2	Graphs.	Sections 16.1–16.7. Assignment 1 due.
3	Graphs.	Sections 16.8 and 16.9. Assignment 2 given out.
4	The greedy method.	Sections 17.1–17.3.5. Assignment 2 due.
5	The greedy method and the divide-and-conquer method.	Sections 17.3.6 and 18.1. Assignment 3 given out.
6	Divide-and-conquer applications.	Section 18.2. Examination.
7	Solving recurrences, lower bounds, and dynamic programming.	Sections 18.3, 18.4, and 19.1. Assignment 3 due.
8	Dynamic-programming applications.	Sections 19.2.1 and 19.2.2. Assignment 4 given out.
9	Dynamic-programming applications.	Sections 19.2.3–19.2.5. Assignment 4 due.
10	Backtracking and branch-and-bound methods.	Chapters 20 and 21.

SEMESTER SCHEDULE

Two weeks of review. Data structures course.

Week	Topic	Reading
1	Review of C++.	Chapter 1. Assignment 1 given out.
2	Review of program performance.	Chapters 2–4.
3	Array-based representation.	Chapter 5. Assignment 1 due.
4	Linked representation.	Sections 6.1–6.4. Assignment 2 given out.
5	Bin sort and equivalence classes.	Sections 6.5.1 and 6.5.4.
6	Arrays and matrices.	Chapter 7. Assignment 2 due. First examination.
7	Stacks and queues. One or two applications.	Chapters 8 and 9. Assignment 3 given out.
8	Skip lists and hashing.	Chapter 10.
9	Binary and other trees.	Sections 11.1–11.8. Assignment 3 due.
10	Union-find application.	Section 11.9.2. Assignment 4 given out. Second examination.
11	Priority queues, heap sort, and Huffman codes.	Chapter 12.
12	Tournament trees and bin packing.	Chapter 13. Assignment 4 due.
13	Binary search trees. Either AVL or red-black trees. Histogramming.	Chapters 14 and 15. Assignment 5 given out.
14	Graphs.	Sections 16.1–16.7.
15	Graphs. Shortest paths.	Sections 16.8, 16.9, 17.3.5, and 19.2.3. Assignment 5 due.
16	Minimum-cost spanning trees. Merge sort and quick sort.	Sections 17.3.6, 18.2.2, and 18.2.3.

SEMESTER SCHEDULE

One week of review. Data structures and algorithms course.

Week	Topic	Reading
1	Review of program performance.	Chapters 1–4.
2	Array-based representation.	Chapter 5. Assignment 1 given out.
3	Linked representation.	Chapter 6.
4	Arrays and matrices.	Chapter 7. Assignment 1 due.
5	Stacks and queues. One or two applications.	Chapters 8 and 9. Assignment 2 given out.
6	Skip lists and hashing.	Chapter 10. Assignment 2 due. First examination.
7	Binary and other trees.	Sections 11.1–11.8. Assignment 3 given out.
8	Union-find application. Heaps and heap sort.	Sections 11.9.2, 12.1–12.4, and 12.6.1.
9	Leftist trees, Huffman codes, and tournament trees.	Sections 12.5 and 12.6.3 and Chapter 13. Assignment 3 due.
10	Binary search trees. Either AVL or red-black trees. Histogramming.	Chapters 14 and 15. Assignment 4 given out. Second examination.
11	Graphs.	Sections 16.1–16.7.
12	Graphs and the greedy method.	Sections 16.8, 16.9, 17.1, and 17.2. Assignment 4 due.
13	Container loading, 0/1 knapsack, shortest paths, and spanning trees.	Section 17.3. Assignment 5 given out.
14	Divide-and-conquer method.	Chapter 18.
15	Dynamic programming.	Chapter 19. Assignment 5 due.
16	Backtracking and branch-and-bound methods.	Chapters 20 and 21.

ACKNOWLEDGMENTS

This book would not have been possible without the assistance, comments, and suggestions of many individuals. I am deeply indebted to the following reviewers of the first edition of this book. Their valuable comments have resulted in a better manuscript.

Jacobo Carrasquel	Carnegie Mellon University
Yu Lo Cyrus Chang	University of New Hampshire
Teofilo F. Gonzalez	University of California at Santa Barbara
Laxmikant V. Kale	University of Illinois
Donald H. Kraft	Louisiana State University
Sang W. Lee	University of Michigan
Jorge Lobo	University of Illinois at Chicago
Brian Malloy	Clemson University
Thomas Miller	University of Idaho
Richard Rasala	Northeastern University
Craig E. Wills	Worcester Polytechnic Institute
Neal E. Young	Dartmouth College

Special thanks go to the students in my data structures and algorithms class who provided valuable feedback and helped debug the manuscript. Additionally, I am grateful to the following individuals at the University of Florida for their contributions: Justin Bullard, Edward Y. C. Cheng, Rajesh Dasari, Thomas Davies, Pinkesh Desai, Vinayak Goel, Haejae Jung, Kun-Suk Kim, Haibin Lu, Jawalant Patel, Sanguthevar Rajasekeran, Gauri Sukhatankar, Gayatri Venkataraman, and Joe Wilson.

Sartaj Sahni
Gainesville
June 2004

CONTENTS IN BRIEF

PART I PRELIMINARIES	1
Chapter 1 C++ Review	1
Chapter 2 Performance Analysis.....	55
Chapter 3 Asymptotic Notation	95
Chapter 4 Performance Measurement.....	121
PART II DATA STRUCTURES	137
Chapter 5 Linear Lists—Array Representation	137
Chapter 6 Linear Lists—Linked Representation.....	170
Chapter 7 Arrays and Matrices.....	222
Chapter 8 Stacks	269
Chapter 9 Queues.....	317
Chapter 10 Skip Lists and Hashing	362
Chapter 11 Binary and Other Trees	418
Chapter 12 Priority Queues	464
Chapter 13 Tournament Trees	505
Chapter 14 Binary Search Trees	529
Chapter 15 Balanced Search Trees	563
Chapter 16 Graphs.....	615
PART III ALGORITHM-DESIGN METHODS	660
Chapter 17 The Greedy Method	660
Chapter 18 Divide and Conquer	704
Chapter 19 Dynamic Programming	757
Chapter 20 Backtracking (On the Web)	793
Chapter 21 Branch and Bound (On the Web).....	829
INDEX	I-1

CONTENTS

PART I PRELIMINARIES

1 C++ Review	1
1.1 Introduction	3
1.2 Functions and Parameters	3
1.2.1 Value Parameters	3
1.2.2 Template Functions	4
1.2.3 Reference Parameters	5
1.2.4 Const Reference Parameters	6
1.2.5 Return Values	6
1.2.6 Overloaded Functions	7
1.3 Exceptions	9
1.3.1 Throwing an Exception	9
1.3.2 Handling Exceptions	10
1.4 Dynamic Memory Allocation	12
1.4.1 The Operator <code>new</code>	12
1.4.2 One-Dimensional Arrays	13
1.4.3 Exception Handling	13
1.4.4 The Operator <code>delete</code>	13
1.4.5 Two-Dimensional Arrays	14
1.5 Your Very Own Data Type	17
1.5.1 The Class <code>currency</code>	17
1.5.2 Using a Different Representation	24
1.5.3 Operator Overloading	26
1.5.4 Friends and Protected Class Members	28
1.5.5 Addition of <code>#ifndef</code> , <code>#define</code> , and <code>#endif</code> Statements	30
1.6 The Exception Class <code>illegalParameterValue</code>	33
1.7 Recursion	33
1.7.1 Recursive Mathematical Functions	35
1.7.2 Induction	36
1.7.3 Recursive C++ Functions	37
1.8 The Standard Templates Library	43

1.9 Testing and Debugging	47
1.9.1 What Is Testing?	47
1.9.2 Designing Test Data	50
1.9.3 Debugging	53
1.10 References and Selected Readings	54
2 Performance Analysis	55
2.1 What Is Performance?	57
2.2 Space Complexity	58
2.2.1 Components of Space Complexity	58
2.2.2 Examples	63
2.3 Time Complexity	66
2.3.1 Components of Time Complexity	66
2.3.2 Operation Counts	67
2.3.3 Best, Worst, and Average Operation Counts	73
2.3.4 Step Counts	80
3 Asymptotic Notation	95
3.1 Introduction	96
3.2 Asymptotic Notation	99
3.2.1 Big Oh Notation (O)	99
3.2.2 Omega (Ω) and Theta (Θ) Notations	102
3.3 Asymptotic Mathematics (Optional)	104
3.3.1 Big Oh Notation (O)	104
3.3.2 Omega Notation (Ω)	107
3.3.3 Theta Notation (Θ)	109
3.3.4 Little Oh Notation (o)	111
3.3.5 Properties	111
3.4 Complexity Analysis Examples	114
3.5 Practical Complexities	117
3.6 References and Selected Readings	120
4 Performance Measurement	121
4.1 Introduction	122
4.2 Choosing Instance Size	123
4.3 Developing the Test Data	123
4.4 Setting Up the Experiment	124
4.5 Your Cache and You	131
4.5.1 A Simple Computer Model	131
4.5.2 Effect of Cache Misses on Run Time	132
4.5.3 Matrix Multiplication	133
4.6 References and Selected Readings	136

PART II DATA STRUCTURES

5 Linear Lists—Array Representation	137
5.1 Data Objects and Structures	139
5.2 The Linear List Data Structure	140
5.2.1 The Abstract Data Type <i>linearList</i>	141
5.2.2 The Abstract Class LinearList	142
5.3 Array Representation	143
5.3.1 The Representation	143
5.3.2 Changing the Length of a One-Dimensional Array	145
5.3.3 The Class arrayList	146
5.3.4 Iterators in C++	153
5.3.5 An Iterator for arrayList	155
5.4 Vector Representation	161
5.5 Multiple Lists in a Single Array	163
5.6 Performance Measurement	166
5.7 References and Selected Readings	169
6 Linear Lists—Linked Representation	170
6.1 Singly Linked Lists and Chains	172
6.1.1 The Representation	172
6.1.2 The Struct chainNode	174
6.1.3 The Class chain	174
6.1.4 Extensions to the ADT <i>linearList</i>	181
6.1.5 The Class extendedChain	182
6.1.6 Performance Measurement	183
6.2 Circular Lists and Header Nodes	192
6.3 Doubly Linked Lists	195
6.4 Glossary of Linked List Terms	197
6.5 Applications	197
6.5.1 Bin Sort	197
6.5.2 Radix Sort	202
6.5.3 Convex Hull	205
6.5.4 Union-Find Problem	209
7 Arrays and Matrices	222
7.1 Arrays	223
7.1.1 The Abstract Data Type	223
7.1.2 Indexing a C++ Array	224
7.1.3 Row- and Column-Major Mappings	224
7.1.4 Array of Arrays Representation	227
7.1.5 Row-Major and Column-Major Representation	227
7.1.6 Irregular Two-Dimensional Arrays	228
7.2 Matrices	230

7.2.1	Definitions and Operations	230
7.2.2	The Class matrix	233
7.3	Special Matrices	239
7.3.1	Definitions and Applications	239
7.3.2	Diagonal Matrices	242
7.3.3	Tridiagonal Matrix	244
7.3.4	Triangular Matrices	245
7.3.5	Symmetric Matrices	246
7.4	Sparse Matrices	252
7.4.1	Motivation	252
7.4.2	Representation Using a Single Linear List	252
7.4.3	Representation Using Many Linear Lists	258
7.4.4	Performance Measurement	262
8	Stacks	269
8.1	Definition and Applications	271
8.2	The Abstract Data Type	274
8.3	Array Representation	274
8.3.1	Implementation as a Derived Class	274
8.3.2	The Class arrayStack	277
8.3.3	Performance Measurement	277
8.4	Linked Representation	282
8.4.1	The Class derivedLinkedStack	282
8.4.2	The Class linkedStack	282
8.4.3	Performance Measurement	282
8.5	Applications	284
8.5.1	Parenthesis Matching	284
8.5.2	Towers of Hanoi	285
8.5.3	Rearranging Railroad Cars	289
8.5.4	Switch Box Routing	294
8.5.5	Offline Equivalence Class Problem	297
8.5.6	Rat in a Maze	300
8.6	References and Selected Readings	316
9	Queues	317
9.1	Definition and Applications	319
9.2	The Abstract Data Type	320
9.3	Array Representation	322
9.3.1	The Representation	322
9.3.2	The Class arrayQueue	325
9.4	Linked Representation	330
9.5	Applications	333
9.5.1	Railroad Car Rearrangement	333
9.5.2	Wire Routing	336

9.5.3	Image-Component Labeling	9.5.1 Definitions and Objects	341
9.5.4	Machine Shop Simulation	9.5.2 The Class Matrix	344
9.6	References and Selected Readings	9.6.1 Backtracking Methods	361
		9.6.2 Definitions and Applications	361
10	Skip Lists and Hashing	10.1 Dictionaries	362
10.1	Dictionaries	10.1.1 Definitions and Applications	364
10.2	The Abstract Data Type	10.2.1 The ADT	366
10.3	Linear List Representation	10.3.1 Linear List Representation	367
10.4	Skip List Representation (Optional)	10.4.1 The Ideal Case	368
10.4.1	The Ideal Case	10.4.1.1 Definitions and Applications	368
10.4.2	Insertions and Deletions	10.4.2.1 Insertion	372
10.4.3	Assigning Levels	10.4.3.1 Representation of Linear Lists	373
10.4.4	The Struct <code>skipNode</code>	10.4.4.1 Definition	374
10.4.5	The Class <code>skipList</code>	10.4.5.1 Definition	374
10.4.6	Complexity of <code>skipList</code> Methods	10.4.6.1 Complexity	378
10.5	Hash Table Representation	10.5.1 Ideal Hashing	381
10.5.1	Ideal Hashing	10.5.1.1 Definitions and Applications	381
10.5.2	Hash Functions and Tables	10.5.2.1 Hash Function Representation	382
10.5.3	Linear Probing	10.5.3.1 Implementation as a Dictionary	387
10.5.4	Hashing with Chains	10.5.4.1 Definition	394
10.6	An Application—Text Compression	10.6.1 LZW Compression	402
10.6.1	LZW Compression	10.6.1.1 LZW Representation	403
10.6.2	Implementation of LZW Compression	10.6.2.1 The Class <code>LZW</code>	403
10.6.3	LZW Decompression	10.6.3.1 The Class <code>lzwDecoder</code>	409
10.6.4	Implementation of LZW Decompression	10.6.4.1 Application	411
10.6.5	Performance Evaluation	10.6.5.1 Performance	413
10.7	References and Selected Readings	10.7.1 Text Compression Methods	416
11	Binary and Other Trees	11.1 Trees	418
11.1	Trees	11.1.1 Definitions and Applications	418
11.2	Binary Trees	11.2.1 Definitions and Applications	420
11.3	Properties of Binary Trees	11.3.1 Properties of Binary Trees	425
11.4	Representation of Binary Trees	11.4.1 Array-Based Representation	426
11.4.1	Array-Based Representation	11.4.1.1 Definitions and Applications	429
11.4.2	Linked Representation	11.4.2.1 Definitions and Applications	430
11.5	Common Binary Tree Operations	11.5.1 Definitions and Applications	432
11.6	Binary Tree Traversal	11.6.1 Definitions and Applications	432
11.7	The ADT <i>BinaryTree</i>	11.7.1 Definitions and Applications	439
11.8	The Class <code>linkedBinaryTree</code>	11.8.1 Definitions and Applications	439
11.9	Applications	11.9.1 Placement of Signal Boosters	443
11.9.1	Placement of Signal Boosters	11.9.1.1 Definitions and Applications	443
11.9.2	Union-Find Problem	11.9.2.1 Definitions and Applications	450
11.10	References and Selected Readings	11.10.1 Definitions and Applications	463

12 Priority Queues	464
12.1 Definition and Applications	466
12.2 The Abstract Data Type	467
12.3 Linear Lists	468
12.4 Heaps	469
12.4.1 Definitions	469
12.4.2 Insertion into a Max Heap	470
12.4.3 Deletion from a Max Heap	471
12.4.4 Max Heap Initialization	472
12.4.5 The Class maxHeap	474
12.4.6 Heaps and the STL	478
12.5 Leftist Trees	479
12.5.1 Height- and Weight-Biased Min and Max Leftist Trees	479
12.5.2 Insertion into a Max HBLT	482
12.5.3 Deletion from a Max HBLT	482
12.5.4 Melding Two Max HBLTs	482
12.5.5 Initialization	485
12.5.6 The Class maxHblt	485
12.6 Applications	491
12.6.1 Heap Sort	491
12.6.2 Machine Scheduling	493
12.6.3 Huffman Codes	497
12.7 References and Selected Readings	504
13 Tournament Trees	505
13.1 Winner Trees and Applications	506
13.2 The ADT WinnerTree	511
13.3 Winner Tree Implementation	511
13.3.1 Representation	511
13.3.2 Initializing a Winner Tree	513
13.3.3 Replaying Matches	514
13.3.4 The Class completeWinnerTree	514
13.4 Loser Trees	515
13.5 Applications	518
13.5.1 Bin Packing Using First Fit	518
13.5.2 Bin Packing Using Next Fit	524
13.6 References and Selected Readings	528
14 Binary Search Trees	529
14.1 Definitions	531
14.1.1 Binary Search Trees	531
14.1.2 Indexed Binary Search Trees	532
14.2 Abstract Data Types	533
14.3 Binary Search Tree Operations and Implementation	535

14.3.1 The Class <code>binarySearchTree</code>	535
14.3.2 Searching	536
14.3.3 Inserting an Element	536
14.3.4 Deleting an Element	536
14.3.5 Height of a Binary Search Tree	540
14.4 Binary Search Trees with Duplicates	544
14.5 Indexed Binary Search Trees	545
14.6 Applications	548
14.6.1 Histogramming	548
14.6.2 Best-Fit Bin Packing	550
14.6.3 Crossing Distribution	553
15 Balanced Search Trees	563
15.1 AVL Trees	566
15.1.1 Definition	566
15.1.2 Height of an AVL Tree	567
15.1.3 Representation of an AVL Tree	567
15.1.4 Searching an AVL Search Tree	567
15.1.5 Inserting into an AVL Search Tree	567
15.1.6 Deletion from an AVL Search Tree	572
15.2 Red-Black Trees	577
15.2.1 Definition	577
15.2.2 Representation of a Red-Black Tree	579
15.2.3 Searching a Red-Black Tree	579
15.2.4 Inserting into a Red-Black Tree	579
15.2.5 Deletion from a Red-Black Tree	584
15.2.6 Implementation Considerations and Complexity	588
15.3 Splay Trees	592
15.3.1 Introduction	592
15.3.2 The Splay Operation	593
15.3.3 Amortized Complexity	595
15.4 B-Trees	598
15.4.1 Indexed Sequential Access Method (ISAM)	598
15.4.2 m -Way Search Trees	599
15.4.3 B-Trees of Order m	602
15.4.4 Height of a B-Tree	603
15.4.5 Searching a B-Tree	604
15.4.6 Inserting into a B-Tree	604
15.4.7 Deletion from a B-Tree	607
15.4.8 Node Structure	612
15.5 References and Selected Readings	614

CHAPTER 1

C++ REVIEW

BIRD'S-EYE VIEW

Well, folks, we are about to begin a journey through the world of data structures, algorithms, and computer programs that solve many real-life problems. The program development process will require us to (1) represent data in an effective way and (2) develop a suitable step-by-step procedure (or algorithm) that can be implemented as a computer program. Effective data representation requires expertise in the field of data structures, and the development of a suitable step-by-step procedure requires expertise in the field of algorithm design methods.

Before you embark on the study of data structures and algorithm design methods, you need to be a proficient C++ programmer and an adept analyst of computer programs. These essential skills are typically gained from introductory C++ and discrete structures courses. The first four chapters of this book are intended as a review of these skills, and much of the material covered in these chapters should already be familiar to you.

In this first chapter we discuss some features of the C++ language. However, this chapter is not intended as a C++ primer, and we do not cover basic constructs such as assignment statements, if statements, and looping statements (e.g., for and while). This chapter covers the following C++ language features:

- The different modes of parameter passing in C++ (by value, by reference, and by const reference).

- The different modes in which a function or method may return a value (by value, by reference, and by const reference).
- Template functions.
- Recursive functions.
- Constant functions.
- The C++ memory allocation and deallocation functions `new` and `delete`.
- The C++ exception handling constructs `try`, `catch`, and `throw`.
- Classes and template classes.
- Public, protected, and private class members.
- Friends of a class.
- Operator overloading.
- The standard templates library.

CHAPTER I**C++ REVIEW**

Additional C++ features that may not have been covered in a first C++ course are introduced in later chapters as needed. Chapter 1 also includes codes for the following applications:

BIRD'S-EYE VIEW

- Dynamic allocation and deallocation of one- and two-dimensional arrays.
- Finding the roots of a quadratic function.
- Generating all permutations of n items.
- Finding the maximum of n elements.

Chapter 1 concludes with tips on how to test and debug a program.

• The different modes of branching based on C++ (if, while, for, etc.) and by control (else).

1.1 INTRODUCTION

Some of the questions we should ask when examining a computer program are

- Is it correct?
- How easy is it to read the program and understand the code?
- Is the program well documented?
- How easy is it to make changes to the program?
- How much memory is needed to run the program?
- For how long will the program run?
- How general is the code? Will it solve problems over a large range of inputs without modification?
- Can the code be compiled and run on a variety of computers or are modifications needed to run it on different computers?

The relative importance of some of these questions depends on the application environment. For example, if we are writing a program that is to be run once and discarded, then correctness, memory and time requirements, and the ability to compile and run the code on a single computer are the dominating considerations. Regardless of the application, the most important attribute of a program is correctness. An incorrect program, no matter how fast, how general, or how well documented, is of little use (until it is corrected). Although we do not explicitly dwell on techniques to establish program correctness, we provide informal proofs of correctness and implicitly develop programming habits conducive to the production of correct codes. The goal is to teach techniques that will enable you to develop correct, elegant, and efficient solutions.

Before we can begin the study of these techniques, we must review some essential aspects of the C++ language, techniques to test and debug programs, and techniques to analyze and measure the performance of a program. This chapter focuses on the first two items. Chapters 2 through 4 review performance analysis and measurement techniques.

1.2 FUNCTIONS AND PARAMETERS

1.2.1 Value Parameters

Consider the C++ function `abc` (Program 1.1). This function computes the expression $a + b * c$ for the case when a , b , and c are integers. The result is also an integer.

In Program 1.1 a , b , and c are the **formal parameters** of the function `abc`. Each is of type integer. If the function is invoked by the statement

```
int abc(int a, int b, int c)
{
    return a + b * c;
}
```

Program 1.1 Compute an integer expression

```
z = abc(2,x,y)
```

then 2, x, and y are the **actual parameters** that correspond to a, b, and c, respectively.

In Program 1.1 the formal parameters a, b, and c are actually **value** formal parameters. At run time the value of the actual parameter that corresponds to a value formal parameter is copied into the formal parameter before the function is executed. This copying is done using the **copy constructor** for the data type of the formal parameter. If the actual and value formal parameters are of different data types, a type conversion is performed from the type of the actual parameter to that of the value formal parameter provided such a type conversion is defined.

When the invocation abc(2,x,y) is executed, a is assigned the value 2, b is assigned the value of x, and c is assigned the value of y. In case x and/or y are not of type **int**, then a type conversion between their type and **int** is performed prior to the assignment of values to b and c (provided such a type conversion is defined). For example, if x is of type **double** and has the value 3.8, then b is assigned the value 3.

When a function terminates, **destructors** for the data types of the formal parameters destroy the value formal parameters. *When a function terminates, formal parameter values are not copied back into the actual parameters.* Consequently, function invocation does not change the actual parameters that correspond to value formal parameters.

1.2.2 Template Functions

Suppose we wish to write another function to compute the same expression as computed by Program 1.1. However, this time a, b, and c are of type **float**, and the result is also of this type. Program 1.2 gives the code. Programs 1.1 and 1.2 differ only in the data type of the formal parameters and of the value returned.

Rather than write a new version of the code for every possible data type of the formal parameters, we can write a generic code in which the data type is a variable whose value is to be determined by the compiler. This generic code is written using the **template** statement as shown in Program 1.3.

From this generic code the compiler can construct Program 1.1 by substituting **int** for T and Program 1.2 by substituting **float** for T. In fact, the compiler can

```
float abc(float a, float b, float c)
{
    return a + b * c;
}
```

Program 1.2 Compute a floating-point expression

```
template<class T>
T abc(T a, T b, T c)
{
    return a + b * c;
}
```

Program 1.3 Compute an expression using a template function

construct a double-precision version or a long-integer version (or both) of the code by substituting `double` or `long` for `T`. Writing `abc` as a template function eliminates the need to know the data type of the formal parameters when we write the code.

1.2.3 Reference Parameters

The use of value parameters in Program 1.3 increases the run-time cost. For example, consider the operations involved when a function is invoked and when it terminates. When `a`, `b`, and `c` are value parameters, the copy constructor for type `T` copies the values of the corresponding actual parameters into the formal parameters `a`, `b`, and `c` upon entry into the function. At the time of exiting the function, the destructor for type `T` is invoked, and the formal parameters `a`, `b`, and `c` are destroyed.

Suppose that `T` is the user-defined data type `matrix` whose copy constructor copies all entries of the matrix and whose destructor destroys the matrix entries one by one (assume that the operators `+`, `*`, and `/` have been defined for the data type `matrix`). If `abc` is invoked with each actual parameter being a matrix with 1000 elements, then copying the three actual parameters into `a`, `b`, and `c` would require 3000 operations. When `abc` terminates, the `matrix` destructor is invoked to destroy `a`, `b`, and `c` at a cost of an additional 3000 operations.

In the code of Program 1.4, `a`, `b`, and `c` are **reference parameters**. If `abc` is invoked by the statement `abc(x,y,z)` where `x`, `y`, and `z` are of the same data type, then these actual parameters are bound to the names `a`, `b`, and `c`, respectively. Therefore, during execution of the function `abc`, the names `x`, `y`, and `z`, respectively, are used in place of the names `a`, `b`, and `c`. Unlike the case of value parameters, this

program does not copy actual parameter values at the time of invocation and does not invoke the type T destructor upon exit.

```
template<class T>
T abc(T& a, T& b, T& c)
{
    return a + b * c;
}
```

Program 1.4 Compute an expression using reference parameters

Consider the case when the actual parameters that correspond to **a**, **b**, and **c** are matrices **x**, **y**, and **z** with 1000 elements each. Since the values of **x**, **y**, and **z** are now not copied into the formal parameters, we save the 3000 operations needed to do the copying when value parameters are used.

1.2.4 Const Reference Parameters

C++ provides yet another mode of parameter passing, **const reference**. This mode designates reference parameters that are not changed by the function. For example, in Program 1.4 the values of **a**, **b**, and **c** do not change, so we may rewrite the code as shown in Program 1.5.

Program 1.5 Compute an expression using const reference parameters

Using the **const** qualifier to designate reference parameters that the function does not change has an important software-engineering value. The function header tells the user that the function will not change the actual parameters.

Using the syntax of Program 1.6, we can obtain a more general version of Program 1.5. In the new version each formal parameter may be of a different type, and the result is of the same type as the first parameter (for example).

1.2.5 Return Values

A function may make a value return, a reference return, or a const reference return. The preceding examples make value returns. In such a return, the object that is

~~overloaded operators and their meanings must be understood first or before reading this section.~~

```
template<class T1, class T2, class T3>
T1 abc(const T2& a, const T2& b, const T3& c)
    return a + b * c;
```

~~This function is used to calculate the sum of two numbers and the product of the second and third numbers. The result is returned as a reference to a T1 object. The parameters are references to T2 objects, and the return value is a reference to a T3 object. The code is part of Program 1.1.~~

Program 1.6 A more general version of Program 1.5**EXERCISES**

being returned is copied into the invoking (or return) environment. This copying is necessary in all versions of the function `abc`, as the result of the expression computed by this function is saved in a local temporary variable. When the function terminates, the space allocated to this temporary variable (as well as to all other temporary variables, local variables and value formal parameters) is freed and its value is no longer available. To avoid losing this value, we copy it from the temporary variable into the return environment before releasing the space allocated to temporary variables, local variables and value formal parameters.

We specify a reference return by adding the symbol `&` as a suffix to the return type. The function header

```
T& mystery(int i, T& z)
```

```
; x = &i
; y = &z
{
```

defines a function `mystery` that makes a reference return of type `T`. It could, for example, return `z` using the following statement:

```
return z;
```

To understand how this works, consider the following statement: `x = &z`.

Such a return would not involve copying the value of `z` into the return environment. When function `mystery` terminates, the space allocated to the value formal parameter `i` and all local variables is released. Because `z` is simply a reference to an actual parameter, it is not affected.

A `const` reference return is specified by adding the keyword `const` to the function header as in

```
const T& mystery(int i, -T& z)
```

A `const` reference return is similar to a reference return except that the item returned is designated a `constant object`.

1.2.6 Overloaded Functions

The signature of a function is defined by the data types of the method's formal parameters and the number of formal parameters. The signature of the method `abc` of Program 1.1 is `(int, int, int)`. C++ allows you to define two or more

functions with the same name provided no two functions with the same name have the same signature. The ability to define several functions with the same name is called **function overloading**. Because of the availability of function overloading, we can write a program that includes both the function `abc` of Program 1.1 and the function `abc` of Program 1.2. By matching the signature used by a function invocation statement to the signature in a function definition, the C++ compiler can determine which of the overloaded functions is meant.

EXERCISES

1. Explain why the `swap` method of Program 1.7 fails to swap (i.e., interchange) the values of the integer actual parameters that correspond to the formal parameters `x` and `y`. How would you change this code so that it correctly swaps the values of the actual parameters.

```
void swap(int x, int y)
{// swap the integers x and y
    int temp = x;
    x = y;
    y = temp;
}
```

Program 1.7 Incorrect code to swap two integers

2. Write a template function `count` that returns the number of occurrences of `value` in the array `a[0:n-1]`. Test your code.
3. Write a template function `fill` that sets `a[start:end-1]` to `value`. Test your code.
4. Write a template function `inner_product` that returns $\sum_{i=0}^{n-1} a[i] * b[i]$. Test your code.
5. Write a template function `iota` that sets `a[i] = value + i, 0 \leq i < n`. Test your code.
6. Write a template function `is_sorted` that returns `true` iff `a[0:n-1]` is sorted. Test your code.
7. Write a template function `mismatch` that returns the smallest `i, 0 \leq i < n` such that `a[i] \neq b[i]`. Test your code.
8. Do the following headers define functions with different signatures? Why?

- (a) `int abc(int a, int b, int c)`
 (b) `float abc(int a, int b, int c)`
9. Suppose we have a program that contains both of the `abc` functions given in Programs 1.1 and 1.2. Which `abc` function is invoked by each of the following statements. Which will result in a compile-time error? Why?
- (a) `cout << abc(1, 2, 3);) << endl;`
 (b) `cout << abc(1.0F, 2.0F, 3.0F);) < endl;`
 (c) `cout << abc(1, 2, 3.0F);) < endl;`
 (d) `cout << abc(1.0, 2.0, 3.0);) << endl;`

1.3 EXCEPTIONS

1.3.1 Throwing an Exception

Exceptions are used to signal the occurrence of errors. For example, the evaluation of the expression `a+b*c+b/c` with `a = 2`, `b = 1`, and `c = 0` requires us to divide by zero, which is an error. Although this error is not detected by C++, your hardware will detect the error and throw an exception.

We can write C++ programs that check for exceptional conditions and throw an exception when such a condition is detected. For example, the task performed by function `abc` (Program 1.1) may be defined only when each of its three parameters is greater than 0. In this case we would modify the code of Program 1.1 to first check that the values of `a`, `b`, and `c` are actually > 0 . If one or more of these parameters is ≤ 0 , we can signal an exceptional condition by throwing an exception as is done in Program 1.8. The exception thrown by this program is of type `char*`.

```
int abc(int a, int b, int c)
{
    if (a <= 0 || b <= 0 || c <= 0)
        throw "All parameters should be > 0";
    return a + b * c;
}
```

Program 1.8 Throwing an exception of type `char*`

We get more flexibility in processing exceptions when we define an exception class for each of the different kinds of exceptions (e.g., divide by zero, illegal parameter value, illegal input value, array index out of range) that our program may throw. For example, C++ has a hierarchy of exception classes with the class `exception`

as root. Standard C++ functions signal exceptional conditions by throwing exceptions of a type that is derived from the root or base class `exception`. For example, the C++ operator `new` that does dynamic memory allocation throws an exception of type `bad_alloc` when it is unable to make the requested memory allocation; `bad_alloc` is derived from the base class `exception`. Similarly, the C++ function `typeid`, which determines the type of an object, throws an exception of type `bad_typeid` when you attempt to determine the type of the NULL object; `bad_typeid` also is derived from the base class `exception`. In Section 1.6 we shall see how to define an exception class.

1.3.2 Handling Exceptions

Exceptions that might be thrown by a piece of code can be handled by enclosing this code within a `try` block. The `try` block is then followed by zero or more `catch` blocks. Each `catch` block has a parameter or argument whose type determines the type of exception that may be caught by that `catch` block. For example, the block

```
catch (char* e) {}
```

catches exceptions of type `char*` while the block

```
catch (bad_alloc e) {}
```

catches exceptions of type `bad_alloc`. The block

```
catch (exception& e) {}
```

catches exceptions of type `exception` as well as of all types derived from `exception` (e.g., `bad_alloc` and `bad_typeid`). The block

```
catch (...) {}
```

catches all exceptions regardless of their type.

A `catch` block typically contains code to recover from the exception that has occurred, or if recovery is not possible, the code in the `catch` block prints out an error message. Program 1.9 shows an example of the `try-catch` construct. The method `abc` that is invoked within the `try` block is the one given in Program 1.8.

Although Program 1.9 has a single `catch` block following the `try` block, it is possible to follow a `try` block with several `catch` blocks. When the code within a `try` block terminates with no exception, we bypass the `catch` blocks. When an exception is thrown, normal execution of the `try` block terminates and we enter the first `catch` block that can catch an exception of the type thrown. Following the execution of the code within this matching `catch` block, we bypass the remaining `catch` blocks. If no `catch` block matches the thrown exception type, then the exception propagates through the hierarchy of nested enclosing `try` blocks to the

```
int main()
{
    try {cout << abc(2,0,4) << endl;}
    catch (char* e)
    {
        cout << "The parameters to abc were 2, 0, and 4" << endl;
        cout << "An exception has been thrown" << endl;
        cout << e << endl;
        return 1;
    }
    return 0;
}
```

Program 1.9 Catching an exception of type `char*`

first `catch` block in this hierarchy that can handle the exception. If the exception is not caught by any `catch` block, the program terminates abnormally.

When Program 1.9 executes, `abc` throws an exception of type `char*`. This exception causes `abc` to terminate without the evaluation of the expression. Also, the `try` block terminates immediately (the `cout` in the `try` block doesn't complete). Since the type of the exception thrown by `abc` is the same as that of the `catch` block's parameter `e`, the exception is caught by this `catch` block; `e` is assigned the thrown exception; and the catch block is entered. Figure 1.1 gives the output generated by Program 1.9.

The parameters to abc were 2, 0, and 4
An exception has been thrown
All parameters should be > 0

Figure 1.1 Output from Program 1.9

EXERCISES

10. Modify Program 1.8 so that it throws an exception of type `int`. The value of the thrown exception should be 1 if `a`, `b`, and `c` are all less than 0; the value should be 2 if all three equal 0. When neither of these conditions is satisfied, no exception is thrown. Write a `main` function that uses your modified code; catches the exception if thrown; and outputs a message that depends on the value of the thrown exception. Test your code.

11. Do Exercise 2. Your code for the function should throw an exception of type `char*` in case $n < 1$. Test your code.

1.4 DYNAMIC MEMORY ALLOCATION

1.4.1 The Operator `new`

Run-time or dynamic allocation of memory may be done using the C++ operator `new`. This operator returns a pointer to the allocated memory. For example, to dynamically allocate memory for an integer, we must declare a variable (e.g., `y`) to be a pointer to an integer using this statement:

```
int *y;
```

When the program needs to actually use the integer, memory may be allocated to it using this syntax:

```
y = new int;
```

The operator `new` allocates enough memory to hold an integer, and a pointer to this memory is returned and saved in `y`. The variable `y` references the pointer to the integer, and `*y` references the integer. To store an integer value, for example 10, in the newly allocated memory, we can use the following syntax:

```
*y = 10;
```

We can combine the three steps—declare `y`, allocate memory, and assign a value to `*y`—into a smaller number of steps as shown in the following examples:

```
int *y = new int;  
*y = 10;
```

or

```
int *y = new int (10);
```

or

```
int *y;  
y = new int (10);
```

1.4.2 One-Dimensional Arrays

This text includes many examples of functions that work with one- and two-dimensional arrays. The size of these arrays may not be known at compile time and may, in fact, change from one invocation of the function to the next. Consequently, memory for these arrays needs to be allocated dynamically.

To create a one-dimensional floating-point array **x** at run time, we must declare **x** as a pointer to a **float** and then allocate enough memory for the array. For example, a floating-point array of size **n** may be created as follows:

```
float *x = new float [n];
```

The operator **new** allocates memory for **n** floating-point numbers and returns a pointer to the first of these. The array elements may be addressed using the syntax **x[0]**, **x[1]**, ..., **x[n-1]**.

1.4.3 Exception Handling

What happens when the statement

```
float *x = new float [n];
```

is executed and the computer doesn't have enough memory for **n** floating-point numbers? In this case **new** cannot possibly allocate the desired amount of memory, and an exception of type **bad_alloc** is thrown. We may detect the failure of **new** by catching the exception with the **try - catch** construct:

```
float *x;
try {x = new float [n];}
catch (bad_alloc e)
{// enter only when new fails
    cerr << "Out of Memory" << endl;
    exit(1);
}
```

1.4.4 The Operator **delete**

Dynamically allocated memory should be freed when it is no longer needed. The freed memory can then be reused to create new dynamically allocated structures. We can use the C++ operator **delete** to free space allocated using the operator **new**. The statements

```
delete y;
delete [] x;
```

free the memory allocated to ***y** and the one-dimensional array **x**.

1.4.5 Two-Dimensional Arrays

Although C++ provides several mechanisms for declaring two-dimensional arrays, most of these mechanisms require that both dimensions be known at compile time. Further, when these mechanisms are used, it is difficult to write functions that allow a formal parameter which is a two-dimensional array whose second dimension is unknown. This is so because when a formal parameter is a two-dimensional array, we must specify the value of the second dimension. For example, `a[] [10]` is a valid formal parameter for a function; `a[] []` is not.

An effective way to overcome these limitations is to use dynamic memory allocation for all two-dimensional arrays. Throughout this text we use dynamically allocated two-dimensional arrays.

When both dimensions of the array are known at compile time, the array may be created using a syntax similar to that used for one-dimensional arrays. For example, a seven by five array of type `char` may be declared using the syntax:

```
char c[7][5];
```

When at least one of the dimensions is unknown at compile time, the array must be created at run time using the `new` operator. A two-dimensional character array for which the number of columns—for example, 5—is known at compile time may be allocated using the following syntax:

```
char (*c)[5];
try {c = new char [n][5];}
catch (bad_alloc)
{// enter only when new fails
    cerr << "Out of Memory" << endl;
    exit(1);
}
```

The number of rows `n` may be determined at run time either via computation or user input. When the number of columns is not known at compile time, the array cannot be allocated by a simple invocation of `new` (even if the number of rows is known). To construct the two-dimensional array, we view it as composed of several rows. Each row is a one-dimensional array and may be created using `new` as discussed earlier. Pointers to each row may be saved in another one-dimensional array. Figure 1.2 shows the structure that needs to be established for the case of a three by five array `x`.

`x[0]`, `x[1]`, and `x[2]` point to the first element of rows 0, 1, and 2, respectively. So if `x` is to be a character array, then `x[0:2]` are pointers to characters and `x` is itself a pointer to a pointer to a character. `x` may be declared using the following syntax:

```
char ***x;
```

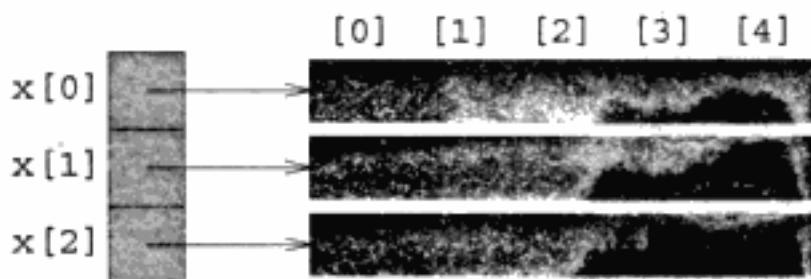


Figure 1.2 Memory structure for a three by five array

To create the memory structure of Figure 1.2, we can use the code of Program 1.10, which creates a two-dimensional array of type T. The array has `numberOfRows` rows and `numberOfCols` columns. The code first gets memory for the pointers `x[0]` through `x[numberOfRows-1]`. Next it gets memory for each row of the array. This code invokes `new` `numberOfRows + 1` times. If one of these invocations of `new` throws an exception, program control is transferred to the `catch` block and the value `false` is returned. If none of the invocations of `new` throws an exception, the array construction is successful and `make2dArray` returns the value `true`. The elements of the created array `x` may be indexed using the standard `x[i][j]` notation, $0 \leq i < \text{numberOfRows}$, $0 \leq j < \text{numberOfColumns}$.

```
template <class T>
bool make2dArray(T ** &x, int numberOfRows, int numberOfColumns)
{// Create a two dimensional array.

    try {
        // create pointers for the rows
        x = new T * [numberOfRows];

        // get memory for each row
        for (int i = 0; i < numberOfRows; i++)
            x[i] = new int [numberOfColumns];
        return true;
    }
    catch (bad_alloc) {return false;}
}
```

Program 1.10 Allocate memory for a two-dimensional array

In Program 1.10 the exception (if any) thrown by `new` is reported to the invoking

function as the Boolean value `false`. The failure of `make2dArray` may also be reported to the invoking function by simply doing nothing. If we use the code of Program 1.11, the invoking function can catch any exception thrown by `new`.

```
template <class T>
void make2dArray(T **&x, int numberOfRows, int numberOfColumns)
{// Create a two-dimensional array.

    // create pointers for the rows
    x = new T * [numberOfRows];

    // get memory for each row
    for (int i = 0; i < numberOfRows; i++)
        x[i] = new T [numberOfColumns];
}
```

Program 1.11 Make a two-dimensional array but do not catch exceptions

When `make2dArray` is defined as in Program 1.11, we can use the code

```
try {make2dArray(x,r,c);}
catch (bad_alloc)
{
    cerr << "Could not create x" << endl;
    exit(1);
}
```

to determine a shortage of memory. Not catching the exception within `make2dArray` not only simplifies the code for this function but also allows the exception to be caught at a point where the user is better able to report a meaningful error or attempt error recovery.

We can free the memory allocated to a two-dimensional array by Program 1.10 by first freeing the memory allocated in the `for` loop to each row and then freeing the memory allocated for the row pointers, as shown in Program 1.12. Notice that this code sets `x` to zero, which prevents the user from accessing the memory that was freed.

EXERCISES

12. Write a general version of `make2dArray` (Program 1.11) whose third parameter is a one-dimensional array `rowSize` rather than the integer `numberOfColumns`. Your function should create a two-dimensional array in which row `i` has `rowSize[i]` positions.

```
template <class T>
void delete2dArray(T ** &x, int numberOfRows)
{// Delete the two-dimensional array x.

    // delete the memory for each row
    for (int i = 0; i < numberOfRows; i++)
        delete [] x[i];

    // delete the row pointers
    delete [] x;
    x = NULL;
}
```

Program 1.12 Free the memory allocated by `make2dArray`

13. Write a template function `changeLength1D` to change the length (i.e., number of positions) of a one-dimensional array from `oldLength` to `newLength`. Your function should allocate space for a new one-dimensional array of length `newLength`; copy the first `min{oldLength, newLength}` elements of the old array into the new one; and free the space allocated to the old array. Test your code.
14. Write a function `changeLength2D` that changes the dimensions of a two-dimensional array (see Exercise 13). Test your code.

1.5 YOUR VERY OWN DATA TYPE

1.5.1 The Class `currency`

The C++ language supports data types such as `int`, `float`, and `char`. Many of the applications we develop in this text require additional data types that are not supported by the language. The most flexible way to define your own data types in C++ is to use the `class` construct. Suppose you wish to deal with objects (also referred to as instances) of type `currency`. Objects of type `currency` have a sign component (plus or minus), a dollar component, and a cents component. Two examples are \$2.35 (sign is plus, 2 dollars, and 35 cents) and -\$6.05 (sign is minus, 6 dollars, and 5 cents). Some of the functions or operations we wish to perform on objects of this type follow:

- Set their value.
- Determine the components (i.e., sign, dollar amount, and number of cents).

- Add two objects of type currency.
- Increment the value.
- Output.

Suppose we choose to represent objects of type currency using an unsigned long variable **dollars**, an unsigned integer **cents**, and a variable **sign** of type **signType** where the data type **signType** is defined as

```
enum signType {plus, minus};
```

We may declare a C++ class **currency** using the syntax of Program 1.13. The first line simply says that we are declaring a class whose name is **currency**. The class declaration is then enclosed in braces (**{}**). The class declaration has been divided into two sections **public** and **private**. The **public** section declares functions (or methods) that operate on objects (or instances) of type **currency**. These functions are *visible* to the users of the class and are the only means by which users can interact with objects of type **currency**. The **private** section declares functions and data members (simple variables, arrays, and so on that may hold data values) that are not visible to users of the class. By having a **public** section and a **private** section, we can let the user see only what he or she needs to see while we hide the remaining information (generally having to do with implementation details). *Although C++ syntax permits you to declare data members in the public section, good software-engineering practice discourages this procedure.*

The first function in the **public** section has the same name as the class name. Member functions that have the same name as that of the class are called **constructor** functions. Constructor functions specify how to create an object of a given type and are not permitted to return a value. In our case the constructor has three parameters whose default values are **plus**, 0, and 0. The implementation of the constructor function is provided later in this section. Constructor functions are invoked automatically when an object of type **currency** is being created. Two ways to create objects of type **currency** are

```
currency f, g(plus, 3, 45), h(minus, 10);
currency *m = new currency (plus, 8, 12);
```

The first line declares three variables (**f**, **g**, and **h**) of type **currency**. **f** is to be initialized using the default values **plus**, 0, and 0, whereas **g** is to be initialized to \$3.45 and **h** to -\$10.00. Notice that the initialization values correspond to the constructor parameters from left to right. If the number of initialization values is less than the number of constructor parameters, the remaining parameters are assigned their default values. Line 2 declares **m** as a pointer to an object of type **currency**. We invoke the **new** operator to create an object of type **currency** and store a pointer to this object in **m**. The created object is to be initialized to \$8.12.

```
class currency
{
    public:
        // constructor
        currency(signType theSign = plus,
                  unsigned long theDollars = 0,
                  unsigned int theCents = 0);
        // destructor
        ~currency() {}
        void setValue(signType, unsigned long, unsigned int);
        void setValue(double);
        signType getSign() const {return sign;}
        unsigned long getDollars() const {return dollars;}
        unsigned int getCents() const {return cents;}
        currency add(const currency&) const;
        currency& increment(const currency&);
        void output() const;
    private:
        signType sign;           // sign of object
        unsigned long dollars;   // number of dollars
        unsigned int cents;      // number of cents
};
```

Program 1.13 Declaration of the class currency

The next function, `~currency`, has a name that is the class name preceded by a tilde (`~`). This function is called the **destructor**. It is automatically invoked whenever an object of type `currency` goes out of scope. The object is destroyed using this function. In our case the destructor is defined as the null function (`{}`). For other classes, the class constructor might create dynamic arrays (for example) and the destructor will need to free the space allocated to these arrays when the object goes out of scope. Destructor functions are not permitted to return a value.

The next two functions allow the user to set the value of a `currency` object. The first requires the user to provide three parameters, while the second permits setting the value by providing a single number. The implementations are provided later in this section. Notice first that both functions have the same name. The compiler and user are able to tell the functions apart because they have different signatures. C++ allows the reuse of function names as long as their signatures are different! Notice also that we have not specified the name of the object whose `sign`, `dollars`, and `cents` values are to be set. This is because the syntax to invoke a class member function is

```
g.setValue(minus,33,0);
h.setValue(20.52);
```

where `g` and `h` are variables of type `currency`. In the first case `g` is the object that invokes `setValue`, while in the second the object `h` invokes `setValue`. When we write the code for the `setValue` functions, we will have a way to access the object that invoked them. Therefore we do not need to include the name of the invoking object in the parameter list.

The functions `getSign`, `getDollars`, and `getCents` return the appropriate data member of the invoking object. The key word `const` states that these functions do not change the invoking object. We refer to functions of this type as **constant functions**.

The function `add` sums the currency amounts of the invoking object and the parameter currency object and then returns the resultant amount. Since this function does not change the invoking object, `add` is a constant function. The function `increment` adds the parameter currency object to the invoking object. This function changes the invoking object and so is not a constant function. The last public member function is `output`, which displays the invoking object by inserting it into the output stream `cout`. The function `output`, which does not change the invoking object, is a constant function.

Although both `add` and `increment` return objects of type `currency`, `add` does a value return, while `increment` does a reference return. As mentioned in Section 1.2.5, value and reference returns work like value and reference parameters. In the case of a value return, the object being returned is copied into the return environment. A reference return avoids this copying, and the return environment makes direct use of the return object. Reference returns are faster than value returns, as no copying is done. The code for `add` shows that it returns a local object, which is destroyed when the function terminates. Therefore the `return` statement must copy this object. In the case of `increment`, a global object is returned and there is no need to copy it.

A **copy constructor** performs the copying for value returns as well as for value parameters. Program 1.13 does not specify a copy constructor, so C++ uses the default copy constructor, which copies the data members only. The use of the default copy constructor is adequate for the class `currency`. We will also see classes where the use of the default copy constructor is not sufficient.

In the `private` section, we declared the three data members needed to represent an object of type `currency`. Each object of type `currency` has its own copy of these three data members.

The functions whose implementation is not given inside the class declaration must be defined outside of it by using the scope resolution operator `::` to specify that the function we are defining is a member of the class `currency`. So the syntax `currency::currency` denotes the constructor of the class `currency`, while `currency::output` denotes the output function of this class. Program 1.14 gives the `currency` constructor, which simply invokes the three-parameter `setValue` function

to initialize the object's data members.

```
currency::currency(signType theSign, unsigned long theDollars,
                   unsigned int theCents)
{ // Create a currency object.
    setValue(theSign, theDollars, theCents);
}
```

Program 1.14 Constructor for currency

Program 1.15 gives the codes for the two `setValue` functions. The first function validates the input and sets the `private` data members of the invoking object only if the parameter values pass the validation test. In case the parameters do not pass the validation test, an exception of type `illegalParameterValue` is thrown (defined later in Section 1.6). The second function does not perform validation and uses only the first two digits after the decimal point. Numbers of the form $d_1.d_2d_3$ may not have an exact computer representation. For example, the computer representation of the number 5.29 is slightly smaller than 5.29. This representation creates an error when extracting the cents component using the following statement:

```
cents = (unsigned int) ((theAmount - dollars) * 100);
```

`(theAmount - dollars) * 100` is slightly smaller than 29, and when the program does the conversion to an unsigned integer, it assigns `cents` the value 28 rather than 29. Adding 0.001 to `theAmount` solves our problem so long as the computer representation of $d_1.d_2d_3$ is not less by more than 0.001 or more by ≥ 0.009 . For example, if the computer representation of 5.29 is equivalent to 5.28999, then adding 0.001 yields 5.29099 and the computed cents amount is 29.

Program 1.16 gives the code for the `add` function. This function begins by converting into integers the two currency values to be added. The amount \$2.32 becomes the integer 232, and -\$4.75 becomes the integer -475. Notice the difference in syntax used to reference the data members of the invoking object and those of the parameter `x`. `x.dollars` specifies the `dollars` data member of `x`, while the use of `dollars` with no object name before it refers to the `dollars` member of the invoking object. When function `add` terminates, the local variables `a1`, `a2`, `a3`, and `ans` are destroyed by the destructor for `long` and the space allocated to these variables freed. Since the currency object `ans` is to be returned as the value of the invocation, it must be copied into the invoking environment. So `add` must do a value return.

Program 1.17 gives the `increment` and `output` codes. In C++, the reserved word `this` points to the invoking object; `*this` is the invoking object itself. Consider the invocation `g.increment(h)`. The first line of function `increment` invokes the `public` member function `add`, which adds `x` (i.e., `h`) to the invoking object `g`.

```
void currency::setValue(signType theSign, unsigned long theDollars,
                        unsigned int theCents)
{ // Set currency value.
  if (theCents > 99)
    // too many cents
    throw illegalParameterValue("Cents should be < 100");

  sign = theSign;
  dollars = theDollars;
  cents = theCents;
}

void currency::setValue(double theAmount)
{ // Set currency value.
  if (theAmount < 0) {sign = minus;
                      theAmount = -theAmount;}
  else sign = plus;

  dollars = (unsigned long) theAmount;
  // extract integer part
  cents = (unsigned int) ((theAmount + 0.001 - dollars) * 100);
  // get two decimal digits
}
```

Program 1.15 Setting the private data members

The result is returned and assigned to the object `*this`, which is `g`. So the value of `g` is incremented by `h`. The function returns `*this`, which is the invoking object. Since this object is not local to function `increment`, it will not be automatically destroyed upon termination of the function. Hence we may do a reference return and save the copying that would take place during a value return.

By making the data members of the class `currency` private, we deny access to these members to the user. So the user cannot change their values using statements such as

```
h.cents = 20;
h.dollars = 100;
h.sign = plus;
```

We can assure the integrity of the data members by writing the member functions to leave behind valid values if they begin with valid data member values. Our codes for the constructor and `setValue` functions validate the data before using it. The

```
currency currency::add(const currency& x) const
{// Add x and *this.
    long a1, a2, a3;
    currency result;
    // convert invoking object to signed integers
    a1 = dollars * 100 + cents;
    if (sign == minus) a1 = -a1;

    // convert x to signed integer
    a2 = x.dollars * 100 + x.cents;
    if (x.sign == minus) a2 = -a2;

    a3 = a1 + a2;

    // convert to currency representation
    if (a3 < 0) {result.sign = minus; a3 = -a3;}
    else result.sign = plus;
    result.dollars = a3 / 100;
    result.cents = a3 - result.dollars * 100;

    return result;
}
```

Program 1.16 Adding two currency values

```
currency& currency::increment(const currency& x)
{// Increment by x.
    *this = add(x);
    return *this;
}

void currency::output() const
{// Output currency value.
    if (sign == minus) cout << '-';
    cout << '$' << dollars << '.';
    if (cents < 10) cout << '0';
    cout << cents;
}
```

Program 1.17 increment and output

remaining functions have the property they leave behind valid data if they start with valid data. As a result, the codes for functions such as `add` and `output` do not need to verify that the number of cents is, in fact, between 0 and 99. If the data members are declared as `public` members, their integrity cannot be assured. The user might (erroneously) set `cents` equal to 305, which would cause functions such as `output` to malfunction. As a result, all functions would need to validate the data before proceeding with their tasks. This validation would slow down the codes and also make them less elegant.

Program 1.18 gives a sample application of the class `currency`. This code assumes that the class declaration and all implementation codes are in the file `currency.h`. We would normally keep the class declaration and the function implementations in separate files. However, such a separation causes difficulties with template functions and template classes that we use heavily in subsequent sections and chapters.

Line 1 of the function `main` declares four variables, `g`, `h`, `i`, and `j` of type `currency`. The class constructor initializes all but `h` to \$0.00. `h` has the initial value \$3.50. In the two calls to `setValue`, `g` and `i` are, respectively, set to -\$2.25 and -\$6.45. The call to function `add` adds `h` and `g` and returns the resulting object whose value is \$1.25. The returned object is assigned to `j`, using the default assignment procedure that copies the data members of the object on the right side into the corresponding data members of the object on the left side. This copying results in `j` having the value \$1.25. These values of `h`, `g` and `j` are output by the next few lines of code. The remaining lines of code are self explanatory.

1.5.2 Using a Different Representation

Suppose that many application codes have been developed using the class `currency` of Program 1.13. Now we desire to change the representation of a currency object to one that results in faster codes for the more frequently performed operations of `add` and `increment` and hence speed the application codes. Since the user interacts with the class `currency` only through the interface provided in the `public` section, changes made to the `private` section do not affect the correctness of the application codes. Hence we can change the `private` section without making any changes in the applications!

The new representation of a currency object has just one private data member, which is of type `long`. The number 132 represents \$1.32, while -20 represents -\$0.20. Programs 1.19, 1.20, and 1.21 give the new declaration of `currency` and the implementation of the various member functions.

Notice that if the new code is placed in the file `currency.h`, we can run the application code of Program 1.18 with no change at all! *An important benefit of hiding the implementation details from the user is that we can replace old representations with new more efficient ones without changing the application codes.*

```
#include <iostream>
#include "currency.h"

using namespace std;

int main()
{
    currency g, h(plus, 3, 50), i, j;

    // try out both forms of setValue
    g.setValue(minus, 2, 25);
    i.setValue(-6.45);

    // do an add and output
    j = h.add(g);
    h.output();
    cout << " + ";
    g.output();
    cout << " = ";
    j.output(); cout << endl;

    // do two adds in a sequence
    j = i.add(g).add(h);
    // output statements omitted

    // do an increment and add
    j = i.increment(g).add(h);
    // output statements omitted

    // test the exception
    cout << "Attempting to initialize with cents = 152" << endl;
    try {i.setValue(plus, 3, 152);}
    catch (illegalParameterValue e)
    {
        cout << "Caught thrown exception" << endl;
        e.outputMessage();
    }
    return 0;
}
```

Program 1.18 Application of the class currency

```
class currency
{
    public:
        // constructor
        currency(signType theSign = plus,
                  unsigned long theDollars = 0,
                  unsigned int theCents = 0);
        // destructor
        ~currency() {}
        void setValue(signType, unsigned long, unsigned int);
        void setValue(double);
        signType getSign() const
        {if (amount < 0) return minus;
         else return plus;}
        unsigned long getDollars() const
        {if (amount < 0) return (-amount) / 100;
         else return amount / 100;}
        unsigned int getCents() const
        {if (amount < 0) return -amount - getDollars() * 100;
         else return amount - getDollars() * 100;}
        currency add(const currency&) const;
        currency& increment(const currency& x)
        {amount += x.amount; return *this;}
        void output() const;
    private:
        long amount;
};
```

Program 1.19 New declaration of the class `currency`

1.5.3 Operator Overloading

The class `currency` includes several member functions that resemble some of the standard operators of C++. For example, `add` does what `+` does, and `increment` does what `+=` does. Using these standard C++ operators is more natural than defining new ones such as `add` and `increment`. We can use `+` and `+=` by a process called **operator overloading** that permits us to extend the applicability of existing C++ operators so that they work with new data types or classes.

Program 1.22 gives the class declaration that substitutes the standard operators `+` and `+=` for `add` and `increment`. The `output` function now takes the name of an output stream as a parameter. Program 1.23 gives the new codes for `add` and `output`. This program also includes code to overload the C++ stream insertion

```
currency::currency(signType theSign, unsigned long theDollars,
                  unsigned int theCents)
{ // Create a currency object.
    setValue(theSign, theDollars, theCents);
}

void currency::setValue(signType theSign, unsigned long theDollars,
                      unsigned int theCents)
{ // Set currency value.
    if (theCents > 99)
        // too many cents
        throw illegalParameterValue("Cents should be < 100");

    amount = theDollars * 100 + theCents;
    if (theSign == minus) amount = -amount;
}

void currency::setValue(double theAmount)
{ // Set currency value.
    if (theAmount < 0)
        amount = (long) ((theAmount - 0.001) * 100);
    else
        amount = (long) ((theAmount + 0.001) * 100);
        // 2 decimal digits only
}
```

Program 1.20 New constructor and set value codes

operator <<.

Notice that we overload the stream insertion operator without declaring a corresponding member function in the class `currency`, and overload + and += by defining these operators as members of the class. We can also overload the stream extraction operator `>>` without defining this operator as a class member. Further, notice the use of the function `output` to assist in the overloading of `<<`. Since the `private` members of `currency` objects are not accessible from functions that are not class members (the overloaded `<<` is not a class member, while the overloaded `+` is), the code that overloads `<<` may not reference the `private` members of the object `x` that it is to insert into the output stream. In particular, the code

```
// overload <<
ostream& operator<<(ostream& out, const currency& x)
    {out << x.amount; return out;}
```

```
currency currency::add(const currency& x) const
{// Add x and *this.
    currency y;
    y.amount = amount + x.amount;
    return y;
}

void currency::output() const
{// Output currency value.
    long theAmount = amount;
    if (theAmount < 0) {cout << '-';
                           theAmount = -theAmount;}
    long dollars = theAmount / 100; // dollars
    cout << '$' << dollars << '.';
    int cents = theAmount - dollars * 100; // cents
    if (cents < 10) cout << '0';
    cout << cents;
}
```

Program 1.21 New code for add and output

is erroneous, as the member `amount` is not accessible.

Program 1.24 is a version of Program 1.18 that assumes that operators have been overloaded and that the codes of Programs 1.22 and 1.23 are in the file `currencyOverload.h`.

1.5.4 Friends and Protected Class Members

As pointed out earlier, `private` members of a class are visible only to class member functions. In some applications, we must grant access to these `private` members to other classes and functions. This access may be granted by declaring these other classes or functions as `friends`.

In our `currency` class example (Program 1.22), we defined a member function `output` to facilitate the overloading of the operator `<<`. Defining this member function was necessary, as the function

```
ostream& operator<<(ostream&, const currency&)
```

cannot access the `private` member `amount`. We may avoid defining the additional function `output` by declaring `ostream& operator<<` a friend of the class `currency`. Thus we grant this function access to all members (`private` and `public`) of `currency`. To make friends, we introduce `friend` statements into the declaration

```
class currency
{
    public:
        // constructor
        currency(signType theSign = plus,
                  unsigned long theDollars = 0,
                  unsigned int theCents = 0);
        // destructor
        ~currency() {}
        void setValue(signType, unsigned long, unsigned int);
        void setValue(double);
        signType getSign() const
            {if (amount < 0) return minus;
             else return plus;}
        unsigned long getDollars() const
            {if (amount < 0) return (-amount) / 100;
             else return amount / 100;}
        unsigned int getCents() const
            {if (amount < 0) return -amount - getDollars() * 100;
             else return amount - getDollars() * 100;}
        currency operator+(const currency&) const;
        currency& operator+=(const currency& x)
            {amount += x.amount; return *this;}
        void output(ostream&) const;
    private:
        long amount;
};
```

Program 1.22 Class declaration using operator overloading

of the class `currency`. For consistency, we shall always place `friend` statements just after the `class` header statement as in

```
class currency {
    friend ostream& operator<<(ostream&, const currency&);
    public:
```

With this `friend` declaration in place, we may overload the `<<` operator using the code of Program 1.25. When the `private` members of `currency` are changed, we will need to examine its friends and make appropriate changes.

Later we shall see how a class A may be derived from another class B. Class A is called the **derived class**, and B is the **base class**. The derived class will

```
currency currency::operator+(const currency& x) const
{// Add x and *this.
    currency result;
    result.amount = amount + x.amount;
    return result;
}

void currency::output(ostream& out) const
{// Insert currency value into stream out.
    long theAmount = amount;
    if (theAmount < 0) {out << '-';
                           theAmount = -theAmount;}
    long dollars = theAmount / 100; // dollars
    out << '$' << dollars << '.';
    int cents = theAmount - dollars * 100; // cents
    if (cents < 10) out << '0';
    out << cents;
}

// overload <<
ostream& operator<<(ostream& out, const currency& x)
{x.output(out); return out;}
```

Program 1.23 Codes for +, output, and <<

need access to some or all of the data members of the base class. To facilitate granting this access, C++ allows for a third category of members called **protected**. Protected members behave like **private** members except that derived classes can access **protected** members.

Class members that are to be accessible by user applications should be declared **public** members. Data members should never be in this category. The remaining members should be divided between the categories **protected** and **private**. Good software-engineering principles dictate that data members remain private. By adding member functions to access and change the value of data members, derived classes obtain indirect access to the data members of the base class. At the same time, we can change the implementation of the base class without having to change its derived classes.

1.5.5 Addition of #ifndef, #define, and #endif Statements

The entire contents of the file **currency.h** (or **currencyOverload.h**) that contains the declaration and implementation of the class **currency** should be preceded by

```
#include <iostream>
#include "currencyOverload.h"

using namespace std;

int main()
{
    currency g, h(plus, 3, 50), i, j;

    // try out both forms of setValue
    g.setValue(minus, 2, 25);
    i.setValue(-6.45);

    // do an add and output
    j = h + g;
    cout << h << " + " << g << " = " << j << endl;

    // do two adds in a sequence
    j = i + g + h;
    cout << i << " + " << g << " + "
        << h << " = " << j << endl;

    // do an increment and add
    cout << "Increment " << i << " by " << g
        << " and then add " << h << endl;
    j = (i += g) + h;
    cout << "Result is " << j << endl;
    cout << "Incremented object is " << i << endl;

    // test the exception
    cout << "Attempting to initialize with cents = 152" << endl;
    try {i.setValue(plus, 3, 152);}
    catch (illegalParameterValue e)
    {
        cout << "Caught thrown exception" << endl;
        e.outputMessage();
    }
    return 0;
}
```

Program 1.24 Using overloaded operators

```
// overload <<
ostream& operator<<(ostream& out, const currency& x)
{// Insert currency value into stream out.
    long theAmount = x.amount;
    if (theAmount < 0) {out << '-';
                           theAmount = -theAmount;}
    long dollars = theAmount / 100; // dollars
    out << '$' << dollars << '.';
    int cents = theAmount - dollars * 100; // cents
    if (cents < 10) out << '0';
    out << cents;
    return out;
}
```

Program 1.25 Overloading the friend <<

the statements

```
#ifndef currency_
#define currency_
```

and followed by the statement

```
#endif
```

These statements ensure that the code for `currency` gets included and compiled only once per program. *You should add corresponding statements to the program listings provided for the remaining class definitions in this book.*

EXERCISES

15. (a) What are the maximum and minimum currency values permissible when the representation of Program 1.13 is used? Assume that objects of type `unsigned long` and `unsigned int` are represented using 4 bytes. So objects of these types have the range 0 through $2^{32} - 1$.
(b) What are the maximum and minimum currency values permissible when the representation of Program 1.13 is used and the data types of `dollars` and `cents` are changed to `int`?
(c) If function `add` (Program 1.16) is used to add two currency amounts, what are their largest possible values so that no error occurs when converting from type `currency` to type `long` as is done to set `a1` and `a2`?

16. Extend the class `currency` of Program 1.13 by adding the following public member functions:

- (a) `input()` inputs a currency value from the standard input stream and assign it to the invoking object.
- (b) `subtract(x)` subtracts the value of the currency object `x` from that of the invoking object and returns the result.
- (c) `percent(x)` returns a currency object whose value is `x` percent of the value of the invoking object. The data type of `x` is `double`.
- (d) `multiply(x)` returns the currency object that results from multiplying the invoking object and the number `x`, which is of type `double`.
- (e) `divide(x)` returns the currency object that results from dividing the invoking object by the number `x`, which is of type `double`.

Implement all member functions and test their correctness using suitable test data.

17. Do Exercise 16 using the implementation of Program 1.19.

18. (a) Do Exercise 16 using the implementation of Program 1.22. Overload the operators `>>`, `-`, `%`, `*`, and `/`. When overloading `>>`, declare it as a friend function and do not define a public input function to facilitate the input.
(b) Replace the two `setValue` functions by overloading the assignment operator `=`. An overload of the type `operator=(int x)` that assigns an integer to an object of type `currency` should replace the three-parameter `setValue` function. `x` represents the sign, dollar amount, and cents rolled into a single integer. An overload of the type `operator=(double x)` should replace the single-parameter `setValue` function.

1.6 THE EXCEPTION CLASS `illegalParameterValue`

Program 1.26 shows a user-defined class `illegalParameterValue` that may be used when signaling errors in which the value of an actual parameter to a function is improper. Program 1.27 is a version of Program 1.8 in which an exception of type `illegalParameterValue` is thrown instead of an exception of type `char*`. Program 1.28 shows how to catch an exception of type `illegalParameterValue`.

1.7 RECURSION

A **recursive function** or method invokes itself. In **direct recursion** the code for function `f` contains a statement that invokes `f`, whereas in **indirect recursion**

```
class illegalParameterValue
{
public:
    illegalParameterValue() :
        message("Illegal parameter value") {}
    illegalParameterValue(char * theMessage)
        {message = theMessage;}
    void outputMessage() {cout << message << endl;}
private:
    char * message;
};
```

Program 1.26 Defining an exception class

```
int abc(int a, int b, int c)
{
    if (a <= 0 || b <= 0 || c <= 0)
        throw illegalParameterValue ("All parameters should be > 0");
    return a + b * c;
}
```

Program 1.27 Throwing an exception of type illegalParameterValue

```
int main()
{
    try {cout << abc(2,0,4) << endl;}
    catch (illegalParameterValue e)
    {
        cout << "The parameters to abc were 2, 0, and 4" << endl;
        cout << "illegalParameterValue exception thrown" << endl;
        e.outputMessage();
        return 1;
    }
    return 0;
}
```

Program 1.28 Catching an exception of type illegalParameterValue

the function f invokes some other function g , which invokes yet another function h , and so on until function f is again invoked. Before delving into recursive C++ functions, we examine two related concepts from mathematics—recursive definitions of mathematical functions and proofs by induction.

1.7.1 Recursive Mathematical Functions

In mathematics we often define a function in terms of itself. For example, the factorial function $f(n) = n!$, for n an integer, is defined as follows:

$$f(n) = \begin{cases} 1 & n \leq 1 \\ nf(n-1) & n > 1 \end{cases} \quad (1.1)$$

This definition states that $f(n)$ equals 1 whenever n is less than or equal to 1; for example, $f(-3) = f(0) = f(1) = 1$. However, when n is more than 1, $f(n)$ is defined recursively, as the definition of f now contains an occurrence of f on the right side. This use of f on the right side does not result in a circular definition, as the parameter of f on the right side is smaller than that on the left side. For example, from Equation 1.1 we obtain $f(2) = 2f(1)$. From Equation 1.1 we also obtain $f(1) = 1$, and substituting for $f(1)$ in $f(2) = 2f(1)$, we obtain $f(2) = 2$. Similarly, from Equation 1.1 we obtain $f(3) = 3f(2)$. We have already seen that Equation 1.1 yields $f(2) = 2$. So $f(3) = 3 * 2 = 6$.

For a recursive definition of $f(n)$ (we assume direct recursion) to be a complete specification of f , it must meet the following requirements:

- The definition must include a **base** component in which $f(n)$ is defined directly (i.e., nonrecursively) for one or more values of n . For simplicity, we assume that the domain of f is the nonnegative integers and that the base covers the case $0 \leq n \leq k$ for some constant k . (It is possible to have recursive definitions in which the base covers the case $n \geq k$ instead, but we encounter these definitions less frequently.)
- In the **recursive component** all occurrences of f on the right side should have a parameter smaller than n so that repeated application of the recursive component transforms all occurrences of f on the right side to occurrences of f in the base.

In Equation 1.1 the base is $f(n) = 1$ for $n \leq 1$; in the recursive component $f(n) = nf(n-1)$, the parameter of f on the right side is $n-1$, which is smaller than n . Repeated application of the recursive component transforms $f(n-1)$ to $f(n-2)$, $f(n-3)$, ..., and finally to $f(1)$ which is included in the base. For example, repeated application of the recursive component gives the following:

$$f(5) = 5f(4) = 20f(3) = 60f(2) = 120f(1)$$

Notice that each application of the recursive component gets us closer to the base. Finally, an application of the base gives $f(5) = 120$. From the example, we see that $f(n) = n(n - 1)(n - 2) \cdots 1$ for $n \geq 1$.

As another example of a recursive definition, consider the Fibonacci numbers that are defined recursively as below:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n > 1 \quad (1.2)$$

In this definition, $F_0 = 0$ and $F_1 = 1$ make up the base component, and $F_n = F_{n-1} + F_{n-2}$ is the recursive component. The function parameters on the right side are smaller than n . For Equation 1.2 to be a complete recursive specification of F , repeated application of the recursive component beginning with any value of $n > 1$ should transform all occurrences of F on the right side to occurrences in the base. Since repeated subtraction of 1 or 2 from an integer $n > 1$ reduces it to either 0 or 1, right-side occurrences of F are always transformed to base occurrences. For example, $F_4 = F_3 + F_2 = F_2 + F_1 + F_0 = 3F_1 + 2F_0 = 3$.

1.7.2 Induction

Now, we turn our attention to the second concept related to recursive computer functions—proofs by induction. In a proof by induction, we establish the validity of a claim such as

$$\sum_{i=0}^n i = n(n + 1)/2, n \geq 0 \quad (1.3)$$

by showing that the claim is true for one or more base values of n (generally, $n = 0$ suffices); we assume the claim is true for values of n from 0 through m where m is an arbitrary integer greater than or equal to the largest n covered in the base; and finally using this assumption, we show the claim is true for the next value of n (i.e., $m + 1$). This methodology leads to a proof that has three components—**induction base**, **induction hypothesis**, and **induction step**.

Suppose we are to prove Equation 1.3 by induction on n . In the induction base we establish correctness for $n = 0$. At this time the left side is $\sum_{i=0}^0 i = 0$, and the right side is also 0. So Equation 1.3 is valid when $n = 0$. In the induction hypothesis we assume the equation is valid for $n \leq m$ where m is an arbitrary integer ≥ 0 . (For the ensuing induction step proof, it is sufficient to assume that Equation 1.3 is valid only for $n = m$.) In the induction step we show that the equation is valid for $n = m + 1$. For this value of n , the left side is $\sum_{i=0}^{m+1} i$, which equals $m + 1 + \sum_{i=0}^m i$. From the induction hypothesis we get $\sum_{i=0}^m i = m(m + 1)/2$. So when $n = m + 1$, the left side becomes $m + 1 + m(m + 1)/2 = (m + 1)(m + 2)/2$, which equals the right side.

At first glance, a proof by induction appears to be a circular proof—we establish a result assuming it is correct. However, a proof by induction is not a circular proof

for the same reasons that a recursive definition is not circular. A correct proof by induction has an induction base similar to the base component of a recursive definition, and the induction step proves correctness using correctness for smaller values of n . Repeated application of the induction step reduces the proof to one that is solely in terms of the base.

1.7.3 Recursive C++ Functions

C++ allows us to write recursive functions. A proper recursive function must include a base component. The recursive component of the function should use smaller values of the function parameters so that repeated invocation of the function results in parameters equal to those included in the base component.

Example 1.1 [Factorial] Program 1.29 gives a C++ recursive function that uses Equation 1.1 to compute $n!$. The base component covers the cases when $n \leq 1$. Consider the invocation `factorial(2)`. To compute `2 * factorial(1)` in the `else` statement, the computation of `factorial(2)` is suspended and `factorial` invoked with $n = 1$. When the computation of `factorial(2)` is suspended, the program state (i.e., values of local variables and value formal parameters, bindings of reference formal parameters, location in code, etc.) is saved in a recursion stack. This state is restored when the computation of `factorial(1)` completes. The invocation `factorial(1)` returns the value 1. The computation of `factorial(2)` resumes, and the expression `2 * 1` is computed.

```
int factorial(int n)
{// Compute n!
 if (n <= 1) return 1;
 else return n * factorial(n - 1);
}
```

Program 1.29 Recursive method to compute $n!$

When computing `factorial(3)`, the computation is suspended when the `else` statement is reached so that `factorial(2)` may be computed. We have already seen how the invocation `factorial(2)` works to produce the result 2. When the computation of `factorial(2)` completes, the computation of `factorial(3)` resumes and the expression `3 * 2` is computed.

Because of the similarity between the code of Program 1.29 and Equation 1.1, the correctness of the code follows from the correctness of the equation. ■

Example 1.2 The template function `sum` (Program 1.30) computes the sum of elements `a[0]` through `a[n-1]` (abbreviated `a[0:n-1]`). When n is 0, the method returns the value 0.

```
template<class T>
T sum(T a[], int n)
{// Return sum of the numbers a[0:n-1].
    T theSum = 0;
    for (int i = 0; i < n; i++)
        theSum += a[i];
    return theSum;
}
```

Program 1.30 Add a[0:n-1]

Program 1.31 is a recursive method to compute the sum of the elements a[0:n-1]. The code for rSum results from a recursive formulation of the problem—when n is 0, the sum is 0; when n is greater than 0, the sum of n elements is the sum of the first $n - 1$ elements plus the last element. ■

```
template<class T>
T rSum(T a[], int n)
{// Return sum of numbers a[0:n - 1].
    if (n > 0)
        return rSum(a, n-1) + a[n-1];
    return 0;
}
```

Program 1.31 Recursive code to add a[0:n-1]

Example 1.3 [Permutations] Often we wish to examine all permutations of n distinct elements to determine the best one. For example, the permutations of the elements a , b , and c are abc , acb , bac , bca , cba , and cab . The number of permutations of n elements is $n!$.

Although developing a nonrecursive C++ function to output all permutations of n elements is quite difficult, we can develop a recursive one with modest effort. Let $E = \{e_1, \dots, e_n\}$ denote the set of n elements whose permutations are to be generated; let E_i be the set obtained by removing element i from E ; let $\text{perm}(X)$ denote the permutations of the elements in set X ; and let $e_i.\text{perm}(X)$ denote the permutation list obtained by prefixing each permutation in $\text{perm}(X)$ with element e_i . For example, if $E = \{a, b, c\}$, then $E_1 = \{b, c\}$, $\text{perm}(E_1) = (bc, cb)$, and $e_1.\text{perm}(E_1) = (abc, acb)$.

For the recursion base, we use $n = 1$. Since only one permutation is possible when we have only one element, $\text{perm}(E) = (e)$ where e is the lone element in E .

When $n > 1$, $\text{perm}(E)$ is the list $e_1.\text{perm}(E_1)$ followed by $e_2.\text{perm}(E_2)$ followed by $e_3.\text{perm}(E_3) \dots$ followed by $e_n.\text{perm}(E_n)$. This recursive definition of $\text{perm}(E)$ defines $\text{perm}(E)$ in terms of n $\text{perm}(X)$ s, each of which involves an X with $n - 1$ elements. Both the base component and recursive component requirements of a complete recursive definition are satisfied.

When $n = 3$ and $E = (a, b, c)$, the preceding definition of $\text{perm}(E)$ yields $\text{perm}(E) = a.\text{perm}(\{b, c\}), b.\text{perm}(\{a, c\}), c.\text{perm}(\{b, a\})$. From the recursive definition $\text{perm}(\{b, c\})$ is $b.\text{perm}(\{c\}), c.\text{perm}(\{b\})$. So $a.\text{perm}(\{b, c\})$ is $ab.\text{perm}(\{c\}), ac.\text{perm}(\{b\}) = ab.c, ac.b = (abc, acb)$. Proceeding in a similar way, we obtain $b.\text{perm}(\{a, c\})$ is $ba.\text{perm}(\{c\}), bc.\text{perm}(\{a\}) = ba.c, bc.a = (bac, bca)$ and $c.\text{perm}(\{b, a\})$ is $cb.\text{perm}(\{a\}), ca.\text{perm}(\{b\}) = cb.a, ca.b = (cba, cab)$. So $\text{perm}(E) = (abc, acb, bac, bca, cba, cab)$.

Notice that $a.\text{perm}(\{b, c\})$ is actually the two permutations abc and acb . a is the prefix of these permutations, and $\text{perm}(\{b, c\})$ gives their suffixes. Similarly, $ac.\text{perm}(\{b\})$ denotes permutations whose prefix is ac and whose suffixes are the permutations $\text{perm}(\{b\})$.

Program 1.32 transforms the preceding recursive definition of $\text{perm}(E)$ into a C++ function. This code outputs all permutations whose prefix is `list[0:k-1]` and whose suffixes are the permutations of `list[k:m]`. The invocation `permutations(list, 0, n-1)` outputs all $n!$ permutations of `list[0:n-1]`. With this invocation, `k` is 0 and `m` is `n-1`. So the prefix of the generated permutations is null, and their suffixes are the permutations of `list[0:n-1]`. When `k` equals `m`, there is only one suffix `list[m]`, and now `list[0:m]` defines a permutation that is to be output. When `k < m`, the `else` clause is executed. Let E denote the elements in `list[k:m]` and let E_i be the set obtained by removing $e_i = list[i]$ from E . The first `swap` in the `for` loop has the effect of setting `list[k] = e_i` and `list[k+1:m] = E_i`. Therefore, the following call to `permutations` computes $e_i.\text{perm}(E_i)$. The second `swap` restores `list[k:m]` to its state prior to the first `swap`.

Figure 1.3 shows the progress of Program 1.32 when invoked with `k = 0`, `m = 2`, and `list[0:2] = [a, b, c]`. The figure shows the contents of `list[0:2]` immediately after each call to `permutations`, as well as immediately after the second `swap` is done following a return from `permutations`. The unshaded entries denote `list[0:k-1]`, and the shaded entries denote `list[k:m]`. Configuration numbers are shown outside the array. Each edge of Figure 1.3 is traversed twice during the execution of `permutations(list, 0, 2)`: once when a call to `permutations` is made in the `for` loop and once when a return from `permutations` is made.

We begin with configuration 1. The first `swap` done in the `for` loop has no effect on the array; configuration 2 shows the state just after `permutations` is invoked from within the `for` loop. From configuration 2 we move to configuration 3. Configuration 3 is output because, in this configuration, `k = m`. Following this output, we make a return and execute the second `swap` statement in the `for` loop. As a result, we restore configuration 2. From configuration 2 we move forward to configuration 4, and the permutation `acb` is output. Then we back up through earlier configura-

```
template<class T>
void permutations(T list[], int k, int m)
{// Generate all permutations of list[k:m].
    int i;
    if (k == m) {// list[k:m] has one permutation, output it
        copy(list, list+m+1,
              ostream_iterator<T>(cout, ""));
        cout << endl;
    }
    else // list[k:m] has more than one permutation
        // generate these recursively
        for (i = k; i <= m; i++)
    {
        swap(list[k], list[i]);
        permutations(list, k+1, m);
        swap(list[k], list[i]);
    }
}
```

Program 1.32 Recursive method for permutations

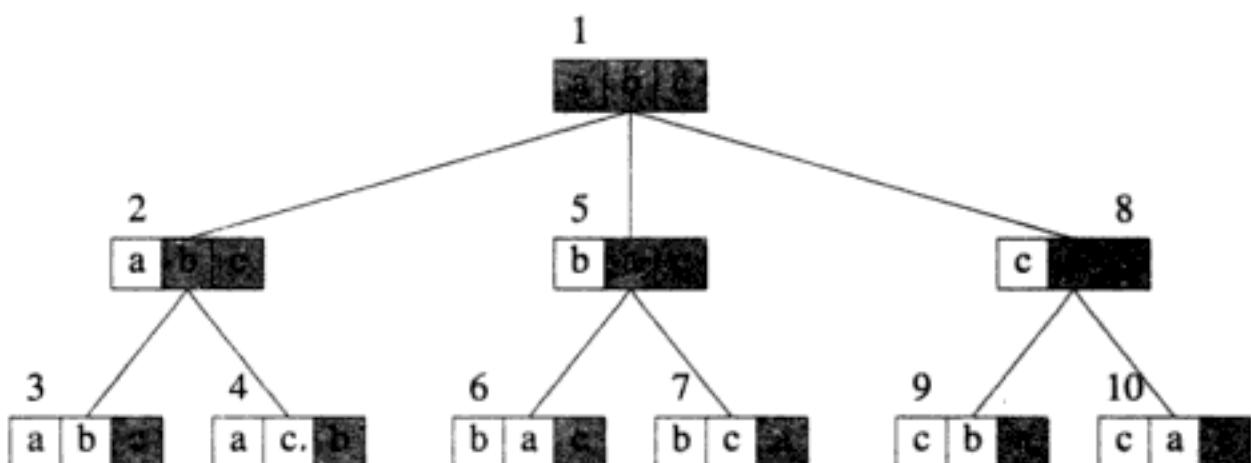


Figure 1.3 Generating the permutations of abc

tions until we can move forward again. We back up through configurations 2 and 1. From configuration 1 we move forward to configurations 5 and 6. The sequence of configurations encountered in the complete execution is 1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 7, 5, 1, 8, 9, 8, 10, 8, 1. ■

EXERCISES

19. Write a nonrecursive function to compute $n!$. Test your code.
20. (a) Write a recursive function to compute the Fibonacci number F_n . Test your code.
- (b) Show that your code for part (a) computes the same F_i more than once when it is invoked to compute F_n for any $n > 2$.
- (c) Write a nonrecursive function to compute the Fibonacci number F_n . Your code should compute each Fibonacci number just once. Test your code.
21. Consider the function f , which is defined in Equation 1.4. n is a nonnegative integer.

$$f(n) = \begin{cases} n/2 & n \text{ is even} \\ f(3n+1) & n \text{ is odd} \end{cases} \quad (1.4)$$

- (a) Use Equation 1.4 to manually compute $f(5)$ and $f(7)$.
- (b) Identify the base and recursive components of the function definition. Show that repeated application of the recursive component transforms the occurrence of f on the right side to the occurrence of f in the base component.
- (c) Write a recursive C++ function to compute $f(n)$. Test your code.
- (d) Use your proof for part (b) to arrive at a nonrecursive C++ function to compute f . Your code should have no loops. Test your code.
22. [Ackermann's Function] Equation 1.5 defines Ackermann's function. In this definition, i and j are integers that are ≥ 1 .

$$A(i, j) = \begin{cases} 2^j & i = 1 \text{ and } j \geq 1 \\ A(i - 1, 2) & i \geq 2 \text{ and } j = 1 \\ A(i - 1, A(i, j - 1)) & i, j \geq 2 \end{cases} \quad (1.5)$$

- (a) Use Equation 1.5 to manually compute $A(1, 2)$, $A(2, 1)$, and $A(2, 2)$.
- (b) Identify the base and recursive components of the function definition.
- (c) Write a recursive C++ function to compute $A(i, j)$. Test your code.

23. [GCD] The **greatest common divisor (GCD)** of two nonnegative integers x and y is 0 when exactly one of them is 0. When at least one of x and y is nonzero, their GCD, $\text{gcd}(x, y)$, is the greatest integer that evenly divides both. So $\text{gcd}(0, 0) = 0$, $\text{gcd}(10, 0) = \text{gcd}(0, 10) = 10$, and $\text{gcd}(20, 30) = 10$. Euclid's GCD algorithm is a recursive algorithm that is believed to date back to 375 B.C.; it is perhaps the earliest example of a recursive algorithm. Euclid's algorithm implements the recursive definition given in Equation 1.6.

$$\text{gcd}(x, y) = \begin{cases} x & y = 0 \\ \text{gcd}(y, x \bmod y) & y > 0 \end{cases} \quad (1.6)$$

In Equation 1.6 **mod** is the modulo operator that is implemented in C++ as the operator `%`. $x \bmod y$ is the remainder of x/y .

- (a) Use Equation 1.6 to manually compute $\text{gcd}(20, 30)$ and $\text{gcd}(112, 42)$.
 - (b) Identify the base and recursive components of the function definition. Show that repeated application of the recursive component transforms the occurrence of gcd on the right side to the occurrence of gcd in the base component.
 - (c) Write a recursive C++ function to compute $\text{gcd}(x, y)$. Test your code.
24. Write a recursive template function to determine whether element x is one of the elements in the array $a[0:n-1]$.
25. [Subset Generation] Write a recursive C++ function to output all subsets of n elements. For example, the subsets of the three-element set $\{a, b, c\}$ are $\{\}$ (empty set), $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$. These subsets may be denoted by the 0/1 vector sequence 000, 100, 010, 001, 110, 101, 011, and 111, respectively (a 0 means that the corresponding element is not in the subset, and a 1 means that it is). So it is sufficient that your C++ code output all 0/1 sequences of length n .
26. [Gray Code] The **Hamming distance** between two vectors is the number of positions in which the vectors differ. For example, the Hamming distance between 100 and 010 is 2. A (binary) Gray code is a subset sequence in which the Hamming distance between every pair of consecutive vectors (also called codes) is 1. The three-element subset sequence given in Exercise 25 is not a Gray code. However, the three-element subset sequence 000, 100, 110, 010, 011, 111, 101, 001 is a Gray code. This sequence also has the property that the first and last vectors differ in exactly one place. In some applications of subset sequences, the cost of going from one subset to the next depends on the Hamming distance between these two subsets. In these applications, we desire a subset sequence that is a Gray code. A Gray code may be compactly

represented by giving the sequence of positions in which the vectors of the code change. For the three-element Gray code given above, the position change sequence is 1, 2, 1, 3, 1, 2, 1. Let $g(n)$ be the position change sequence for a Gray code for n elements. Equation 1.7 gives a recursive definition for $g(n)$.

$$g(n) = \begin{cases} 1 & n = 1 \\ g(n - 1), n, g(n - 1) & n > 1 \end{cases} \quad (1.7)$$

- (a) Use Equation 1.6 to manually compute $g(4)$.
- (b) Identify the base and recursive components of the function definition. Show that repeated application of the recursive component transforms both occurrences of g on the right side to the occurrence of g in the base component.
- (c) Write a recursive C++ function to compute $g(n)$. Test your code.

1.8 THE STANDARD TEMPLATES LIBRARY

The C++ standard templates library (STL) is a collection of containers, adaptors, iterators, function objects (also known as functors) and algorithms. Through the judicious use of elements of the STL, the task of writing application codes is greatly simplified. In this book, we shall often solve a problem first using basic C++ language constructs to illustrate the mechanics of solving the problem. Later, we will show how the same problem can be solved more simply by using an element of the STL.

Example 1.4 [The STL algorithm accumulate] The STL has an algorithm **accumulate** that may be used to sum the elements in a sequence. The syntax is

```
accumulate(start, end, initialValue)
```

where **start** points to the first element to be accumulated and **end** points to one position after the last element to be accumulated. So, elements in the range [**start**, **end**) are accumulated. The invocation

```
accumulate(a, a+n, initialValue)
```

where **a** is a one-dimensional array, for example, returns the value

$$initialValue + \sum_{i=0}^{n-1} a[i]$$

```
template<class T>
T sum(T a[], int n)
{// Return sum of the numbers a[0:n-1].
    T theSum = 0;
    return accumulate(a, a+n, theSum);
}
```

Program 1.33 Summing a[0:n-1] using the STL algorithm `accumulate`

Program 1.33 uses the STL algorithm `accumulate` to provide the same functionality as is provided by Programs 1.30 and 1.31.

The STL algorithm `accumulate` accesses successive elements of the sequence to be summed by performing the `++` operator on `start` and terminating when the pointer value becomes `end`. So, this algorithm may be used to sum the values of any sequence whose elements may be obtained by repeated application of the `++` operator. One-dimensional arrays and the STL container `vector` are two examples of sequences whose elements may be accessed in this way. We shall see other examples later in this book.

The STL has a more general form of the `accumulate` algorithm, which has the following syntax

```
accumulate(start, end, initialValue, operator)
```

where `operator` is a function that defines the operation to be used during the accumulation process. Using the STL functor `multiples`, for example, we can find the product of an array of elements using the code of Program 1.34. ■

```
template<class T>
T product(T a[], int n)
{// Return sum of the numbers a[0:n-1].
    T theProduct = 1;
    return accumulate(a, a+n, theProduct, multiples<T>());
}
```

Program 1.34 Compute the product of the elements a[0:n-1]

Example 1.5 [The STL algorithms `copy` and `next_permutation`] The `copy` algorithm copies a sequence of elements from one location to another. The syntax is

```
copy(start, end, to)
```

where `to` gives the location to which the first element is to be copied. So, elements are copied from locations `start`, `start + 1`, ..., `end - 1` to the locations `to`, `to + 1`, ..., `to + end - start`.

The algorithm `next_permutation`, which has the syntax

```
next_permutation(start, end)
```

creates the next lexicographically larger permutation of the elements in the range `[start, end]`; it returns the value `true` iff such a next permutation exists. By starting with the smallest lexicographic permutation of a sequence of distinct elements and making successive calls to `next_permutation`, we can obtain all permutations. Program 1.35 does just this. The invocation of `copy` in this program copies the elements `list[0:m]` to the output stream `cout`; each copied element is followed by the null string (""). Program 1.35 is equivalent to Program 1.32 provided the initial sequence is the smallest lexicographic sequence. Notice that Program 1.35 outputs no permutation that is lexically smaller than the initial sequence whereas Program 1.32 outputs all permutations regardless of the initial ordering. The exercises examine ways to modify Program 1.35 so as to obtain all permutations.

```
template<class T>
void permutations(T list[], int k, int m)
{// Generate all permutations of list[k:m].
// Assume k <= m.
// output the permutations one by one
do {
    copy(list, list+m+1,
          ostream_iterator<T>(cout, ""));
    cout << endl;
} while (next_permutation(list, list+m+1));
}
```

Program 1.35 Permutations using the STL algorithm `next_permutation`

A more general form of the `next_permutation` algorithm takes a third parameter `compare` as in

```
next_permutation(start, end, compare)
```

When this form is used, the binary function `compare` is used to determine whether one element is smaller than another. In the two-parameter version, this comparison is done using the operator `<`. ■

The STL contains many algorithms in addition to the ones used in the preceding examples. Exercises 2 through 7 are solved quite easily using the appropriate STL algorithms. The exercises for this section explore STL algorithms further.

EXERCISES

27. Write C++ code for the three-parameter template function `accumulate`. Test your code.
28. Write C++ code for the four-parameter template function `accumulate`. Test your code.
29. Write C++ code for the template function `copy`. Test your code.
30. Modify Program 1.35 so that it outputs all permutations of distinct elements. Do this by sorting the list elements into ascending order prior to generating the permutations. To sort, use the STL algorithm

```
sort(start, end)
```

which sorts elements in the range `[start, end]` into ascending order. Test your code.

31. Modify Program 1.35 so that it outputs all permutations of distinct elements. Do this by first using `next_permutation` to generate permutations that are lexically larger than the initial permutation and then using the STL algorithm `prev_permutation` to generate permutations that are lexically smaller than the initial permutation. Test your code.
32. Modify Program 1.35 so that it outputs all permutations of distinct elements. Do this by using the fact that when `next_permutation` returns the value `false`, the sequence `[start, end]` is the lexically smallest sequence. Hence, subsequent invocations of `next_permutation` will get you the remaining (if any) permutations you need. Test your code.
33. Do Exercise 2 using the STL algorithm `count`, which has the syntax

```
count(start, end, value)
```

34. Do Exercise 3 using the STL algorithm `fill`, which has the syntax

35. Do Exercise 4 using the STL algorithm `inner_product`, which has the syntax


```
inner_product(start1, end1, start2, initialValue)
```

36. Do Exercise 5 using the STL algorithm `iota`, which has the syntax

```
iota(start, end, value)
```

37. Do Exercise 6 using the STL algorithm `is_sorted`, which has the syntax

```
is_sorted(start, end)
```

38. Do Exercise 7 using the STL algorithm `mismatch`, which has the syntax

```
mismatch(start1, end1, start2)
```

39. Write C++ code for the STL template function `count` of Exercise 33. Test your code.

40. Write C++ code for the STL template function `fill` of Exercise 34. Test your code.

41. Write C++ code for the STL template function `inner_product` of Exercise 35. Test your code.

42. Write C++ code for the STL template function `iota` of Exercise 36. Test your code.

43. Write C++ code for the STL template function `is_sorted` of Exercise 37. Test your code.

44. Write C++ code for the STL template function `mismatch` of Exercise 38. Test your code.

1.9 TESTING AND DEBUGGING

1.9.1 What Is Testing?

As indicated in Section 1.1, correctness is the most important attribute of a program. Because providing a mathematically rigorous proof of correctness for even a small program is quite difficult, we resort to a process called **program testing** in which we execute the program on the target computer using input data, called **test data**, and compare the program's behavior with the expected behavior. If these two behaviors are different, we have a problem with the program. Unfortunately, however, even if the two behaviors are the same, we cannot conclude that the program is correct, as the two behaviors may not be the same on other input data. By using many sets of input data and verifying that the observed and expected behaviors are the same, we can increase our confidence in the correctness of the program. By using all possible input data, we can verify that the program is correct. However, for most practical programs, the number of possible input data is too large to perform such exhaustive testing. The subset of the input data space that is actually used for testing is called the **test set**.

Example 1.6 [Quadratic Roots] A **quadratic function** (or simply a **quadratic**) in x is a function that has the form

$$ax^2 + bx + c$$

where the values of a , b , and c are real numbers and $a \neq 0$. $3x^2 - 2x + 4$, $-9x^2 - 7x$, $3.5x^2 + 4$, and $5.8x^2 + 3.2x + 5$ are examples of quadratic functions. $5x + 3$ is not a quadratic function.

The **roots** of a quadratic function are the values of x at which the function value is 0. For example, the roots of $f(x) = x^2 - 5x + 6$ are 2 and 3, as $f(2) = f(3) = 0$. Every quadratic has exactly two roots, and these roots are given by the formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For the function $f(x) = x^2 - 5x + 6$, $a = 1$, $b = -5$, and $c = 6$. Substituting these into the above formula, we get

$$\frac{5 \pm \sqrt{25 - 4 * 1 * 6}}{2} = \frac{5 \pm 1}{2}$$

So the roots of $f(x)$ are $x = 3$ and $x = 2$.

When $d = b^2 - 4ac = 0$, the two roots are the same; when $d > 0$, the two roots are different and real numbers; and when $d < 0$, the two roots are different and complex numbers. In this last case each root has a *real* part and an *imaginary* part. The real part is $\text{real} = -b/(2a)$, and the imaginary part is $\text{imag} = \sqrt{-d}/(2a)$. The complex roots are $\text{real} + \text{imag} * i$ and $\text{real} - \text{imag} * i$ where $i = \sqrt{-1}$.

The function `outputRoots` (Program 1.36) computes and outputs the roots of a quadratic. We shall not attempt a formal correctness proof for this function. Rather, we wish to establish correctness by testing. The number of possible inputs is the number of different triples (a, b, c) with $a \neq 0$. Even if we restrict a , b , and c to 16-bit nonnegative integers, the number of possible input triples is too large for us to test the program on all inputs. With 16 bits per integer, there are 2^{16} different values for b and c and $2^{16} - 1$ for a (recall that a cannot be 0). The number of different triples is $2^{32}(2^{16} - 1)$. If our target computer can test at the rate of 1,000,000 triples per second, it would take almost 9 years to complete! A faster computer executing at the rate of 1,000,000,000 triples per second would take almost 3 days. So a practical test set can contain only a small subset of the entire space of input data.

If we run the program using the data set $(a, b, c) = (1, -5, 6)$, the roots 2 and 3 are output. The program behavior agrees with the expected behavior, and we conclude that the program is correct for this input. However, verifying agreement between observed and expected behavior on a proper subset of the possible inputs does not prove that the program works correctly on all inputs. ■

```
void outputRoots(const double& a, const double& b, const double& c)
{// Compute and output the roots of the quadratic.

    double d = b * b - 4 * a * c;
    if (d > 0) { // two real roots
        double sqrtD = sqrt(d);
        cout << "There are two real roots "
            << (-b + sqrtD) / (2 * a) << " and "
            << (-b - sqrtD) / (2 * a)
            << endl;
    }
    else if (d == 0)
        // both roots are the same
        cout << "There is only one distinct root "
            << -b / (2 * a)
            << endl;
    else // complex conjugate roots
        cout << "The roots are complex"
            << endl
            << "The real part is "
            << -b / (2 * a) << endl
            << "The imaginary part is "
            << sqrt(-d) / (2 * a) << endl;
}
```

Program 1.36 Compute and output the roots of the quadratic $ax^2 + bx + c$

Since the number of different inputs that can be provided to a program is generally very large, testing is often limited to a very small subset of the possible inputs. Testing with this subset cannot conclusively establish the correctness of the program. As a result, *the objective of testing is not to establish correctness, but to expose the presence of errors.* The test set must be chosen so as to expose any errors that may be present in the program. Different test sets can expose different errors in a program.

Example 1.7 The test data $(a, b, c) = (1, -5, 6)$ causes `outputRoots` to execute the code for the case when there are two real roots. If the roots 2 and 3 are output, we can have some confidence that the statements executed during this test are correct. Notice that an erroneous code could still give the correct results. For example, if we omitted the `a` from the expression for `d` and mistakenly typed

```
double d = b * b - 4 * c;
```

the value of d is the same for our test data because $a = 1$. Since the test data $(1, -5, 6)$ did not execute all statements of the code, we have less confidence in the correctness of the statements that are not executed.

The test set $\{(1, -5, 6), (1, 3, 2), (2, 5, 2)\}$ can expose errors only in the first seven lines of `outputRoots`, as each triple in this test set executes only these seven lines of code. However, the test set $\{(1, -5, 6), (1, -8, 16), (1, 2, 5)\}$ causes all statements of `outputRoots` to execute and so has a better chance of exposing the errors in the code. ■

1.9.2 Designing Test Data

When developing test data, we should keep in mind that the objective of testing is to expose the presence of errors. If data designed to expose errors fails to expose any errors, then we may have confidence in the correctness of the program. To tell whether or not a program malfunctions on given test data, we must be able to verify the correctness of the program behavior on the test data.

Example 1.8 For our quadratic roots example, the behavior on any test data may be verified in one of two ways. First, we might know the roots of the test quadratic. For example, the roots of the quadratic with $(a, b, c) = (1, -5, 6)$ are 2 and 3. We can verify the correctness of Program 1.36 on the test data $(1, -5, 6)$ by comparing the output roots with the correct roots 2 and 3. Another possibility is to substitute the roots produced by the program into the quadratic function and verify that the function value is 0. So if our program outputs 2 and 3 as the roots, we compute $f(2) = 2^2 - 5 \cdot 2 + 6 = 0$ and $f(3) = 3^2 - 5 \cdot 3 + 6 = 0$. We can implement these verification methods as computer programs. In the first case, the test program inputs the triple (a, b, c) as well as the expected roots and then checks the computed roots against the expected ones. For the second method we write code to evaluate the quadratic at the computed roots and verify that the result is 0. ■

We can evaluate any candidate test data using the following criteria:

- What is these data's potential to expose errors?
- Can we verify the correctness of the program behavior on this data?

Techniques for test data development fall into two categories: black box methods and white box methods. In a **black box method**, we consider the program's function, not the actual code, when we develop test data. In a **white box method**, we examine the code in an attempt to develop test data whose execution results in a good coverage of the program's statements and execution paths.

Black Box Methods

The most popular black box methods are I/O partitioning and cause-effect graphing. This section elaborates on the I/O partitioning method only. In this method we

partition the input and/or output data space into classes. The data in different classes cause the program to exhibit qualitatively different behaviors, while data in the same class cause qualitatively similar behaviors. The quadratic roots example has three different qualitative behaviors: the roots are complex, the roots are real and distinct, and the roots are real and the same. We can use these three behaviors to partition the input space into three classes. Data in the first class cause the first kind of behavior; data in the second cause the second kind of behavior; and data in the third cause the third kind of behavior. A test set should include at least one input from each class.

White Box Methods

White box methods create test data based on an examination of the code to be generated. The weakest condition we can place on a test set is that it results in each program statement being executed at least once. This condition is called **statement coverage**. For our quadratic roots example, the test set $\{(0,1,2), (1,-5,6), (1,-8,16), (1,2,5)\}$ causes all statements in Program 1.36 to execute. So this test set provides statement coverage. The test set $\{(0, 1, 2), (1,-5,6), (1,3,2), (2,5,2)\}$ does not provide statement coverage.

In **decision coverage** we require the test set to cause each conditional in the program to take on both true and false values. The code of Program 1.36 has three conditionals: $a == 0$, $d > 0$, and $d == 0$. In decision coverage we require at least one set of test data for which $a == 0.0$ is true and at least one for which it is false. We also require at least one test data for which $d > 0$ is true and at least one for which it is false; there should also be at least one set of test data for which $d == 0$ is true and at least one for which it is false.

Example 1.9 [Maximum Element] Program 1.37 returns the position of the largest element in the array $a[0:n]$. The program finds this position by scanning the array from positions 0 to n , using the variable `indexOfMax` to keep track of the position of the largest element seen so far. The data set $\{(a,-1), (a,4)\}$ with $a[0:4] = [2, 4, 6, 8, 9]$ provides statement coverage, but not decision coverage, as the condition $a[\text{indexOfMax}] < a[i]$ never becomes false. When $a[0:4] = [4, 2, 6, 8, 9]\}$, we get both decision and statement coverage. ■

We can strengthen the decision coverage criterion to require each clause of each conditional to take on both true and false values. This strengthened criterion is called **clause coverage**. A **clause** is formally defined to be a Boolean expression that contains no Boolean operator (i.e., `&&`, `||`, `!`). The expressions $x > y$, $x + y < y * z$, and c (where c is of type Boolean) are examples of clauses. Consider the statement

```
if ((C1 && C2) || (C3 && C4)) S1;
else S2;
```

```
template<class T>
int indexOfMax(T a[], int n)
{// Locate the largest element in a[0:n-1].
    if (n <= 0)
        throw illegalParameterValue("n must be > 0");

    int indexOfMax = 0;
    for (int i = 1; i < n; i++)
        if (a[indexOfMax] < a[i])
            indexOfMax = i;
    return indexOfMax;
}
```

Program 1.37 Finding the position of the largest element in a[0:n-1]

where C_1 , C_2 , C_3 , and C_4 are clauses and S_1 and S_2 are statements. Under the decision coverage criterion, we need to use one test set that causes $((C_1 \&& C_2) \mid\mid (C_3 \&& C_4))$ to be true and another that results in this conditional being false. Clause coverage requires us to use a test set that causes each of the four clauses C_1 through C_4 to evaluate to true at least once and to false at least once.

We can further strengthen clause coverage to require testing for all combinations of clause values. In the case of the conditional $((C_1 \&& C_2) \mid\mid (C_3 \&& C_4))$, this strengthening requires the use of 16 sets of test data: one for each truth combination of the four conditions. However, several of these combinations may not be possible.

If we sequence the statements of a program in their order of execution by a certain set of test data, we get an execution path. Different test data may yield different execution paths. Program 1.36 has only four execution paths—lines 1 through 7 (lines are numbered beginning with the line `double d = ...`); 1, 2, 8 through 12; and lines 1, 2, 8, 13 through 19. The number of execution paths of Program 1.37 grows as n increases. When $n < 0$, there is just one execution path—1, 2 (line 1 is the first `if` statement and line 3 is a blank line); when $n = 0$, there is again just one path—lines 1, 4, 5, 8; when $n = 1$, there are two paths—lines 1, 4, 5, 6, 5, 8 and 1, 4, 5, 6, 7, 5, 8; and when $n = 2$, there are four paths—1, 4, 5, 6, 5, 6, 5, 8; 1, 4, 5, 6, 7, 5, 6, 8; 1, 4, 5, 6, 7, 5, 8; and 1, 4, 5, 6, 7, 5, 6, 7, 5, 8. For a general n , $n \geq 0$, the number of execution paths is 2^n .

Execution path coverage requires the use of a test set that causes all execution paths to be executed. For the quadratic roots code, statement coverage, decision coverage, clause coverage, and execution path coverage are equivalent requirements. But for Program 1.37, statement coverage, decision coverage, and execution path coverage are different, and decision and clause coverage are equivalent.

Of the white box coverage criteria we have discussed, execution path coverage

is generally the most demanding. A test set that results in total execution path coverage also results in statement and decision coverage. It may, however, not result in clause coverage. Total execution path coverage often requires an infinite number of test data or at least a prohibitively large number of test data. Hence total path coverage is often impossible in practice.

Many exercises in this book ask you to test the correctness of your codes. The test data you use should at least provide statement coverage. Additionally, you should test for special cases that could cause your program to malfunction. For example, a program designed to sort $n \geq 0$ elements should be tested with $n = 0$ and 1 in addition to other values of n . If such a program uses an array $a[0:99]$, it should also be tested with $n = 100$. $n = 0, 1$, and 100 represent the **boundary conditions** empty, singleton, and full.

1.9.3 Debugging

Testing exposes the presence of errors in a program. Once a test run produces a result different from the one expected, we know that something is wrong with the program. The process of determining and correcting the cause of the discrepancy between the desired and observed behaviors is called **debugging**. Although a thorough study of debugging methods is beyond the scope of this book, we do provide some suggestions for debugging.

- Try to determine the cause of an error by logical reasoning. If this method fails, then you may wish to perform a program trace (using a debugger such as the one that comes with Microsoft Visual C++ .NET) to determine when the program started performing incorrectly. This approach becomes infeasible when the program executes many instructions with that test data and the program trace becomes too long to examine manually. In this case you must try to isolate the part of the code that is suspect and obtain a trace of this part.
- Do not attempt to correct errors by creating special cases. The number of special cases will soon become very large, and your code will look like a dish of spaghetti. Errors should be corrected by first determining their cause and then redesigning your solution as necessary.
- When correcting an error, be certain that your correction does not result in errors where there were none before. Run your corrected program on the test data on which it originally worked correctly to ensure that it still works correctly on these data.
- When testing and debugging a multimethod program, begin with a single method that is independent of the others. This method would typically be an input or output method. Then introduce additional methods one at a time, testing and debugging the larger program for correctness. This strategy is

called **incremental testing and debugging**. When this strategy is used, the cause of a detected error can reasonably be expected to lie in the most recently introduced method.

EXERCISES

45. Show that test sets that provide statement coverage for Program 1.36 also provide decision and execution path coverage.
46. Develop a test set for Program 1.37 that provides execution path coverage for the `for` loop when $n = 3$.
47. How many execution paths are in Program 1.30?
48. How many execution paths are in method `rSum` of Program 1.31?

1.10 REFERENCES AND SELECTED READINGS

A good introduction to programming in C++ can be found in the texts *C++ Program Design: An Introduction to Programming and Object-Oriented Design* by J. Cohoon and J. Davidson, 3rd Edition, McGraw Hill, NY, 2002 and *C++ How to Program*, 4th Edition, by H. Deitel and P. Deitel, Prentice Hall, Englewood Cliffs, NJ, 2002.

You can find a description of all components of the STL at the Web site <http://codeguru.earthweb.com/spp/stlguide>.

The Art of Software Testing by G. Myers, John Wiley, New York, NY, 1979 and *Software Testing Techniques* by B. Beizer, Second Edition, Van Nostrand Reinhold, New York, NY, 1990 have more thorough treatments of software testing and debugging techniques.

CHAPTER 2

PERFORMANCE ANALYSIS

BIRD'S-EYE VIEW

The most important attribute of a program is correctness. A program that does not correctly perform the task it was designed to do is of little use. However, correct programs may also be of little use. This is the case, for example, when a correct program takes more memory than is available on the computer it is to run on as well as when a correct program takes more time than the user is willing to wait. We use the term *program performance* to refer to the memory and time requirements of a program. To appreciate the need for good data structures and algorithm design methods, you must be able to evaluate the performance of a program.

This chapter focuses on paper-and-pencil methods to determine the memory and time requirements of a program. The operation count and step-count approaches to estimate run time are developed, and the notions of best-case, worst-case, and average run time are introduced. A more advanced measure of run time—amortized complexity—is developed in the Web site for this book. You should not attempt to read the material on amortized complexity until you have completed Chapter 9.

Chapter 3 reviews asymptotic notations such as big oh, omega, theta, and little oh, which make up the lingua franca for performance analysis. The use of asymptotic notation often simplifies the analysis. Chapter 4 shows you how to measure the actual run time of a program by using a clocking method.

Many application codes are developed in this chapter. These applications, which will prove useful in later chapters, include

- Searching an array of elements for an element with a specified characteristic.
- Sorting an array of elements. Codes for the rank (or count) sort, selection sort, bubble sort, and insertion sort methods are developed.
- Evaluating a polynomial using Horner's rule.
- Performing matrix operations such as add, transpose, and multiply.

2.1 WHAT IS PERFORMANCE?

By the **performance of a program**, we mean the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In **performance analysis** we use analytical methods, while in **performance measurement** we conduct experiments.

The **space complexity** of a program is the amount of memory it needs to run to completion. We are interested in the space complexity of a program for the following reasons:

- If the program is to be run on a multiuser computer system, then we may need to specify the amount of memory to be allocated to the program.
- For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.
- A problem might have several possible solutions with different space requirements. For instance, one C++ compiler for your computer might need only 1 MB of memory, while another might need 4 MB. The 1 MB compiler is the only choice if your computer has less than 4 MB of memory. Even users whose computers have the extra memory will prefer the smaller compiler if its capabilities are comparable to those of the bigger compiler. The smaller compiler leaves the user with more memory for other tasks.
- We can use the space complexity to estimate the size of the largest problem that a program can solve. For example, we may have a circuit simulation program that requires $10^6 + 100(c + w)$ bytes of memory to simulate circuits with c components and w wires. If the total amount of memory available is 5.01×10^8 bytes, then we can simulate circuits with $c + w \leq 5,000,000$.

The **time complexity** of a program is the amount of computer time it needs to run to completion. We are interested in the time complexity of a program for the following reasons:

- Some computer systems require the user to provide an upper limit on the amount of time the program will run. Once this upper limit is reached, the program is aborted. An easy way out is to simply specify a time limit of a few thousand years. However, this solution could result in serious fiscal problems if the program runs into an infinite loop caused by some discrepancy in the data and you actually get billed for the computer time used. We would like to provide a time limit that is just slightly above the expected run time.
- The program we are developing might need to provide a satisfactory real-time response. For example, all interactive programs must provide such a response. A text editor that takes a minute to move the cursor one page down or one

page up will not be acceptable to many users. A spreadsheet program that takes several minutes to reevaluate the cells in a sheet will be satisfactory only to very patient users. A database management system that allows its users adequate time to drink two cups of coffee while it is sorting a relation will not find too much acceptance. Programs designed for interactive use must provide satisfactory real-time response. From the time complexity of the program or program module, we can decide whether or not the response time will be acceptable. If not, we need to either redesign the algorithm or give the user a faster computer.

If we have alternative ways to solve a problem, then the decision on which to use will be based primarily on the expected performance difference among these solutions. We will use some weighted measure of the space and time complexities of the alternative solutions.

EXERCISES

1. Give two more reasons why analysts are interested in the space complexity of a program.
2. Give two more reasons why analysts are interested in the time complexity of a program.

2.2 SPACE COMPLEXITY

2.2.1 Components of Space Complexity

The space needed by a program has the following components:

- *Instruction space*

Instruction space is the space needed to store the compiled version of the program instructions.

- *Data space*

Data space is the space needed to store all constant and variable values. Data space has two components:

1. Space needed by constants (for example, the numbers 0 and 1 in Programs 1.29 and 1.30) and simple variables (such as *a*, *b*, and *c* in Program 1.1).
2. Space needed by dynamically allocated objects such as arrays and class instances.

- *Environment stack space.*

The environment stack is used to save information needed to resume execution of partially completed functions and methods. For example, if function `foo` invokes function `goo`, then we must at least save a pointer to the instruction of `foo` to be executed when `goo` terminates.

Instruction Space

The amount of instruction space that is needed depends on factors such as

- The compiler used to compile the program into machine code.
- The compiler options in effect at the time of compilation.
- The target computer.

The compiler is a very important factor in determining how much space the resulting code needs. Figure 2.1 shows three possible codes for the evaluation of $a+b+b*c+(a+b-c)/(a+b)+4$. These codes need a different amount of space, and the compiler in use determines exactly which code will be generated.

Even with the same compiler, the size of the generated program code can vary. For example, a compiler might provide the user with optimization options. These could include code-size optimization as well as execution-time optimization. In Figure 2.1, for instance, the compiler might generate the code of Figure 2.1(b) in nonoptimization mode. In optimization mode, the compiler might use the knowledge that $a+b+b*c = b*c+(a+b)$ and generate the shorter and more time-efficient code of Figure 2.1(c). The use of the optimization mode will generally increase the time needed to compile the program.

The example of Figure 2.1 brings to light an additional contribution to the space requirements of a program. Space is needed for temporary variables such as `t1`, `t2`, ..., `t6`.

Another option that can have a significant effect on program space is the overlay option in which space is assigned only to the program module that is currently executing. When a new module is invoked, it is read in from a disk or other device, and the code for the new module overwrites the code of the old module. So program space corresponding to the size of the largest module (rather than the sum of the module sizes) is needed.

The configuration of the target computer also can affect the size of the compiled code. If the computer has floating-point hardware, then floating-point operations will translate into one machine instruction per operation. If this hardware is not installed, then code to simulate floating-point computations will be generated.

Data Space

The C++ language does not specify the space to be allocated for the various C++ data types. Figure 2.2 gives the space allocation used by most C++ compilers. The

```
LOAD a  
ADD b  
STORE t1  
LOAD b  
MULT c  
STORE t2  
LOAD t1  
ADD t2  
STORE t3  
LOAD a  
ADD b  
SUB c  
STORE t4  
LOAD a  
ADD b  
STORE t5  
LOAD t4  
DIV t5  
STORE t6  
LOAD t3  
ADD t6  
ADD 4
```

(a)

```
LOAD a  
ADD b  
STORE t1  
SUB c  
DIV t1  
STORE t2  
LOAD b  
MUL c  
STORE t3  
LOAD t1  
ADD t3  
ADD t2  
ADD 4
```

(b)

```
LOAD a  
ADD b  
STORE t1  
SUB c  
DIV t1  
STORE t2  
LOAD b  
MUL c  
ADD t2  
ADD t1  
ADD 4
```

(c)

Figure 2.1 Three equivalent codes

data type `int` is typically assigned as many bytes (1 byte = 8 bits) as there are in a word. So, on a 4-byte per word computer, an `int` is 4 bytes long while on a 2-byte per word computer, an `int` is typically 2 bytes. We shall use the data of Figure 2.2 when computing the space required by variables and constants.

We can obtain the space requirement for a structured variable by adding up the space requirements of all its components. Similarly, we can obtain the space requirement of an array by multiplying the array size and the space needs of a single array element.

Consider the following array declarations:

```
double a[100];  
int maze[rows][cols];
```

When computing the space allocated to an array, we shall be concerned only with the space allocated for the array elements. The array `a` has space for 100 elements of type `double`, each taking 8 bytes. The total space allocated to the array is therefore

Type	Space (bytes)	Range
bool	1	{true, false}
char	1	[-128, -127]
unsigned char	1	[0, 255]
short	2	[-32768, 32767]
unsigned short	2	[0, 65535]
long	4	$[-2^{31}, 2^{31} - 1]$
unsigned long	4	$[0, 2^{32} - 1]$
int	4	$[-2^{31}, 2^{31} - 1]$
unsigned int	4	$[0, 2^{32} - 1]$
float	4	$\pm 3.4E \pm 38$ (7 digits)
double	8	$\pm 1.7E \pm 308$ (15 digits)
long double	10	$\pm 1.2E \pm 4932$ (19 digits)
pointer	2	(near, .cs, .ds, .es, .ss pointers)
pointer	4	(far, huge pointers)

Figure 2.2 Space typically allocated to C++ data types on a 32-bit/word computer

800 bytes. The array `maze` has space for `rows*cols` elements of type `int`. The total space taken by this array is `4*rows*cols` bytes.

Environment Stack

Beginning performance analysts often ignore the space needed by the environment stack because they don't understand how functions (and in particular recursive ones) are invoked and what happens on termination. Each time a function is invoked the following data are saved on the environment stack:

- The return address.
- The values of all local variables and formal parameters in the function being invoked (necessary for recursive functions only).

Each time the recursive function `rSum` (Program 1.31) is invoked, whether from outside the function or from within, the current values of `a` and `n` and the program location to return to on completion are saved in the environment stack.

It is worth noting that some compilers may save the values of the local variables and formal parameters for both recursive and nonrecursive methods, while others may do so for recursive methods alone. So the compiler in use will affect the amount of space needed by the environment stack.

Summary

The space needed by a program depends on several factors. Some of these factors are not known at the time the program is conceived or written (e.g., the computer or the compiler that will be used). Until these factors have been determined, we cannot make an accurate analysis of the space requirements of a program.

We can, however, determine the contribution of those components that depend on characteristics of the problem instance to be solved. These characteristics typically include factors that determine the size of the problem instance (e.g., the number of inputs and outputs or magnitude of the numbers involved) being solved. For example, if we have a program that sorts n elements, we can determine space requirements as a function of n . For a program that adds two $n \times n$ matrices, we may use n as the instance characteristic, and for one that adds two $m \times n$ matrices, we may use m and n as the instance characteristics.

The size of the instruction space is relatively insensitive to the particular problem instance being solved. The contribution of the constants and simple variables to the data space is also independent of the characteristics of the problem instance to be solved except when the magnitude of the numbers involved becomes too large for the chosen data type. At this time we will need to either change the data type or rewrite the program using multiprecision arithmetic and then analyze the new program.

The space needed by some of the dynamically allocated memory may also be independent of the problem size. The environment stack space is generally independent of the instance characteristics unless recursive functions are in use. When recursive functions are in use, the instance characteristics will generally (but not always) affect the amount of space needed for the environment stack.

The amount of stack space needed by recursive functions is called the **recursion stack space**. For each recursive function this space depends on the space needed by the local variables and the formal parameters, the maximum depth of recursion (i.e., the maximum number of nested recursive calls), and the compiler being used. For Program 1.31 recursive calls get nested until n equals 0. At this time the nesting resembles Figure 2.3. The maximum depth of recursion for this program is therefore $n+1$. A smart compiler would replace a recursive call that is the last statement of a method (known as tail recursion) by equivalent iterative code. This technique could reduce, even eliminate, the recursion stack space.

We can divide the total space needed by a program into two parts:

- A fixed part that is independent of the instance characteristics. This part typically includes the instruction space (i.e., space for the code), space for simple variables, space for constants, and so on.
- A variable part that consists of the dynamically allocated space (to the extent that this space depends on the instance characteristics); and the recursion stack space (insofar as this space depends on the instance characteristics).

```

rSum(a,n)
  rSum(a,n-1)
    rSum(a,n-2)

.
.
.

  rSum(a,1)
    rSum(a,0)

```

Figure 2.3 Nesting of recursive calls for Program 1.31

The space requirement of any program P may therefore be written as

$$c + S_P(\text{instance characteristics})$$

where c is a constant that denotes the fixed part of the space requirements and S_P denotes the variable component. An accurate analysis should also include the space needed by temporary variables generated during compilation (refer to Figure 2.1). This space is compiler dependent and, except in the case of recursive functions, independent of the instance characteristics. We will ignore the space needs of these compiler-generated variables.

When analyzing the space complexity of a program, we will concentrate solely on estimating S_P (instance characteristics). For any given problem we need to first determine which instance characteristics to use to measure the space requirements. The choice of instance characteristics is very problem specific, and we will resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the program. At times we also use more complex measures of the interrelationships among the data items.

2.2.2 Examples

Example 2.1 Consider Program 1.1. Before we can determine S_P , we must select the instance characteristics to be used for the analysis. Suppose we use the magnitudes of a , b , and c as the instance characteristic. Since a , b , and c are of type `int`, 4 bytes are allocated to each of the formal parameters. In addition, space is needed for the code. Neither the data space nor the instruction space is affected by the magnitudes of a , b , and c . Therefore, $S_{abc}(\text{instance characteristics}) = 0$. ■

Example 2.2 [Sequential Search] Program 2.1 examines the elements of the array a from left to right to see whether one of these elements equals x . If an element

```
template<class T>
int sequentialSearch(T a[], int n, const T& x)
{// Search the unordered list a[0:n-1] for x.
// Return position if found; return -1 otherwise.
    int i;
    for (i = 0; i < n && a[i] != x; i++);
    if (i == n) return -1;
    else return i;
}
```

Program 2.1 Sequential search

equal to x is found, the function returns the position of the first occurrence of x . If the array has no element equal to x , the function of Program 2.1 returns -1 .

We wish to obtain the space complexity of Program 2.1 in terms of the instance characteristic n . Although we need space for the formal parameters a , x and n , the constants 0 and -1 , and the code, the space needed is independent of n . Therefore, $S_{\text{sequentialSearch}}(n) = 0$.

Note that the array a must be large enough to hold the n elements being searched. The space needed by this array ($n * s$ bytes, where s is the number of bytes needed by an object of type T). This space is, however, allocated in the function where the actual parameter corresponding to a is declared. As a result, we do not add the space requirements of this array into the space requirements of the function `sequentialSearch`. ■

Example 2.3 For method `sum` (Program 1.30), suppose we are interested in measuring space requirements as a function of the number of elements to be summed. Space is required for the formal parameters a and n , the local variables i and `theSum`, the constant 0 , and the instructions. The amount of space needed does not depend on the value of n , so $S_{\text{sum}}(n) = 0$. ■

Example 2.4 Consider the function `rSum` (Program 1.31). As in the case of `sum`, assume that the instances are characterized by n . The recursion stack space includes space for the formal parameters a and n and the return address. In the case of a , a reference (4 bytes) is saved, while in the case of n , a value of type `int` (also 4 bytes) is saved on the recursion stack. If we assume that the return address also takes 4 bytes, we determine that each call to `rSum` requires 12 bytes of recursion stack space. Since the depth of recursion is $n+1$, the recursion stack space needed is $12(n+1)$ bytes. So $S_{\text{rsum}}(n) = 12(n+1)$.

Program 1.30 has a smaller space requirement than does Program 1.31. ■

Example 2.5 [Factorial] The space complexity of Program 1.29, which computes the factorial function, is analyzed as a function of n rather than as a function of the number of inputs (one) or outputs (one). The recursion depth is $\max\{n, 1\}$. The recursion stack saves a return address (4 bytes) and the value of n (4 bytes) each time `factorial` is invoked. No additional space that is dependent on n is used, so $S_{\text{factorial}}(n) = 8 * \max\{n, 1\}$. ■

Example 2.6 [Permutations] Program 1.32 outputs all permutations of a list of elements. With the initial invocation `permutations(list, 0, n-1)`, the depth of recursion is n . Since each recursive call requires 20 bytes of recursion stack space (4 for each of return address, `list`, `k`, `m`, and `i`), the recursion stack space needed is $20n$ bytes, so $S_{\text{permutations}}(n) = 20n$. ■

EXERCISES

3. Compile a sample C++ program using two C++ compilers. Is the code length the same or different?
4. List additional factors that may influence the space complexity of a program.
5. Using the data provided in Figure 2.2, determine the number of bytes needed by the following arrays:
 - (a) `double y[3]`
 - (b) `int matrix[10][100]`
 - (c) `double x[100][5][20]`
 - (d) `float z[10][10][10][5]`
 - (e) `bool a[2][3][4]`
 - (f) `long b[3][3][3][3]`
6. Program 2.2 gives a recursive function `rSequentialSearch` that searches the elements of the array `a[0:n-1]` for the element `x`. If `x` is found, the function returns the position of `x` in `a`. Otherwise, the function returns `-1`. Determine $S_P(n)$. Determine $S_{\text{rSequentialSearch}}(n)$.
7. Write a nonrecursive function to compute $n!$ (see Example 1.1). Compare the space requirements of your nonrecursive function and those of the recursive version (Program 1.29).

```
template<class T>
int rSequentialSearch(T a[], int n, const T& x)
{// Search the unordered list a[0:n-1] for x.
// Return position if found; return -1 otherwise.
    if (n < 1) return -1;
    if (a[n-1] == x) return n - 1;
    return rSequentialSearch(a, n-1, x);
}
```

Program 2.2 Recursive sequential search

2.3 TIME COMPLEXITY

2.3.1 Components of Time Complexity

The time complexity of a program depends on all the factors that the space complexity depends on. A program will run faster on a computer capable of executing 10^9 instructions per second than on one that can execute only 10^7 instructions per second. The code of Figure 2.1(c) will require less execution time than the code of Figure 2.1(a). Some compilers will take less time than others to generate the corresponding computer code. Smaller problem instances will generally take less time than larger instances.

The time taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we can assume that a compiled program will be run several times without recompilation. Consequently, we will concern ourselves with just the run time of a program. This run time is denoted by $t_P(\text{instance characteristics})$.

Because many of the factors t_P depends on are not known when a program is conceived, it is reasonable to only estimate t_P . If we knew the characteristics of the compiler to be used, we could determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on that the code for P would make. Then we could obtain a formula for t_P . Letting n denote the instance characteristics, we might have an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \dots \quad (2.1)$$

where c_a , c_s , c_m , and c_d respectively denote the time needed for an addition, subtraction, multiplication, and division, and ADD , SUB , MUL , and DIV are functions whose value is the number of additions, subtractions, multiplications, and divisions that will be performed when the code for P is used on an instance with characteristic n .

Since the time needed for an arithmetic operation depends on the type (`int`, `float`, `double`, etc.) of the numbers in the operation, an exact formula for run time must separate the operation counts by data type. Fine-tuning Equation 2.1 in this way still does not give us an accurate formula to predict run time because today's computers do not necessarily perform arithmetic operations in sequence. For example, computers can perform an integer operation and a floating-point operation at the same time. Further, the capability to pipeline arithmetic operations and the fact that modern computers have a memory hierarchy (Section 4.5) means that the time to perform m additions isn't necessarily m times the time to perform one.

Since the analytical approach to determine the run time of a program is fraught with complications, we attempt only to estimate run time. Two more manageable approaches to estimating run time are (1) identify one or more key operations and determine the number of times these are performed and (2) determine the total number of steps executed by the program.

2.3.2 Operation Counts

One way to estimate the time complexity of a program or method is to select one or more operations, such as add, multiply, and compare, and to determine how many of each is done. The success of this method depends on our ability to identify the operations that contribute most to the time complexity. Several examples of this method follow.

Example 2.7 [Max Element] Program 1.37 returns the position of the largest element in the array `a[0:n-1]`. The time complexity of Program 1.37 can be estimated by determining the number of comparisons made between elements of the array `a`. When $n \leq 0$, an exception is thrown and the number of comparisons is 0. When $n = 1$, the `for` loop is not entered. So no comparisons between elements of `a` are made. When $n > 1$, each iteration of the `for` loop makes one comparison between two elements of `a`, and the total number of element comparisons is $n-1$. Therefore, the number of element comparisons is $\max\{n-1, 0\}$. The function `indexOfMax` performs other comparisons (for example, each iteration of the `for` loop is preceded by a comparison between `i` and `n`) that are not included in the estimate. Other operations such as initializing `indexOfMax` and incrementing the `for` loop index `i` are also not included in the estimate. If we included these other operations into our count, the count would increase by a constant factor. ■

Example 2.8 [Polynomial Evaluation] Consider the polynomial $P(x) = \sum_{i=0}^n c_i x^n$. If $c_n \neq 0$, $P(x)$ is a polynomial of degree n . Program 2.3 gives one way to compute $P(x)$ for a given value of x . Its time complexity can be estimated by determining the number of additions and multiplications performed inside the `for` loop. We will use the degree n as the instance characteristic. The `for` loop is entered a total of n times, and each time we enter the `for` loop one addition and two multiplications are done. (This operation count excludes the add performed each time the loop variable

i is incremented.) The number of additions is n , and the number of multiplications is $2n$.

```
template<class T>
T polyEval(T coeff[], int n, const T& x)
{// Evaluate the degree n polynomial with
// coefficients coeff[0:n] at the point x.
    T y = 1, value = coeff[0];
    for (int i = 1; i <= n; i++)
        {// add in next term
            y *= x;
            value += y * coeff[i];
        }
    return value;
}
```

Program 2.3 Evaluating a polynomial

Horner's rule evaluates $P(x)$ as shown below.

$$P(x) = (\cdots (c_n * x + c_{n-1}) * x + c_{n-2}) * x + c_{n-3} * x \cdots) * x + c_0$$

So $P(x) = 5 * x^3 - 4 * x^2 + x + 7$ is computed as $((5 * x - 4) * x + 1) * x + 7$. The corresponding C++ function is given in Program 2.4. Using the same measure as used for Program 2.3, we estimate the complexity of Program 2.4 as n additions and n multiplications. Since Program 2.3 performs the same number of additions but twice as many multiplications as does Program 2.4, we expect Program 2.4 to be faster. ■

Example 2.9 [Ranking] The **rank** of an element in a sequence is the number of smaller elements in the sequence plus the number of equal elements that appear to its left. For example if the sequence is given as the array $a = [4, 3, 9, 3, 7]$, then the ranks are $r = [2, 0, 4, 1, 3]$. Function **rank** (Program 2.5) computes the ranks of the elements in the array a . We can estimate the complexity of **rank** by counting the number of comparisons between elements of a . These comparisons are done in the **if** statement. For each value of i , the number of element comparisons is i . So the total number of element comparisons is $1 + 2 + 3 + \cdots + n - 1 = (n - 1)n/2$ (see Equation 1.3).

Note that our complexity estimate excludes the overhead associated with the **for** loops, the cost of initializing the array r , and the cost of incrementing r each time two elements of a are compared. ■

```

template<class T>
T horner(T coeff[], int n, const T& x)
{// Evaluate the degree n polynomial with
// coefficients coeff[0:n] at the point x.
    T value = coeff[n];
    for (int i = 1; i <= n; i++)
        value = value * x + coeff[n - i];
    return value;
}

```

Program 2.4 Horner's rule for polynomial evaluation

```

template<class T>
void rank(T a[], int n, int r[])
{// Rank the n elements a[0:n-1].
// Element ranks returned in r[0:n-1]
    for (int i = 0; i < n; i++)
        r[i] = 0; // initialize

    // compare all element pairs
    for (int i = 1; i < n; i++)
        for (int j = 0; j < i; j++)
            if (a[j] <= a[i]) r[i]++;
            else r[j]++;
}

```

Program 2.5 Computing ranks

Example 2.10 [Rank Sort] Once the elements have been ranked using Program 2.5, they may be rearranged in increasing order so that $a[0] \leq a[1] \leq \dots \leq a[n-1]$ by moving elements to positions corresponding to their ranks. If we have space for an additional array u , we can use the function `rearrange` given in Program 2.6.

Assume that the invocation of `new` succeeds in allocating space to the array u . The number of element moves performed during the execution of function `rearrange` is $2n$. The complete sort requires $(n - 1)n/2$ comparisons and $2n$ element moves. This method of sorting is known as `rank` or `count` sort. An alternative method to rearrange the elements is considered later (Program 2.11). This alternative method does not use an additional array such as u . ■

```
template<class T>
void rearrange(T a[], int n, int r[])
{// Rearrange the elements of a into sorted order
// using an additional array u.
T *u = new T [n]; // create additional array

// move to correct place in u
for (int i = 0; i < n; i++)
    u[r[i]] = a[i];

// move back to a
for (i = 0; i < n; i++)
    a[i] = u[i];

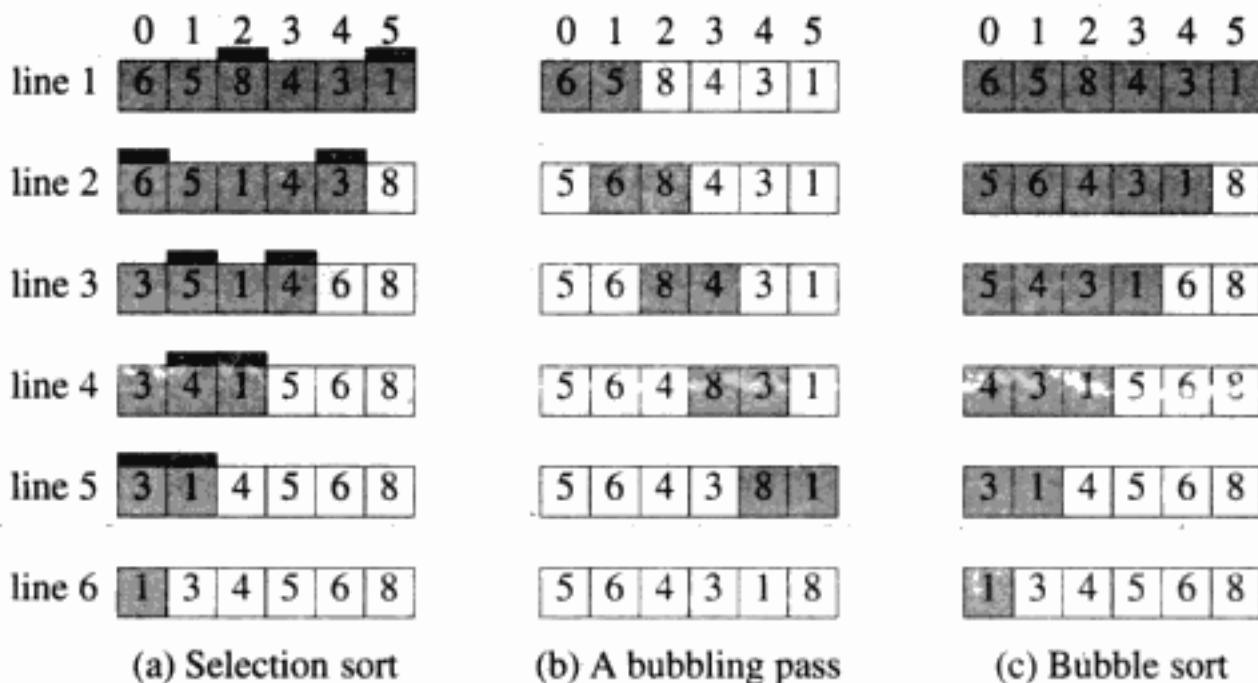
delete [] u;
}
```

Program 2.6 Rearranging elements using an additional array

Example 2.11 [Selection Sort] Example 2.10 examined one way to rearrange the elements in an array a so that $a[0] \leq a[1] \leq \dots \leq a[n-1]$. An alternative strategy is to determine the largest element and move it to $a[n-1]$, then determine the largest of the remaining $n - 1$ elements and move it to $a[n - 2]$, and so on. Figure 2.4(a) shows an example in which selection sort is used to sort the six-element array $a[0:5] = [6, 5, 8, 4, 3, 1]$. Shaded array positions designate the, as yet, unsorted part of the array. A heavy bar over an array position marks the maximum element, and a light bar marks the position into which the maximum element is to move.

Line 1 of the figure shows the initial configuration; the entire array is considered unsorted, the maximum element is in $a[2]$, and this maximum element is to be moved to $a[5]$. The move is accomplished by swapping the elements at the positions designated by the bars. Following the swap, we need concern ourselves only with sorting the elements $a[0:4]$ because $a[5]$ is known to contain the maximum element. Line 2 shows the configuration after the swap; the maximum element of $a[0:4]$ is $a[0]$, and this element is to be swapped with $a[4]$. Line 3 shows the result. Line 6 shows the result following three more stages of find the max and swap. At this time the unsorted part of the array ($a[0:0]$) has a single element that is known to be less than or equal to the other elements in the array. So the entire array is sorted.

Program 2.7 gives the C++ function, `selectionSort`, which implements the above strategy. Program 1.37 gave the function `indexOfMax`. We can estimate the complexity of `selectionSort` by counting the number of element comparisons

**Figure 2.4** Selection and bubble sort

made. From Example 2.7 we know that each invocation `indexOfMax(a, size)`, $\text{size} \geq 1$, results in $\text{size}-1$ comparisons being made. So the total number of comparisons is $n-1 + n-2 + \dots + 1 = (n-1)n/2$. The number of element moves is $3(n-1)$. Selection sort uses the same number of comparisons rank sort uses (Example 2.10) but requires 50 percent more element moves. We consider another version of selection sort in Example 2.16. ■

```
template<class T>
void selectionSort(T a[], int n)
{// Sort the n elements a[0:n-1].
    for (int size = n; size > 1; size--)
    {
        int j = indexOfMax(a, size);
        swap(a[j], a[size - 1]);
    }
}
```

Program 2.7 Selection sort

Example 2.12 [Bubble Sort] Bubble sort is another simple way to sort elements. This sort employs a “bubbling strategy” to get the largest element to the right. In a

bubbling pass, pairs of adjacent elements are compared. The elements are swapped in case the one on the left is greater than the one on the right. Suppose we have six integers in the order [6, 5, 8, 4, 3, 1] (see line 1 of Figure 2.4(b)). First the 6 and the 5 are compared and swapped to get the sequence shown in line 2. Next the 6 and 8 are compared, and no swap takes place. Then 8 and 4 are compared (line 3) and swapped; line 4 shows the result. The next comparison is between 8 and 3, and the two are swapped. The last comparison is between 8 and 1; these are swapped to get the configuration shown in line 6. The bubbling pass is now complete. At the end of the bubbling pass, we are assured that the largest element is in the right-most position.

The function `bubble` (Program 2.8) performs a bubbling pass over `a[0:n-1]`. The number of comparisons between pairs of elements of `a` is $n-1$.

```
template<class T>
void bubble(T a[], int n)
{// Bubble largest element in a[0:n-1] to right.
    for (int i = 0; i < n - 1; i++)
        if (a[i] > a[i+1]) swap(a[i], a[i + 1]);
}
```

Program 2.8 A bubbling pass

Since `bubble` causes the largest element to move to the right-most position, it can be used in place of `indexOfMax` in `selectionSort` (Program 2.7) to obtain a new sorting function (Program 2.9). The number of element comparisons is $(n - 1)n/2$ as it is for `selectionSort`. Figure 2.4(c) shows an initial array configuration as well as the array configuration after each bubbling pass. ■

```
template<class T>
void bubbleSort(T a[], int n)
{// Sort a[0:n - 1] using bubble sort.
    for (int i = n; i > 1; i--)
        bubble(a, i);
}
```

Program 2.9 Bubble sort

2.3.3 Best, Worst, and Average Operation Counts

In the examples so far, the operation counts were nice functions of fairly simple instance characteristics like the number of inputs and/or outputs. Some of our examples would have been more complicated if we had chosen to count some other operations. For example, the number of swaps performed by `bubble` (Program 2.8) depends not only on the instance characteristic n but also on the particular values of the elements in the array a . The number of swaps varies from a low of 0 to a high of $n - 1$. Since the operation count isn't always uniquely determined by the chosen instance characteristics, we ask for the best, worst, and average counts. The average operation count is often quite difficult to determine. As a result, in several of the following examples we limit our analysis to determining the best and worst counts.

Example 2.13 [Sequential Search] We are interested in determining the number of comparisons between x and the elements of a during an execution of the sequential search code of Program 2.1. A natural instance characteristic to use is n . Unfortunately, the number of comparisons isn't uniquely determined by n . For example, if $n = 100$ and $x = a[0]$, then only 1 comparison is made. However, if x isn't equal to any of the $a[i]$ s, then 100 comparisons are made.

A search is **successful** when x is one of the $a[i]$ s. All other searches are **unsuccessful**. Whenever we have an unsuccessful search, the number of comparisons is n . For successful searches the best comparison count is 1, and the worst is n . For the average count assume that all array elements are distinct and that each is searched for with equal frequency. The average count for a successful search is

$$\frac{1}{n} \sum_{i=1}^n i = (n + 1)/2$$

Example 2.14 [Insertion into a Sorted Array] You are to insert a new element into a sorted array so that the result is also a sorted array. For example, when you insert 3 into $a[0:4] = [2,4,6,8,9]$, the result is $a[0:5] = [2,3,4,6,8,9]$. The insertion may be done by beginning at the right end and successively moving array elements one position right until we find the location for the new element. Figure 2.5(a) illustrates the process. In our example we moved 9, 8, 6, and 4 one position right and inserted 3 into the now-vacant spot $a[1]$.

Program 2.10 implements the above strategy to insert an element x into a sorted array $a[0:n-1]$.

We wish to determine the number of comparisons made between x and the elements of a . The natural instance characteristic to use is the number n of elements initially in a . The best or minimum number of comparisons is 1, which happens when the new element x is to be inserted at the right end. The maximum number of comparisons is n , which happens when x is to be inserted at the left end. For

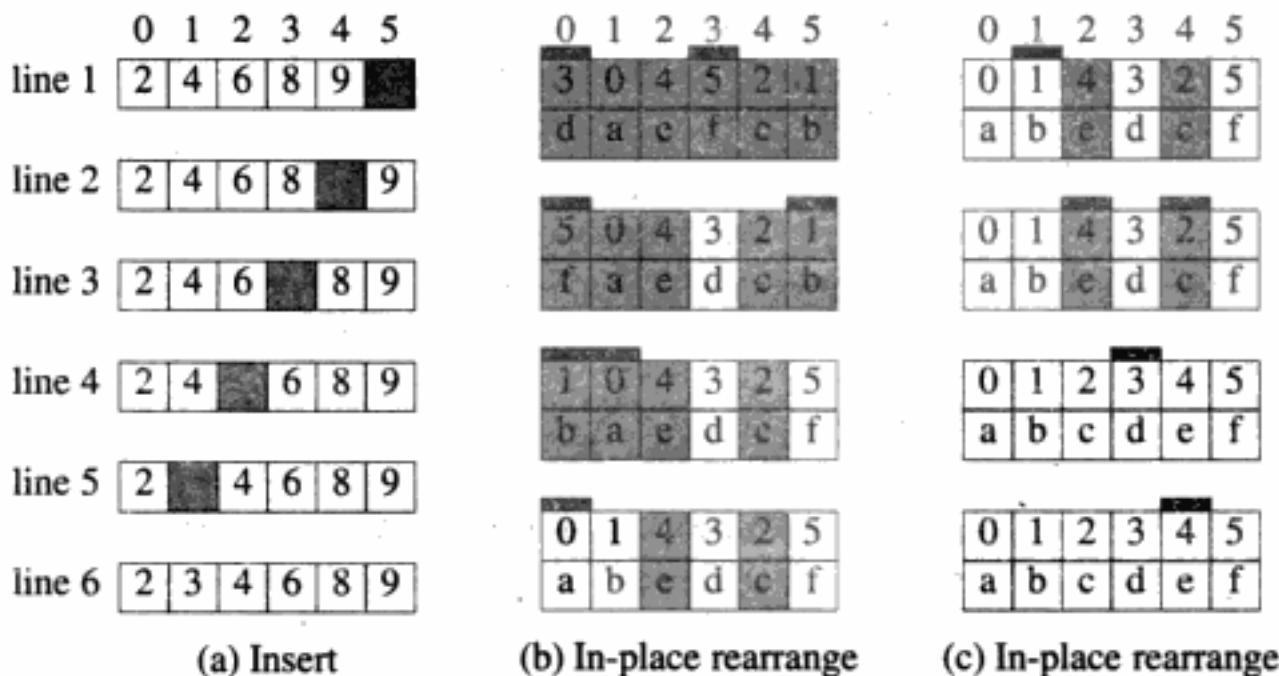


Figure 2.5 Insert and rearrange

```
template<class T>
void insert(T a[], int& n, const T& x)
{// Insert x into the sorted array a[0:n-1].
// Assume a is of size > n
    int i;
    for (i = n-1; i >= 0 && x < a[i]; i--)
        a[i+1] = a[i];
    a[i+1] = x;
    n++; // one element added to a
}
```

Program 2.10 Inserting into a sorted array

the average assume that x has an equal chance of being inserted into any of the possible $n+1$ positions. If x is eventually inserted into position $i+1$ of a , $i \geq 0$, then the number of comparisons is $n-i$. If x is inserted into $a[0]$, the number of comparisons is n . So the average count is

$$\frac{1}{n+1} \left(\sum_{i=0}^{n-1} (n-i) + n \right) = \frac{1}{n+1} \left(\sum_{j=1}^n j + n \right)$$

$$\begin{aligned}
 &= \frac{1}{n+1} \left(\frac{n(n+1)}{2} + n \right) \\
 &= \frac{n}{2} + \frac{n}{n+1}
 \end{aligned}$$

So the average count is almost 1 more than half the worst-case count. ■

Example 2.15 [Rank Sort Revisited] Suppose the elements of an array have been ranked using method `rank` (Program 2.5, Example 2.9). We can perform an in-place rearrangement of elements into sorted order by examining the array positions one at a time beginning with position 0. If we are currently examining position i and $r[i] = i$, then we may advance to the next position. If $r[i] \neq i$, then we swap the elements in positions i and $r[i]$. This swap moves the element previously in position i into its correct sorted position. The swap operation is repeated at position i until the element that belongs in position i in the sorted order is swapped into position i . Then we advance i to the next position.

Parts (b) and (c) of Figure 2.5 show how the above rearrangement strategy works. The initial array is $a[0:5] = [d,a,e,f,c,b]$. Element ranks are shown above the elements. So the initial rank array is $r[0:5] = [3,0,4,5,2,1]$. We begin at array position 0. Since $r[0] \neq 0$, $a[0]$ and $a[r[0]] = a[3]$ are to be swapped. The configurations of parts (b) and (c) of Figure 2.5 have a heavy bar above the position $a[i]$ being examined (initially, $i = 0$) and a light bar above the position $a[r[i]]$ where $a[i]$ is to move to. When $r[a[i]] = i$, the figure has only a heavy bar above $a[i]$. Shaded array positions denote elements that are not in their proper place (i.e., elements with $r[i] \neq i$).

We begin with $i = 0$ and swap elements $a[i]$ and $a[r[i]] = a[3]$; $r[0]$ and $r[3]$ are also swapped. This process results in the second configuration. Notice that $a[3]$ now contains the proper element and $r[3] = 3$. Next elements $a[0]$ and $a[r[0]] = a[5]$ together with their ranks are swapped to get the third configuration of Figure 2.5(b). When $a[0]$ and $a[r[0]] = a[1]$ (and their ranks) are swapped, we get the fourth configuration. Now $r[0] = 0$, and we increment i to the next position 1. The new configuration is shown at the top of Figure 2.5(c). Since $r[i] = r[1] = 1$, we advance i to the next position 2 (see the second configuration of Figure 2.5(c)). Now $a[i] = a[2]$ and $a[r[2]] = a[4]$ (as well as their ranks) are swapped. Following the swap, $r[2] = 2$. Even though the rearrangement is complete at this time, our code will not be able to detect this and we continue to advance i to the right, making sure that each element is in its proper position. So i is advanced to the next position 3 (see the third configuration of Figure 2.5(c)). Then i is advanced to positions 4 and 5.

Program 2.11 gives the in-place rearrangement function `rearrange`.

The number of swaps performed varies from a low of 0 (when the elements are initially in sorted order) to a high of $2(n - 1)$. Notice that each swap involving the $a[]$ s moves at least one element into its sorted position (i.e., element $a[i]$).

```
template<class T>
void rearrange(T a[], int n, int r[])
{// In-place rearrangement into sorted order.
    for (int i = 0; i < n; i++)
        // get proper element to a[i]
        while (r[i] != i)
        {
            int t = r[i];
            swap(a[i], a[t]);
            swap(r[i], r[t]);
        }
}
```

Program 2.11 In-place rearrangement of elements

So after at most $n - 1$ swaps, all n elements must be in sorted order. Exercise 20 establishes that this many element swaps may be needed on certain inputs. Hence the number of swaps is 0 in the best case and $2(n - 1)$ in the worst case (includes rank swaps). When this in-place rearrangement function is used in place of the one in Program 2.6, the worst-case execution time increases because we need more element moves (each swap requires three moves). However, the space requirements are reduced. ■

Example 2.16 [Selection Sort Revisited] A shortcoming of the selection sort code of Program 2.7 is that it continues to work even after the elements have been sorted. For example, the **for** loop iterates $n - 1$ times, even though the array may be sorted after the second iteration. To eliminate the unnecessary iterations, during the scan for the largest element we can check to see whether the array is already sorted. Program 2.12 gives the resulting selection sort function. Here we have incorporated the loop to find the largest element directly into **selectionSort**, rather than write it as a separate method.

Figure 2.6(a) shows the progress of Program 2.12 when started with $a[0:5] = [6, 5, 4, 3, 2, 1]$. In the first iteration of the outer **for** loop, **size** = 6 and the line **sorted = false** is executed when $i = 1, 2, 3, 4$, and 5. So following the swap of $a[0]$ and $a[5]$, which results in the configuration of line 2, the outer **for** loop is reentered with **size** = 5. This time the line **sorted = false** is executed when $i = 2, 3$, and 4. Therefore, following the swap of $a[1]$ and $a[4]$, the outer **for** loop is reentered; this time **size** = 4. Now the line **sorted = false** is executed when $i = 4$, and following the swap of $a[2]$ and $a[3]$, the outer **for** loop is reentered. This time we are working with the array configuration of line 4; the line **sorted = false** is not executed, and execution of the outer **for** loop terminates. On the initial data of line 1 of Figure 2.6(a), the early terminating version makes one less

```

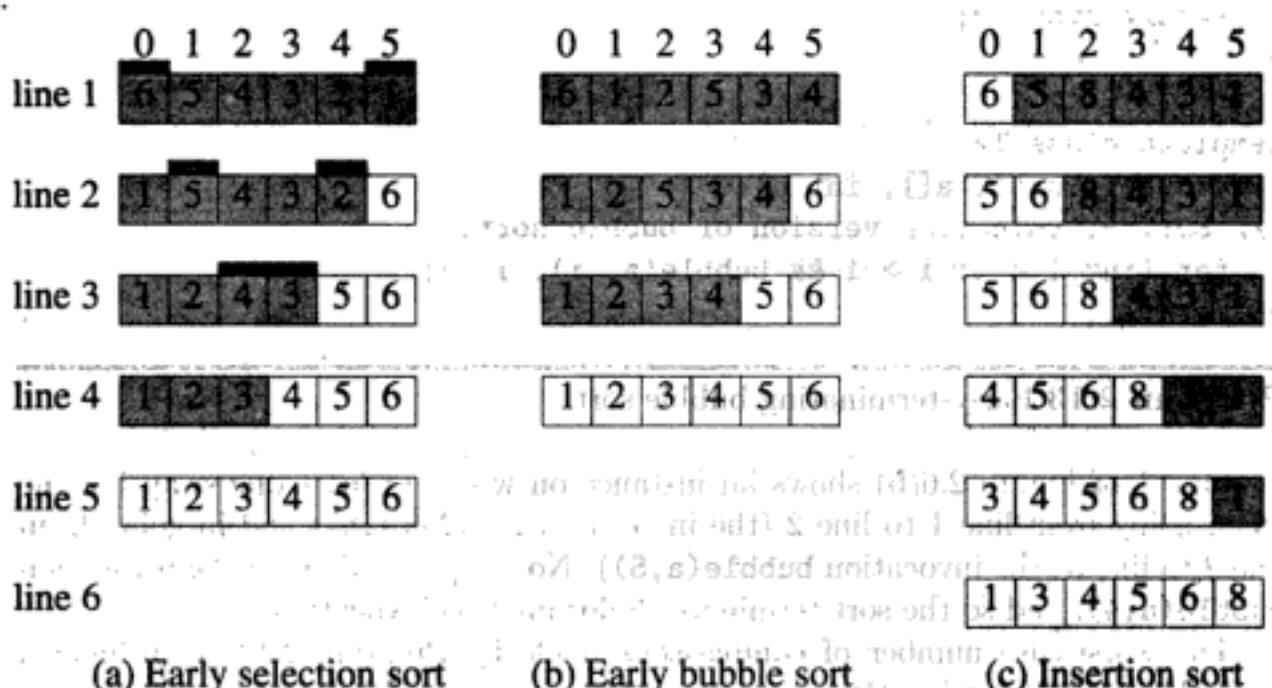
template<class T> void selectionSort(T a[], int n)
{
    // Early-terminating version of selection sort
    bool sorted = false;
    for (int size=n; !sorted && (size>0); size--)
    {
        int indexOfMax = 0;
        sorted = true;
        // find largest
        for (int i = 1; i < size; i++)
            if (a[indexOfMax] <= a[i]) indexOfMax = i;
            else sorted = false; // out of order
        swap(a[indexOfMax], a[size - 1]);
    }
}

```

Program 2.12 Early-terminating version of selection sort

pass than it does when started with the data shown in line 1 of **Figure 2.4(a)**.

early sort pass \\\ sorted = true

**Figure 2.6** Sorting examples

The best case for the early-terminating version of selection sort arises when the

array `a` is sorted to begin with. Now the outer `for` loop iterates just once, and the number of comparisons between elements of `a` is $n - 1$. In the worst case the outer `for` loop is iterated until `size = 1` and the number of comparisons is $(n - 1)n/2$. The best- and worst-case number of swaps remains the same as for Program 2.7. Notice that in the worst case we expect the early-terminating version to be slightly slower because of the additional work to maintain the variable `sorted`. ■

Example 2.17 [Bubble Sort Revisited] As in the case of selection sort, we can devise an early-terminating version of bubble sort. If a bubbling pass results in no swaps, then the array is in sorted order and no further bubbling passes are necessary. Program 2.13 gives the early-terminating version of bubble sort.

```
template<class T>
bool bubble(T a[], int n)
{// Bubble largest element in a[0:n-1] to right.
    bool swapped = false; // no swaps so far
    for (int i = 0; i < n - 1; i++)
        if (a[i] > a[i+1])
        {
            swap(a[i], a[i + 1]);
            swapped = true; // swap was done
        }
    return swapped;
}

template<class T>
void bubbleSort(T a[], int n)
{// Early-terminating version of bubble sort.
    for (int i = n; i > 1 && bubble(a, i); i--);
}
```

Program 2.13 Early-terminating bubble sort

Line 1 of Figure 2.6(b) shows an instance on which at least one swap is done when going from line 1 to line 2 (the invocation `bubble(a,6)`) and in going from line 2 to line 3 (the invocation `bubble(a,5)`). No swaps are done in the invocation `bubble(a,4)`, and so the sort terminates following this invocation.

The worst-case number of comparisons made by Program 2.13 is unchanged from the original version (Program 2.9). The best-case number of comparisons is $n - 1$. ■

Example 2.18 [Insertion Sort] Program 2.10 can be used as the basis of a method to sort n elements. Since an array with one element is a sorted array, we start with

an array that contains just the first of the n elements to be sorted. By inserting the second element into this one-element array, we get a sorted array of size 2. The insertion of the third element yields a sorted array of size 3. Continuing in this way, we obtain a sorted array of size n .

Line 1 of Figure 2.6(c) shows the unsorted array $a[0:5]$. We start with a sorted segment $a[0:0]$; the remaining elements $a[1:5]$ define the unsorted segment. The unsorted segment is the shaded segment in Figure 2.6(c). First $a[1]$ is inserted into the sorted segment $a[0:0]$ to get the configuration of line 2; $a[0:1]$ is now the sorted segment, and $a[2:5]$ is the unsorted segment. Next $a[2]$ is inserted into the sorted segment, and we get line 3 of the figure. $a[0:2]$ becomes the sorted segment, and $a[3:5]$ is the unsorted segment. After three more inserts, the entire array is sorted.

Function `insertionSort` (Program 2.14) implements this strategy. We have rewritten function `insert` for this application, as the original version (Program 2.10) performs some unnecessary operations. Actually, we could have embedded the code of the new `insert` function directly into the `sort` function to get the insertion sort version of Program 2.15. Equivalently, we could make `insert` an `inline` function.

```
template<class T>
void insert(T a[], int n, const T& x)
{// Insert x into the sorted array a[0:n-1].
    int i;
    for (i = n-1; i >= 0 && x < a[i]; i--)
        a[i+1] = a[i];
    a[i+1] = x;
}

template<class T>
void insertionSort(T a[], int n)
{// Sort a[0:n-1] using the insertion sort method.
    for (int i = 1; i < n; i++)
    {
        T t = a[i];
        insert(a, i, t);
    }
}
```

Program 2.14 Insertion sort

Both versions of insertion sort perform the same number of comparisons. In the best case the number of comparisons is $n - 1$, and in the worst case it is $(n - 1)n/2$.

■

```
template<class T>
void insertionSort(T a[], int n)
{// Sort a[0:n-1] using the insertion sort method.
    for (int i = 1; i < n; i++)
        {// insert a[i] into a[0:i-1]
            T t = a[i];
            int j;
            for (j = i-1; j >= 0 && t < a[j]; j--)
                a[j+1] = a[j];
            a[j+1] = t;
        }
}
```

Program 2.15 Another version of insertion sort

2.3.4 Step Counts

As noted in some of the examples on operation counts, the operation-count method of estimating time complexity omits accounting for the time spent on all but the chosen operations. In the **step-count** method, we attempt to account for the time spent in all parts of the program/method. As was the case for operation counts, the step count is a function of the instance characteristics. Although any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number of steps is computed as a function of some subset of these. Usually we choose the characteristics that are of interest to us. For example, we might wish to know how the computing (or run) time (i.e., time complexity) increases as the number of inputs increases. In this case the number of steps will be computed as a function of the number of inputs alone. For a different program we might want to determine how the computing time increases as the magnitude of one of the inputs increases. In this case the number of steps will be computed as a function of the magnitude of this input alone. Thus before the step count of a program can be determined, we need to know exactly which characteristics of the problem instance are to be used. These characteristics define not only the variables in the expression for the step count but also how much computing can be counted as a single step.

After the relevant instance characteristics have been selected, we can define a step. A **step** is any computation unit that is independent of the selected characteristics. Thus 10 additions can be one step; 100 multiplications can also be one step; but n additions, where n is an instance characteristic, cannot be one step. Nor can $m/2$ additions or $p+q$ subtractions, where m , p , and q are instance characteristics, be counted as one step.

Definition 2.1 A program step is loosely defined to be a syntactically or semantically meaningful segment of a program for which the execution time is independent of the instance characteristics. ■

The amount of computing represented by one program step may be different from that represented by another. For example, the entire statement

```
return a+b+b*c+(a+b-c)/(a+b)+4;
```

can be regarded as a single step if its execution time is independent of the instance characteristics we are using. We may also count a statement such as

```
x = y;
```

as a single step.

We can determine the number of steps that a program or method takes to complete its task by creating a global variable `stepCount` with initial value 0. Next we introduce statements into the program to increment `stepCount` by the appropriate amount. Therefore, each time a statement in the original program or method is executed, `stepCount` is incremented by the step count of that statement. The value of `stepCount` when the program or method terminates is the number of steps taken.

Example 2.19 When statements to increment `stepCount` are introduced into Program 1.30, the result is Program 2.16. The change in the value of `stepCount` by the time this program terminates is the number of steps executed by Program 1.30.

```
template<class T>
T sum(T a[], int n)
{ // Return sum of numbers a[0:n - 1].
    T theSum = 0;
    stepCount++; // for theSum = 0
    for (int i = 0; i < n; i++)
    {
        stepCount++; // for the for statement
        theSum += a[i];
        stepCount++; // for assignment
    }
    stepCount++; // for last execution of for statement
    stepCount++; // for return
    return theSum;
}
```

Program 2.16 Counting steps in Program 1.30

Program 2.17, which is a simplified version of Program 2.16, determines only the change in the value of `stepCount`. We see that for every initial value of `stepCount`, both Programs 2.16 and 2.17 compute the same final value for `stepCount`. In the `for` loop of Program 2.17, the value of `stepCount` increases by a total of $2n$. If `stepCount` is 0 to start with, then `stepCount` will be $2n+3$ on termination. Therefore, each invocation of `sum` (Program 1.30) executes a total of $2n+3$ steps. ■

```
template<class T>
T sum(T a[], int n)
{// Return sum of numbers a[0:n - 1].
    for (int i = 0; i < n; i++)
        stepCount += 2;
    stepCount += 3;
    return 0;
}
```

Program 2.17 Simplified version of Program 2.16

Example 2.20 When we introduce statements to increment `stepCount` into function `rSum` (Program 1.31), we obtain Program 2.18. Note that since Program 1.31 is a recursive function, it requires space for the recursion stack. Thus the function may fail to complete its task for lack of sufficient memory for the recursion stack. For the step-count analysis, we will assume that sufficient memory is available for the function `rSum` to successfully complete its task.

```
template<class T>
T rSum(T a[], int n)
{// Return sum of numbers a[0:n - 1].
    stepCount++; // for if conditional
    if (n > 0) {stepCount++; // for return and rSum invocation
                return rSum(a, n-1) + a[n-1];}
    stepCount++; // for return
    return 0;
}
```

Program 2.18 Counting steps in Program 1.31

Let $t_{rSum}(n)$ be the increase in the value of `stepCount` between the time `rSum` is initially invoked and the time it terminates. We see that $t_{rSum}(0) = 2$. When n

> 0 , `stepCount` increases by 2 plus whatever increase results from the invocation of `rSum` from within the `then` clause. From the definition of t_{rSum} , it follows that this additional increase is $t_{rSum}(n-1)$. So if the value of `stepCount` is 0 initially, its value at the time of termination is $2 + t_{rSum}(n-1)$, $n > 0$.

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count (such as $t_{rSum}(n) = 2 + t_{rSum}(n-1)$, $n > 0$ and $t_{rSum}(0) = 2$). This recursive formula is referred to as a **recurrence equation** (or simply as a **recurrence**). We can solve this recurrence by repeatedly substituting for t_{rSum} as shown:

$$\begin{aligned}t_{rSum}(n) &= 2 + t_{rSum}(n-1) \\&= 2 + 2 + t_{rSum}(n-2) \\&= 4 + t_{rSum}(n-2) \\&\vdots \\&= 2n + t_{rSum}(0) \\&= 2n + 2, \quad n \geq 0\end{aligned}$$

So the step count for function `rSum` (Program 1.31) is $2n+2$. ■

Comparing the step counts of Programs 1.30 and 1.31, we see that the count for Program 1.31 is less than that for Program 1.30. However, we cannot conclude that Program 1.30 is slower than Program 1.31, because a step doesn't correspond to a definite time unit. A step of `rSum` may take more time than a step of `sum`, so `rSum` might well be (and we expect it to be) slower than `sum`.

The step count is useful in that it tells us how the run time for a program changes with changes in the instance characteristics. From the step count for `sum`, we see that if `n` is doubled, the run time will also double (approximately); if `n` increases by a factor of 10, we expect the run time to increase by a factor of 10; and so on. So we expect the run time to grow *linearly* in `n`.

Rather than introduce statements to increment `stepCount`, we can build a table in which we list the total number of steps that each statement contributes to `stepCount`. We can arrive at this figure by first determining the number of steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed. Combining these two quantities gives us the total contribution of each statement. We can then add the contributions of all statements to obtain the step count for the entire program. This approach to obtaining the step count is called **profiling**.

The s/e of a statement is the amount by which stepCount changes as a result of the execution of that statement. An important difference between the step count of a statement and its s/e is illustrated by the following example. The statement

```
x = sum(a,m);
```

has a step count of 1, while the total change in `stepCount` resulting from the execution of this statement is actually 1 plus the change resulting from the invocation of `sum` (i.e., $2m+3$). Therefore, the s/e of the above statement is $1+2m+3 = 2m+4$.

Figure 2.7 lists the number of steps per execution and the frequency of each of the statements in function `sum` (Program 1.30). The total number of steps required by the program is $2n+3$. Note that the frequency of the `for` statement is $n+1$ and not n because `i` has to be incremented to n before the `for` loop can terminate.

Statement	s/e	Frequency	Total steps
T sum(T a[], int n)	0	0	0
{	0	0	0
T theSum = 0;	1	1	1
for (int i = 0; i < n; i++)	1	$n + 1$	$n + 1$
theSum += a[i];	1	n	n
return theSum;	1	1	1
}	0	0	0
Total			$2n + 3$

Figure 2.7 Step count for Program 1.30

Program 2.19 transposes a `rows × rows` matrix `a[0:rows-1][0:rows-1]`. Recall that `b` is the transpose of `a` iff (if and only if) `b[i][j] = a[j][i]` for all `i` and `j`.

```
template<class T>
void transpose(T **a, int rows)
// In-place transpose of matrix a[0:rows-1][0:rows-1].
{
    for (int i = 0; i < rows; i++)
        for (int j = i+1; j < rows; j++)
            swap(a[i][j], a[j][i]);
}
```

Program 2.19 Matrix transpose

Figure 2.8 gives the step-count table. Let us derive the frequency of the second `for` statement. For each value of `i`, this statement is executed `rows-i` times. So its frequency is

$$\sum_{i=0}^{rows-1} (rows - i) = \sum_{q=1}^{rows} q = rows(rows + 1)/2$$

Statement	s/e	Frequency	Total steps
void transpose(T **a, int rows)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	rows + 1	rows + 1
for (int j = i+1; j < rows; j++)	1	rows(rows + 1)/2	rows(rows + 1)/2
swap(a[i][j], a[j][i]);	1	rows(rows - 1)/2	rows(rows - 1)/2
}	0	0	0
Total			$rows^2 + rows + 1$

Figure 2.8 Step count for Program 2.19

The frequency for the `swap` statement is

$$\sum_{i=0}^{rows-1} (rows - i - 1) = \sum_{q=0}^{rows-1} q = rows(rows - 1)/2$$

In some cases the number of steps per execution of a statement varies from one execution to the next, for example, for the assignment statement of function `inef` (Program 2.20). Function `inef` is a very inefficient way to compute the prefix sums $b[j]$:

$$b[j] = \sum_{i=0}^j a[i] \text{ for } j = 0, 1, \dots, n - 1$$

```
template <class T>
void inef(T a[], T b[], int n)
{// Compute prefix sums.
    for (int j = 0; j < n; j++)
        b[j] = sum(a, j + 1);
}
```

Program 2.20 Inefficient prefix sums

The step count for `sum(a, m)` has already been determined to be $2m+3$ (see Example 2.19). Therefore, the number of steps per execution of the assignment statement $b[j] = \text{sum}(a, j + 1)$ of `inef` is $2j+6$. To arrive at this step count, we have added 1 to the step count of `sum` to account for the cost of invoking the function `sum` and

of assigning the function value to $b[j]$. The frequency of this assignment statement is n . But the total number of steps resulting from this statement is not $(2j+6)n$. Instead, the number of steps is

$$\sum_{j=0}^{n-1} (2j + 6) = n(n + 5)$$

Figure 2.9 gives the complete analysis for this function.

Statement	s/e	Frequency	Total steps
void inef(T a[], T b[], int n)	0	0	0
{	0	0	0
for (int j = 0; j < n; j++)	1	$n + 1$	$n + 1$
b[j] = sum(a, j + 1);	$2j + 6$	n	$n(n + 5)$
}	0	0	0
Total			$n^2 + 6n + 1$

Figure 2.9 Step count for Program 2.20

The notions of best, worst, and average operation counts are easily extended to the case of step counts. Examples 2.21 and 2.22 illustrate these notions.

Example 2.21 [Sequential Search] Figures 2.10 and 2.11 show the best- and worst-case step-count analyses for function `sequentialSearch` (Program 2.1).

Statement	s/e	Frequency	Total steps
int sequentialSearch(T a[], int n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++);	1	1	1
if (i == n) return -1;	1	1	1
else return i;	1	1	1
}	0	0	0
Total			4

Figure 2.10 Best-case step count for Program 2.1

For the average step-count analysis for a successful search, we assume that the n values in a are distinct and that in a successful search, x has an equal probability of being any one of these values. Under these assumptions the average step count for a successful search is the sum of the step counts for the n possible successful searches divided by n . To obtain this average, we first obtain the step count for the case $x = a[j]$ where j is in the range $[0, n - 1]$ (see Figure 2.12).

Now we obtain the average step count for successful searches:

Statement	s/e	Frequency	Total steps
int sequentialSearch(T a[], int n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++);	1	n + 1	n + 1
if (i == n) return -1;	1	1	1
else return i;	1	0	0
}	0	0	0
Total			n + 3

Figure 2.11 Worst-case step count for Program 2.1

Statement	s/e	Frequency	Total steps
int sequentialSearch(T a[], int n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++);	1	j + 1	j + 1
if (i == n) return -1;	1	1	1
else return i;	1	1	1
}	0	0	0
Total			j + 4

Figure 2.12 Step count for Program 2.1 when x = a[j]

$$\frac{1}{n} \sum_{j=0}^{n-1} (j + 4) = (n + 7)/2$$

This value is a little more than half the step count for an unsuccessful search.

Now suppose that successful searches occur only 80 percent of the time and that each `a[i]` still has the same probability of being searched for. The average step count for `sequentialSearch` is

$$\begin{aligned} &.8 * (\text{average count for successful searches}) + .2 * (\text{count for an unsuccessful search}) \\ &= .8(n + 7)/2 + .2(n + 3) \\ &= .6n + 3.4 \end{aligned}$$

Example 2.22 [Insertion into a Sorted Array] The best- and worst-case step counts for function `insert` (Program 2.10) are obtained in Figures 2.13 and 2.14, respectively.

Statement	s/e	Frequency	Total steps
void insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = n - 1; i >= 0 && x < a[i]; i--)	1	1	1
a[i+1] = a[i];	1	0	0
a[i+1] = x;	1	1	1
n++; // one element added to a	1	1	1
}	0	0	0
Total			4

Figure 2.13 Best-case step count for Program 2.10

Statement	s/e	Frequency	Total steps
void insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = n - 1; i >= 0 && x < a[i]; i--)	1	$n + 1$	$n + 1$
a[i+1] = a[i];	1	n	n
a[i+1] = x;	1	1	1
n++; // one element added to a	1	1	1
}	0	0	0
Total			$2n + 4$

Figure 2.14 Worst-case step count for Program 2.10

For the average step count, assume that x has an equal chance of being inserted into any of the possible $n+1$ positions. If x is eventually inserted into position j , $j \geq 0$, then the step count is $2n-2j+4$. So the average count is

$$\begin{aligned}
 \frac{1}{n+1} \sum_{j=0}^n (2n - 2j + 4) &= \frac{1}{n+1} [2 \sum_{j=0}^n (n - j) + \sum_{j=0}^n 4] \\
 &= \frac{1}{n+1} [2 \sum_{k=0}^n k + 4(n+1)] \\
 &= \frac{1}{n+1} [n(n+1) + 4(n+1)] \\
 &= \frac{(n+4)(n+1)}{n+1} \\
 &= n+4
 \end{aligned}$$

The average count is 2 more than half the worst-case count. ■

EXERCISES

8. According to the analysis in Example 2.8, Program 2.3 makes four additions and eight multiplications when evaluating the polynomial $3x^4 + 4x^3 + 5x^2 + 6x + 7$, and Program 2.4 makes four additions and four multiplications. Identify these additions and multiplications for the case $x = 2$. Do this by showing the precise numbers that are being added or multiplied.
9. Give the rank array r for the case when $a[0:8] = [3, 2, 6, 5, 9, 4, 7, 1, 8]$ (see Example 2.9).
10. Consider the selection sort function of Program 2.7. Draw a figure similar to Figure 2.4(a) for the case when $a[0:6] = [3, 2, 6, 5, 9, 4, 8]$.
11. Consider the bubbling pass function of Program 2.8. Draw a figure similar to Figure 2.4(b) for the case when $a[0:6] = [3, 2, 6, 5, 9, 4, 8]$.
12. For the bubble sort function of Program 2.9, draw a figure similar to Figure 2.4(c) for the case when $a[0:6] = [3, 2, 6, 5, 9, 4, 8]$.
13. Suppose that we are to insert 3 into the sorted array $a[0:6] = [1, 2, 4, 6, 7, 8, 9]$. Draw a figure similar to Figure 2.5(a). Your figure should show the progress of Program 2.10.
14. The array $a[0:8] = [g, h, i, f, c, a, d, b, e]$ is to be sorted using a rank sort. The ranks are determined to be $r[0:8] = [6, 7, 8, 5, 2, 0, 3, 1, 4]$. Draw a figure similar to Figures 2.5(b) and (c) to show the progress of the in-place rearrangement function of Program 2.11.
15. (a) Suppose that the array $a[0:9] = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$ is sorted using the early-terminating version of selection sort (Program 2.12). Draw a figure similar to Figure 2.6(a) to show the progress of the sort.
 (b) Do part (a) for the case when $a[0:8] = [8, 4, 5, 2, 1, 6, 7, 3, 0]$.
16. The array $a[0:9] = [4, 2, 6, 7, 1, 0, 9, 8, 5, 3]$ is sorted using the early-terminating version of bubble sort (Program 2.13). Draw a figure similar to Figure 2.6(b) to show the progress of the sort.
17. The array $a[0:9] = [4, 2, 6, 7, 1, 0, 9, 8, 5, 3]$ is to be sorted using insertion sort (Program 2.14). Draw a figure similar to Figure 2.6(c) to show the progress of the sort.
18. How many additions (i.e., invocations of `increment`) are done in the `for` loop of function `sum` (Program 1.30)?

19. How many multiplications are performed by the function `factorial` (Program 1.29)?
 20. Create an input array `a` that causes function `rearrange` (Program 2.11) to do $n - 1$ element reference swaps and $n - 1$ rank swaps.
 21. How many additions are performed between pairs of array elements by function `matrixAdd` (Program 2.21)?
-

```
template<class T>
void matrixAdd( T **a, T **b, T **c, int numberOfRows,
                int numberOfColumns)
{// Add matrices a and b to obtain matrix c.
    for (int i = 0; i < numberOfRows; i++)
        for (int j = 0; j < numberOfColumns; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

Program 2.21 Matrix addition

22. How many swap operations are performed by function `transpose` (Program 2.19)?
 23. Determine the number of multiplications done by function `squareMatrixMultiply` (Program 2.22), which multiplies two $n \times n$ matrices.
-

```
template<class T>
void squareMatrixMultiply(T **a, T **b, T **c, int n)
{// Multiply the n x n matrices a and b to get c.
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
        {
            T sum = 0;
            for (int k = 0; k < n; k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
}
```

Program 2.22 Multiply two $n \times n$ matrices

24. Determine the number of multiplications done by function `matrixMultiply` (Program 2.23), which multiplies an $m \times n$ matrix and an $n \times p$ matrix.

```
template<class T>
void matrixMultiply(T **a, T **b, T **c, int m, int n, int p)
{// Multiply the m x n matrix a and the n x p matrix b
// to get c.
    for (int i = 0; i < m; i++)
        for (int j = 0; j < p; j++)
    {
        T sum = 0;
        for (int k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Program 2.23 Multiply an $m \times n$ and an $n \times p$ matrix

25. Determine the number of swap operations performed by function **permutations** (Program 1.32).
26. Method **minmax** (Program 2.24) determines the locations of the minimum and maximum elements in an array **a**. Let **n** be the instance characteristic. What is the number of comparisons between elements of **a**? Program 2.25 gives an alternative function to determine the locations of the minimum and maximum elements. What are the best-case and worst-case numbers of comparisons between elements of **a**? What can you say about the expected relative performance of the two functions?
27. How many comparisons between the **a[]**s and **x** are made by the recursive function **rSequentialSearch** (Program 2.2)?
28. Program 2.26 gives an alternative iterative sequential search function. What is the worst-case number of comparisons between **x** and the elements of **a**? Compare this number with the corresponding number for Program 2.1. Which function should run faster? Why?
29. (a) Introduce statements to increment **stepCount** at all appropriate points in Program 2.27.
(b) Simplify the resulting program by eliminating statements. Both the simplified program and the program of part (a) should compute the same value for **stepCount**.
(c) What is the exact value of **stepCount** when the program terminates? You may assume that the initial value of **stepCount** is 0.

```
template<class T>
bool minmax(T a[], int n, int& indexOfMin, int& indexOfMax)
{// Locate min and max elements in a[0:n-1].
// Return false if less than one element.
    if (n < 1) return false;
    indexOfMin = indexOfMax = 0; // initial guess
    for (int i = 1; i < n; i++)
    {
        if (a[indexOfMin] > a[i]) indexOfMin = i;
        if (a[indexOfMax] < a[i]) indexOfMax = i;
    }
    return true;
}
```

Program 2.24 Finding the minimum and maximum

```
template<class T>
bool minmax(T a[], int n, int& indexOfMin, int& indexOfMax)
{// Locate min and max elements in a[0:n-1].
// Return false if less than one element.
    if (n < 1) return false;
    indexOfMin = indexOfMax = 0; // initial guess
    for (int i = 1; i < n; i++)
        if (a[indexOfMin] > a[i]) indexOfMin = i;
        else if (a[indexOfMax] < a[i]) indexOfMax = i;
    return true;
}
```

Program 2.25 Another function to find the minimum and maximum

- (d) Use the frequency method to determine the step count for Program 2.27.
Clearly show the step-count table.
30. Do Exercise 29 for each of the following functions:
- indexOfMax (Program 1.37).
 - minmax (Program 2.24).
 - minmax (Program 2.25). Determine the worst-case step count.
 - factorial (Program 1.29).
 - polyEval (Program 2.3).

```
template<class T>
int sequentialSearch(T a[], int n, const T& x)
{// Search the unordered list a[0:n-1] for x.
// Return position if found; return -1 otherwise.
    a[n] = x; // assume extra position available
    int i;
    for (i = 0; a[i] != x; i++);
    if (i == n) return -1;
    return i;
}
```

Program 2.26 Another sequential search function

```
void d(int x[], int n)
{
    for (int i = 0; i < n; i += 2)
        x[i] += 2;

    i = 1;
    while (i <= n/2)
    {
        x[i] += x[i+1];
        i++;
    }
}
```

Program 2.27 Method for Exercise 29

- (f) `horner` (Program 2.4).
- (g) `rank` (Program 2.5).
- (h) `permutations` (Program 1.32).
- (i) `sequentialSearch` (Program 2.26). Determine the worst-case step count.
- (j) `selectionSort` (Program 2.7). Determine the best- and worst-case step counts.
- (k) `selectionSort` (Program 2.12). Determine the best- and worst-case step counts.
- (l) `insertionSort` (Program 2.14). Determine the worst-case step count.
- (m) `insertionSort` (Program 2.15). Determine the worst-case step count.
- (n) `bubbleSort` (Program 2.9). Determine the worst-case step count.

- (o) `bubbleSort` (Program 2.13). Determine the worst-case step count.
 - (p) `matrixAdd` (Program 2.21).
 - (q) `squareMatrixMultiply` (Program 2.22).
31. Do Exercise 29 parts (a), (b), and (c) for the following functions:
- (a) `transpose` (Program 2.19).
 - (b) `inef` (Program 2.20).
32. Determine the average step counts for the following functions:
- (a) `rSequentialSearch` (Program 2.2).
 - (b) `sequentialSearch` (Program 2.26).
 - (c) `insert` (Program 2.10).
33. (a) Do Exercise 29 for Program 2.23.
 (b) Under what conditions will it be profitable to interchange the two outermost `for` loops?
34. Compare the worst-case number of element reference moves made by functions `selectionSort` (Program 2.12), `insertionSort` (Program 2.15), `bubbleSort` (Program 2.13), and rank sort using Program 2.11.
35. Must a program exhibit its worst-case time behavior and worst-case space behavior at the same time (i.e., for the same input)? Prove your answer.
36. Use repeated substitution to solve the following recurrences (see Example 2.20).

$$(a) t(n) = \begin{cases} 2 & n = 0 \\ 2 + t(n - 1) & n > 0 \end{cases}$$

$$(b) t(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 1 + t(n - 2) & n > 0 \end{cases}$$

$$(c) t(n) = \begin{cases} 0 & n = 0 \\ 2n + t(n - 1) & n > 0 \end{cases}$$

$$(d) t(n) = \begin{cases} 1 & n = 0 \\ 2 * t(n - 1) & n > 0 \end{cases}$$

$$(e) t(n) = \begin{cases} 1 & n = 0 \\ 3 * t(n - 1) & n > 0 \end{cases}$$

CHAPTER 3

ASYMPTOTIC NOTATION

BIRD'S-EYE VIEW

Chapter 2 showed you how to analyze the space and time complexities of a program. The methods of that chapter are somewhat cumbersome because they attempt to obtain exact counts, rather than estimates. In this chapter we review asymptotic notation, which is used to make statements about program performance when the instance characteristics are large. When we use this notation, we need only estimate the step count. Although the big oh notation is the most popular asymptotic notation used in the performance analysis of programs, the omega, theta, and little oh notations also are in common use.

Asymptotic notation is first introduced in an informal manner in Section 3.2. The informal treatment of this section is adequate to follow all the analyses done in this book. A more rigorous treatment is presented in Section 3.3. You may omit Section 3.3 and not face any dire consequences.

The use of asymptotic notation is illustrated through the applications developed in Chapters 1 and 2. Additionally, an important and efficient search method—binary search of a sorted array—is developed and analyzed in this chapter. This search method is also available as the STL algorithm `binary_Search`.

3.1 INTRODUCTION

Two important reasons to determine operation and step counts are (1) to predict the growth in run time as the instance characteristics increase and (2) to compare the time complexities of two programs that perform the same task. When using operation counts, we focus on certain “key” operations and ignore all others. Therefore, you must be very cautious when using an operation count for either of the above purposes. For example, a program may do $2n$ comparisons, but the total number of computation steps could be $6n^2 + 8n$. It would be incorrect to use the comparison count of $2n$ to conclude that the run time grows linearly in n . It would also be incorrect to conclude that a program that makes $2n$ comparisons is faster than a program, for the same task, that makes $3n$ comparisons; the $3n$ comparison program may actually do less total work than is done by the $2n$ comparison program.

The operation count method accounts for only some of the work that is done in a program. Step counts attempt to overcome this deficiency by accounting for all work. However, the notion of a step is inexact. Both the instructions $x = y$ and $x = y+z+(x/y)$ count as one step. Therefore, two analysts may arrive at $4n^2 + 6n + 2$ and $7n^2 + 3n + 4$ as the step count for the same program. We cannot conclude that the run time will grow as either $4n^2 + 6n + 2$ or $7n^2 + 3n + 4$, because any step count of the form $c_1n^2 + c_2n + c_3$, where $c_1 > 0$, c_2 , and c_3 are constants, could be a correct step count for the program. Because of the inexactness of what a step stands for, the exact step count isn’t very useful for comparative purposes either. However, when the difference in the step counts of two programs is very large as in $3n + 3$ versus $900n + 10$. We might feel quite safe in predicting that the program with step count $3n + 3$ will run in less time than the one with step count $900n + 10$.

We can use the step count to accurately predict the growth in run time for large instance sizes (i.e., in the asymptote as n approaches infinity) and to predict the relative performance of two programs when the instance size becomes large. Suppose that the step count of a program is determined to be $c_1n^2 + c_2n + c_3$, $c_1 > 0$. When n becomes large, the c_1n^2 term will be much larger than the remaining terms $c_2n + c_3$. The ratio of these two expressions is $r(n) = (c_2n + c_3)/(c_1n^2) = c_2/(c_1n) + c_3/(c_1n^2)$. Figure 3.1 plots $r(n)$ for the case $c_1 = 1$, $c_2 = 2$, and $c_3 = 3$. Even though $r(n)$ never equals 0 for any finite n , $r(n)$ gets closer and closer to 0 as we make n bigger and bigger.

Regardless of the values of $c_1 > 0$, c_2 , and c_3 , the ratio $r(n)$ approaches 0 as n approaches infinity; that is,

$$\lim_{n \rightarrow \infty} \left(\frac{c_2}{c_1n} + \frac{c_3}{c_1n^2} \right) = 0$$

So for large n , $c_2n + c_3$ is insignificant when compared to c_1n^2 , and run time may be approximated by the c_1n^2 term. Let n_1 and n_2 be two large values of n . We conclude that

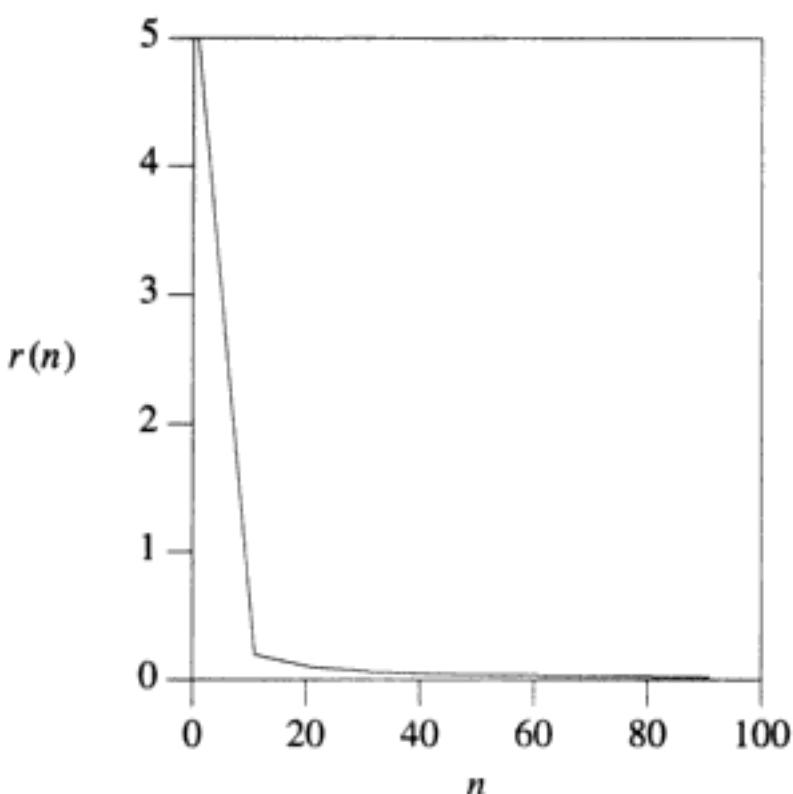


Figure 3.1 Plot of $r(n) = 2/n + 3/n^2$

$$\frac{t(n_1)}{t(n_2)} \approx \frac{c_1 n_1^2}{c_1 n_2^2} = \left(\frac{n_1}{n_2}\right)^2$$

Therefore, the run time is expected to increase by a factor of 4 (approximately) when the instance size is doubled; the run time increases by a factor of 9 when the instance size is tripled; and so on. To make this conclusion, all we need to know is that the biggest term in the step count is an n^2 term; the value of the coefficient c_1 is irrelevant to the conclusion.

Suppose that programs A and B perform the same task. Assume that John has determined the step counts of these programs to be $t_A(n) = n^2 + 3n$ and $t_B(n) = 43n$. It is entirely possible that Mary's analysis of the same programs yields $t_A(n) = 2n^2 + 3n$ and $t_B(n) = 83n$. In fact, assuming that John's analysis is correct, all other correct analyses would result in $t_A(n) = c_1 n^2 + c_2 n + c_3$ and $t_B(n) = c_4 n$, where c_1, c_2, c_3 , and c_4 are constants, $c_1 > 0$, and $c_4 > 0$.

To see what conclusion we can draw about the relative performance of programs A and B knowing that the constant coefficients may vary from analyst to analyst, examine the plot of Figure 3.2. First, look at the curves for John's analysis, $t_A(n) = n^2 + 3n$ and $t_B(n) = 43n$. We conclude that for $n < 40$, program A is faster; for

$n > 40$, program B is faster; and $n = 40$ is the break-even point between the two programs. Now suppose that the analysis had instead concluded that $t_B(n) = 83n$. In this case we would conclude that for $n < 80$, program A is faster; for $n > 80$, program B is faster; and $n = 80$ is the break-even point between the two programs. Our conclusion that program B is faster than program A for large n does not change; only the break-even point changes.

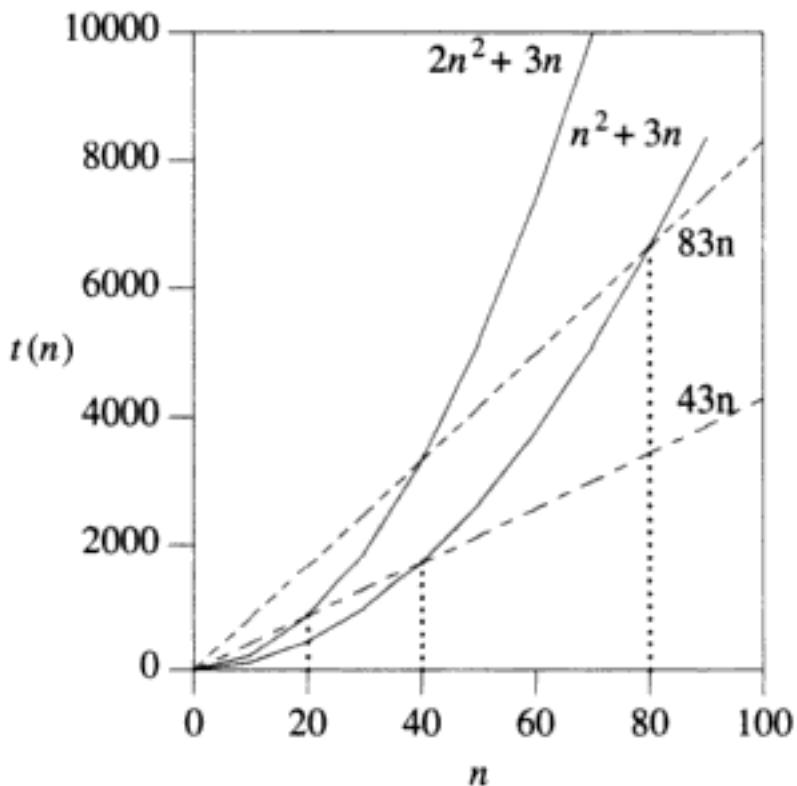


Figure 3.2 Comparing run time functions

What if John's analysis had concluded $t_A(n) = 2n^2 + 3n$? From Figure 3.2, we see that regardless of whether $t_B(n) = 43n$ or $83n$, program B remains faster than program A when n is suitably large ($n > 20$ when $t_B(n) = 43n$ and $n > 40$ when $t_B(n) = 83n$).

To arrive at the conclusion that program B is faster than program A when n is large, all we need to know is that the biggest term in the step count for program A is an n^2 term while that for program B is an n term; the values of the coefficients c_1 through c_s are irrelevant to this conclusion. Asymptotic analysis focuses on determining the biggest terms (but not their coefficients) in the complexity function.

3.2 ASYMPTOTIC NOTATION

3.2.1 Big Oh Notation (O)

Definition 3.1 Let $p(n)$ and $q(n)$ be two nonnegative functions. $p(n)$ is **asymptotically bigger** ($p(n)$ *asymptotically dominates* $q(n)$) than the function $q(n)$ iff

$$\lim_{n \rightarrow \infty} \frac{q(n)}{p(n)} = 0 \quad (3.1)$$

$q(n)$ is **asymptotically smaller** than $p(n)$ iff $p(n)$ is asymptotically bigger than $q(n)$. $p(n)$ and $q(n)$ are **asymptotically equal** iff neither is asymptotically bigger than the other. ■

Example 3.1 Since

$$\lim_{n \rightarrow \infty} \frac{10n + 7}{3n^2 + 2n + 6} = \frac{10/n + 7/n^2}{3 + 2/n + 6/n^2} = 0/3 = 0$$

$3n^2 + 2n + 6$ is asymptotically bigger than $10n + 7$ and $10n + 7$ is asymptotically smaller than $3n^2 + 2n + 6$. A similar derivation shows that $8n^4 + 9n^2$ is asymptotically bigger than $100n^3 - 3$, and that $2n^2 + 3n$ is asymptotically bigger than $83n$. $12n + 6$ is asymptotically equal to $6n + 2$. ■

In the following discussion the function $f(n)$ denotes the time or space complexity of a program as a function of the instance characteristic n . Since the time or space requirements of a program are nonnegative quantities, we assume that the function f has a nonnegative value for all values of n . Further, since n denotes an instance characteristic, we assume that $n \geq 0$. The function $f(n)$ will, in general, be a sum of terms. For example, the terms of $f(n) = 9n^2 + 3n + 12$ are $9n^2$, $3n$, and 12 . We may compare pairs of terms to determine which is bigger (see Definition 3.1). The biggest term in the example $f(n)$ is $9n^2$.

Figure 3.3 gives the terms that occur frequently in a step-count analysis. Although all the terms in Figure 3.3 have a coefficient of 1, in an actual analysis, the coefficients of these terms may have a different value.

We do not associate a logarithmic base with the functions in Figure 3.3 that include $\log n$ because for any constants a and b greater than 1, $\log_a n = \log_b n / \log_b a$. So $\log_a n$ and $\log_b n$ are asymptotically equal.

Using Definition 3.1, we obtain the following ordering for the terms of Figure 3.3 ($<$ is to be read as “is asymptotically smaller than”):

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

Term	Name
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Figure 3.3 Commonly occurring terms

Asymptotic notation describes the behavior of the time or space complexity for large instance characteristics. Although we will develop asymptotic notation with reference to step counts alone, our development also applies to space complexity and operation counts. The terms *time complexity* and *step count* are used as synonyms. When the instance characteristic is described by a single variable, say n , asymptotic notation describes the complexity using a single term, the asymptotically biggest term in the step count.

The notation $f(n) = O(g(n))$ (read as “ $f(n)$ is big oh of $g(n)$ ”) means that $f(n)$ is asymptotically smaller than or equal to $g(n)$. Therefore, in an asymptotic sense $g(n)$ is an upper bound for $f(n)$. You may use this as a working definition of “big oh”; a formal definition is provided in Section 3.3.1.

Example 3.2 From Example 3.1 and the working definition of big oh, it follows that $10n + 7 = O(3n^2 + 2n + 6)$; $100n^3 - 3 = O(8n^4 + 9n^2)$; $12n + 6 = O(6n + 2)$; $3n^2 + 2n + 6 \neq O(10n + 7)$; and $8n^4 + 9n^2 \neq O(100n^3 - 3)$. ■

Although Example 3.2 uses the big oh notation in a correct way, it is customary to use $g(n)$ functions that are **unit terms** (i.e., $g(n)$ is a single term whose coefficient is 1) except when $f(n) = 0$. In addition, it is customary to use, for $g(n)$, the smallest unit term for which the statement $f(n) = O(g(n))$ is true. When $f(n) = 0$, it is customary to use $g(n) = 0$.

Example 3.3 The customary way to describe the asymptotic behavior of the functions used in Example 3.2 is $10n + 7 = O(n)$; $100n^3 - 3 = O(n^3)$; $12n + 6 = O(n)$; $3n^2 + 2n + 6 \neq O(n)$; and $8n^4 + 9n^2 \neq O(n^3)$. ■

In asymptotic complexity analysis, we determine the biggest term in the complexity; the coefficient of this biggest term is set to 1. The unit terms of a step-count

function are step-count terms with their coefficients changed to 1. For example, the unit terms of $3n^2 + 6n \log n + 7n + 5$ are n^2 , $n \log n$, n , and 1; the biggest unit term is n^2 . So when the step count of a program is $3n^2 + 6n \log n + 7n + 5$, we say that its asymptotic complexity is $O(n^2)$.

Example 3.4 In Example 2.19, we determined that $t_{\text{sum}}(n) = 2n + 3$. Since the biggest unit term in $t_{\text{sum}}(n)$ is n , $t_{\text{sum}}(n) = O(n)$.

Since $t_{\text{rSum}}(n) = 2n + 2$ (see Example 2.20), $t_{\text{rSum}}(n) = O(n)$.

The step count for Program 2.19 is $\text{rows}^2 + \text{rows} + 1$ (see Figure 2.8). The biggest unit term is rows^2 . Therefore, $t_{\text{transpose}}(\text{rows}) = O(\text{rows}^2)$. ■

Notice that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, saying that $O(g(n)) = f(n)$ is meaningless. The use of the symbol $=$ is unfortunate, as this symbol commonly denotes the equals relation. We can avoid some of the confusion that results from the use of this symbol (which is standard terminology) by reading the symbol $=$ as “is” and not as “equals.”

Definition 3.2 Let $t(m, n)$ and $u(m, n)$ be two terms. $t(m, n)$ is asymptotically bigger than $u(m, n)$ (equivalently, $u(m, n)$ is asymptotically smaller than $t(m, n)$) iff either

$$\lim_{n \rightarrow \infty} \frac{u(m, n)}{t(m, n)} = 0 \text{ and } \lim_{m \rightarrow \infty} \frac{u(m, n)}{t(m, n)} \neq \infty$$

or

$$\lim_{n \rightarrow \infty} \frac{u(m, n)}{t(m, n)} \neq \infty \text{ and } \lim_{m \rightarrow \infty} \frac{u(m, n)}{t(m, n)} = 0$$

We may obtain a working definition of big oh for the case of functions in more than one variable as follows.

- Let $f(m, n)$ be the step count of a program. From $f(m, n)$ remove all terms that are asymptotically smaller than at least one other term in $f(m, n)$.
- Change the coefficients of all remaining terms to 1.

Example 3.5 Consider $f(m, n) = 3m^2n + m^3 + 10mn + 2n^2$. $10mn$ is smaller than $3m^2n$ because

$$\lim_{n \rightarrow \infty} \frac{10mn}{3m^2n} = \frac{10}{3m} \neq \infty \text{ and } \lim_{m \rightarrow \infty} \frac{10mn}{3m^2n} = \lim_{m \rightarrow \infty} \frac{10}{3m} = 0$$

None of the remaining terms is smaller than another. Dropping the $10mn$ term and changing the coefficients of the remaining terms to 1, we get $f(m, n) = O(m^2n + m^3 + n^2)$. ■

3.2.2 Omega (Ω) and Theta (Θ) Notations

Although the big oh notation is the most frequently used asymptotic notation, the omega and theta notations are sometimes used to describe the asymptotic complexity of a program. We provide a working definition of these notations in this section. See Sections 3.3.2 and 3.3.3 for a more formal definition.

The notation $f(n) = \Omega(g(n))$ (read as “ $f(n)$ is omega of $g(n)$ ”) means that $f(n)$ is asymptotically bigger than or equal to $g(n)$. Therefore, in an asymptotic sense, $g(n)$ is a lower bound for $f(n)$. The notation $f(n) = \Theta(g(n))$ (read as “ $f(n)$ is theta of $g(n)$ ”) means that $f(n)$ is asymptotically equal to $g(n)$.

Example 3.6 $10n + 7 = \Omega(n)$ because $10n + 7$ is asymptotically equal to n ; $100n^3 - 3 = \Omega(n^3)$; $12n + 6 = \Omega(n)$; $3n^3 + 2n + 6 = \Omega(n)$; $8n^4 + 9n^2 = \Omega(n^3)$; $3n^3 + 2n + 6 \neq \Omega(n^5)$; and $8n^4 + 9n^2 \neq \Omega(n^5)$.

$10n + 7 = \Theta(n)$ because $10n + 7$ is asymptotically equal to n ; $100n^3 - 3 = \Theta(n^3)$; $12n + 6 = \Theta(n)$; $3n^3 + 2n + 6 \neq \Theta(n)$; $8n^4 + 9n^2 \neq \Theta(n^3)$; $3n^3 + 2n + 6 \neq \Theta(n^5)$; and $8n^4 + 9n^2 \neq \Theta(n^5)$.

Since $t_{\text{sum}}(n) = 2n + 3$ (see Example 2.19) and $2n + 3$ is asymptotically equal to n , $t_{\text{sum}}(n) = \Theta(n)$.

Since $t_{\text{rSum}}(n) = 2n + 2$ (see Example 2.20) and $2n + 2$ asymptotically equals n , $t_{\text{rSum}}(n) = \Theta(n)$.

The step count for Program 2.19 is $\text{rows}^2 + \text{rows} + 1$ (see Figure 2.8), and $\text{rows}^2 + \text{rows} + 1$ asymptotically equals rows^2 . Therefore, $t_{\text{transpose}}(\text{rows}) = \Theta(\text{rows}^2)$.

The best-case step count for `sequentialSearch` (Program 2.1) is 4 (Figure 2.10), the worst-case step count is $n + 3$, and the average step count is $0.6n + 3.4$. So the best-case asymptotic complexity of `sequentialSearch` is $\Theta(1)$, and the worst-case and average complexities are $\Theta(n)$. It is also correct to say that the complexity of `sequentialSearch` is $\Omega(1)$ and $O(n)$ because 1 is a lower bound (in an asymptotic sense) and n is an upper bound (in an asymptotic sense) on the step count.

From Figures 2.13 and 2.14, it follows that $4 \leq t_{\text{insert}}(n) \leq 2n + 4$. Therefore, $t_{\text{insert}}(n)$ is both $\Omega(1)$ and $O(n)$. ■

At times it is useful to interpret $O(g(n))$, $\Omega(g(n))$, and $\Theta(g(n))$ as being the following sets:

$$O(g(n)) = \{f(n) | f(n) = O(g(n))\}$$

$$\Omega(g(n)) = \{f(n) | f(n) = \Omega(g(n))\}$$

$$\Theta(g(n)) = \{f(n) | f(n) = \Theta(g(n))\}$$

Under this interpretation, statements such as $O(g_1(n)) = O(g_2(n))$ and $\Theta(g_1(n)) = \Theta(g_2(n))$ are meaningful. When using this interpretation, it is also convenient to read $f(n) = O(g(n))$ as “ f of n is in (or is a member of) big oh of g of n ” and so on.

The working definitions of big oh, omega, and theta are all you need to understand the analyses done in this book. The next section contains a more formal treatment of asymptotic notation that will help you with more complex analyses.

EXERCISES

1. Use Equation 3.1 to show that $p(n)$ is asymptotically bigger than $q(n)$ for the following functions:
 - $p(n) = 3n^4 + 2n^2$, $q(n) = 100n^2 + 6$
 - $p(n) = 6n^{1.5} + 12$, $q(n) = 100n$
 - $p(n) = 7n^2 \log n$, $q(n) = 10n^2$
 - $p(n) = 17n^2 2^n$, $q(n) = 100n 2^n + 33n$
2. Express the following step counts using big oh notation. Your $g(n)$ function should be the smallest possible unit term.
 - $2n^3 - 6n + 30$
 - $44n^{1.5} + 33n - 200$
 - $16n^2 \log n + 5n^2$
 - $31n^3 + 17n^2 \log n$
 - $23n2^n - 3n^3$
3. Use the working definition of big oh and Equation 3.1 to show the following:
 - $2n + 7 \neq O(1)$
 - $12n^2 + 8n + 7 \neq O(n)$
 - $5n^3 + 6n^2 \neq O(n^2)$
 - $15n^3 \log n + 16n^2 \neq O(n^3)$
4. Express the step counts of Exercise 2 using omega notation.
5. Use the working definition of omega and Equation 3.1 to show the following:
 - $2n + 7 \neq \Omega(n^2)$
 - $12n^2 + 8n + 7 \neq \Omega(n^3)$
 - $5n^3 + 6n^2 \neq \Omega(n^3 \log n)$
 - $15n^3 \log n + 16n^2 \neq \Omega(n^4)$
6. Express the step counts of Exercise 2 using theta notation.
7. Let $t(n)$ be the step count of a program. Express the following step-count information using asymptotic notation. Use the most appropriate $g(n)$ functions.

- (a) $6 \leq t(n) \leq 20$
 (b) $6 \leq t(n) \leq 2n$
 (c) $3n^2 + 1 \leq t(n) \leq 4n^2 + 3n + 9$
 (d) $3n^2 + 1 \leq t(n) \leq 4n^2 \log n + 3n^2 + 9$
 (e) $t(n) \geq 5n^3 + 7$
 (f) $t(n) \geq 32n \log n + 77n - 6$
 (g) $t(n) = 17n^2 + 3n$
8. Express the following step counts using big oh notation. m and n are instance characteristics.
- (a) $7m^2n^2 + 2m^3n + mn + 5mn^2$
 (b) $2m^2 \log n + 3mn + 5m \log n + m^2n^2$
 (c) $m^4 + n^3 + m^3n^2$
 (d) $3mn^2 + 7m^2n + 4mn + 8m + 2n + 16$

3.3 ASYMPTOTIC MATHEMATICS (OPTIONAL)

3.3.1 Big Oh Notation (O)

The big oh notation describes an upper bound on the asymptotic growth rate of the function f .

Definition 3.3 [Big oh] $f(n) = O(g(n))$ iff positive constants c and n_0 exist such that $f(n) \leq cg(n)$ for all n , $n \geq n_0$. ■

The definition states that the function f is at most c times the function g except possibly when n is smaller than n_0 . Here c is some positive constant. Thus g is an upper bound (up to a constant factor c) on the value of f for all suitably large n (i.e., $n \geq n_0$). Figure 3.4 illustrates what it means for a function $g(n)$ to upper bound (up to a constant factor c) another function $f(n)$. Although $f(n)$ may be less than, equal to, or greater than $cg(n)$ for several values of n , there must exist a value m of n beyond which $f(n)$ is never greater than $cg(n)$. The n_0 in the definition of big oh could be any integer $\geq m$.

When providing an upper-bound function g for f , we will normally use only simple functional forms. These typically contain a single term in n with a multiplicative constant of 1.

Example 3.7 [Linear Function] Consider $f(n) = 3n + 2$. When n is at least 2, $3n + 2 \leq 3n + n \leq 4n$. So $f(n) = O(n)$. Thus $f(n)$ is bounded from above by a linear function. We can arrive at the same conclusion in other ways. For example,

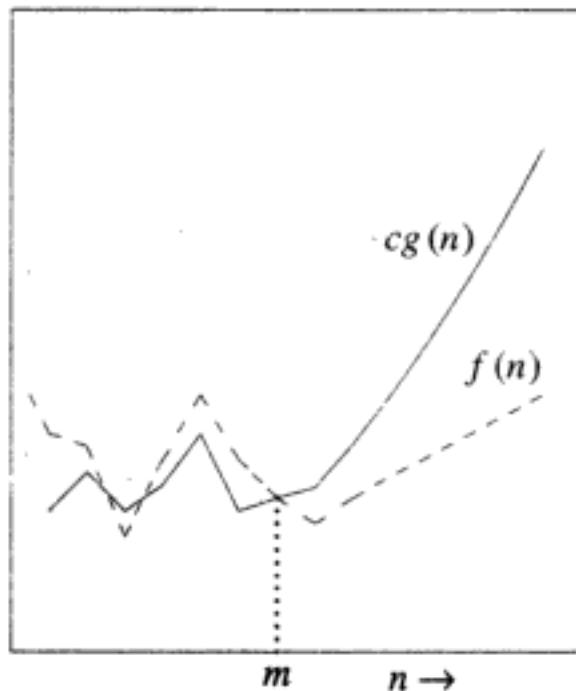


Figure 3.4 $g(n)$ is an upper bound (up to a constant factor c) on $f(n)$

$3n + 2 \leq 10n$ for $n > 0$. Therefore, we can also satisfy the definition of big oh by selecting $c = 10$ and n_0 equal to any integer greater than 0. Alternatively, $3n + 2 \leq 3n + 2n = 5n$ for $n \geq 1$, so we can satisfy the definition of big oh by setting $c = 5$ and $n_0 = 1$. The values of c and n_0 used to satisfy the definition of big oh are not important because we will be saying only that $f(n)$ is big oh of $g(n)$ and in this statement neither c nor n_0 play a role.

For $f(n) = 3n + 3$, we note that for $n \geq 3$, $3n + 3 \leq 3n + n \leq 4n$. So $f(n) = O(n)$. Similarly, $f(n) = 100n + 6 \leq 100n + n = 101n$ for $n \geq n_0 = 6$. Therefore, $100n + 6 = O(n)$. As expected, $3n + 2$, $3n + 3$, and $100n + 6$ are all big oh of n ; that is, they are bounded from above by a linear function (for suitably large n). ■

Example 3.8 [Quadratic Function] Suppose that $f(n) = 10n^2 + 4n + 2$. We see that for $n \geq 2$, $f(n) \leq 10n^2 + 5n$. Now we note that for $n \geq 5$, $5n \leq n^2$. Hence for $n \geq n_0 = 5$, $f(n) \leq 10n^2 + n^2 = 11n^2$. Therefore, $f(n) = O(n^2)$.

As another example of a quadratic complexity, consider $f(n) = 1000n^2 + 100n - 6$. We easily see that $f(n) \leq 1000n^2 + 100n$ for all n . Furthermore, $100n \leq n^2$ for $n \geq 100$. Hence $f(n) < 1001n^2$ for $n \geq n_0 = 100$. So $f(n) = O(n^2)$. ■

Example 3.9 [Exponential Function] As an example of exponential complexity, consider $f(n) = 6 * 2^n + n^2$. Observe that for $n \geq 4$, $n^2 \leq 2^n$. So $f(n) \leq 6 * 2^n + 2^n = 7 * 2^n$ for $n \geq 4$. Therefore, $6 * 2^n + n^2 = O(2^n)$. ■

Example 3.10 [Constant Function] When $f(n)$ is a constant, as in $f(n) = 9$ or $f(n) = 2033$, we write $f(n) = O(1)$. The correctness of this is easily established. For example, $f(n) = 9 \leq 9 * 1$; setting $c = 9$ and $n_0 = 0$ satisfies the definition of big oh. Similarly, $f(n) = 2033 \leq 2033 * 1$, and the definition of big oh is satisfied by setting $c = 2033$ and $n_0 = 0$. ■

Example 3.11 [Loose Bounds] $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. Although n^2 is an upper bound for $3n + 3$, it is not a tight upper bound; we can find a smaller function (in this case linear) that also satisfies the big oh relation.

$10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. Once again, n^4 does not provide a tight upper bound for $100n^2 + 4n + 2$.

Similarly, $6n2^n + 20 = O(n^22^n)$, but it is not a tight upper bound because we can find a smaller function, namely, $n2^n$, for which the definition of big oh is satisfied. That is, $6n2^n + 20 = O(n2^n)$. ■

Note that the strategy in each of the preceding derivations is to replace the low-order terms by higher-order terms until only a single term remains.

Example 3.12 [Incorrect Bounds] $3n + 2 \neq O(1)$, as there is no $c > 0$ and n_0 such that $3n + 2 \leq c$ for all n , $n \geq n_0$. We can use contradiction to prove this condition formally. Suppose that such a c and n_0 exist. Then $n \leq (c - 2)/3$ for all n , $n \geq n_0$. This is not true for $n > \max\{n_0, (c - 2)/3\}$.

To prove $10n^2 + 4n + 2 \neq O(n)$, suppose the equality holds. That is, $10n^2 + 4n + 2 = O(n)$. There exists a positive c and an n_0 such that $10n^2 + 4n + 2 \leq cn$ for all $n \geq n_0$. Dividing both sides of the relation by n , we get $10n + 4 + 2/n \leq c$ for $n \geq n_0$. This relation cannot be true because the left side increases as n increases, whereas the right side does not change. In particular, we get a contradiction for $n \geq \max\{n_0, (c - 4)/10\}$.

$f(n) = 3n^22^n + 4n2^n + 8n^2 \neq O(2^n)$. To prove this inequality, suppose that $f(n) = O(2^n)$. Then a $c > 0$ and an n_0 exist such that $f(n) \leq c * 2^n$ for $n \geq n_0$. Dividing both sides by 2^n , we get $3n^2 + 4n + 8n^2/2^n \leq c$ for $n \geq n_0$. Once again, the left side of the relation is an increasing function of n while the right side is constant. So the relation cannot hold for “large” n . ■

As illustrated in Example 3.11, the statement $f(n) = O(g(n))$ states only that $cg(n)$ is an upper bound on the value of $f(n)$ for all n , $n \geq n_0$. It doesn't say anything about how good or tight this bound is. Notice that $n = O(n^2)$, $n = O(n^{2.5})$, $n = O(n^3)$, and $n = O(2^n)$. For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as possible for which $f(n) = O(g(n))$. So although we often say $3n + 3 = O(n)$, we almost never say $3n + 3 = O(n^2)$, even though the latter statement is correct.

Theorem 3.1 obtains a very useful result concerning the order of $f(n)$ (i.e., the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n .

Theorem 3.1 If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = O(n^m)$.

Proof $f(n) \leq \sum_{i=0}^m |a_i|n^i \leq n^m \sum_0^m |a_i|n^{i-m} \leq n^m \sum_0^m |a_i|$ for $n \geq 1$. So $f(n) = O(n^m)$. ■

Example 3.13 Let us apply Theorem 3.1 to the functions of Examples 3.7, 3.8, and 3.10. For the three linear functions of Example 3.7, $m = 1$, and so these functions are $O(n)$. For the functions of Example 3.8, $m = 2$, and so all are $O(n^2)$. For the constants of Example 3.10, $m = 0$, so both constants are $O(1)$. ■

We can extend the strategy used in Example 3.12 to show that an upper bound is incorrect to the case when an upper bound is correct, as shown in the following theorem. *It is usually easier to show $f(n) = O(g(n))$ by using this theorem than by using the definition of big oh.*

Theorem 3.2 [Big oh ratio theorem] Let $f(n)$ and $g(n)$ be such that $\lim_{n \rightarrow \infty} f(n)/g(n)$ exists. $f(n) = O(g(n))$ iff $\lim_{n \rightarrow \infty} f(n)/g(n) \leq c$ for some finite constant c .

Proof If $f(n) = O(g(n))$, then positive c and an n_0 exist such that $f(n)/g(n) \leq c$ for all $n \geq n_0$. Hence $\lim_{n \rightarrow \infty} f(n)/g(n) \leq c$. Suppose that $\lim_{n \rightarrow \infty} f(n)/g(n) \leq c$. It follows that an n_0 exists for which $f(n) \leq \max\{1, c\} * g(n)$ for all $n \geq n_0$. ■

Example 3.14 $3n+2 = O(n)$ as $\lim_{n \rightarrow \infty} (3n+2)/n = 3$. $10n^2 + 4n + 2 = O(n^2)$ as $\lim_{n \rightarrow \infty} (10n^2 + 4n + 2)/n^2 = 10$. $6*2^n + n^2 = O(2^n)$ as $\lim_{n \rightarrow \infty} (6*2^n + n^2)/2^n = 6$. $2n^2 - 3 = O(n^4)$ as $\lim_{n \rightarrow \infty} (2n^2 - 3)/n^4 = 0$. $3n^2 + 5 \neq O(n)$ as $\lim_{n \rightarrow \infty} (3n^2 + 5)/n = \infty$. ■

3.3.2 Omega Notation (Ω)

The omega notation, which is the lower-bound analog of the big oh notation, permits us to bound the asymptotic growth rate of f from below.

Definition 3.4 [Omega] $f(n) = \Omega(g(n))$ iff positive constants c and n_0 exist such that $f(n) \geq cg(n)$ for all n , $n \geq n_0$. ■

When we write $f(n) = \Omega(g(n))$, we are saying that f is at least c times the function g except possibly when n is smaller than n_0 . Here c is some positive constant. Thus g is a lower bound (up to a constant factor c) on the value of f for all suitably large n (i.e., $n \geq n_0$). Figure 3.5 illustrates what it means for a function $g(n)$ to lower bound (up to a constant factor c) another function $f(n)$. Although $f(n)$ may be less than, equal to, or greater than $cg(n)$ for several values of n , there must exist a value m of n beyond which $f(n)$ is never less than $cg(n)$. The n_0 in the definition of omega could be any integer $\geq m$.

As in the case of the big oh notation, we normally use only simple functional forms for g .

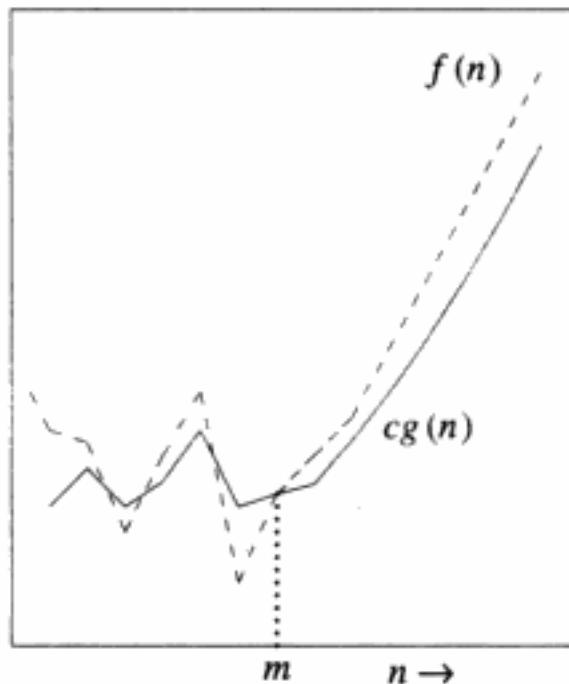


Figure 3.5 $g(n)$ is a lower bound (up to a constant factor c) on $f(n)$

Example 3.15 $f(n) = 3n + 2 > 3n$ for all n . So $f(n) = \Omega(n)$. Also, $f(n) = 3n + 3 > 3n$, and so $f(n) = \Omega(n)$. Since $f(n) = 100n + 6 > 100n$, $100n + 6 = \Omega(n)$. So $3n + 2$, $3n + 3$, and $100n + 6$ are all bounded from below by a linear function.

$f(n) = 10n^2 + 4n + 2 > 10n^2$ for $n \geq 0$. So $f(n) = \Omega(n^2)$. Similarly, $1000n^2 + 100n - 6 = \Omega(n^2)$. Furthermore, since $6 * 2^n + n^2 > 6 * 2^n$, $6 * 2^n + n^2 = \Omega(2^n)$.

Observe also that $3n + 3 = \Omega(1)$; $10n^2 + 4n + 2 = \Omega(n)$; $10n^2 + 4n + 2 = \Omega(1)$; $6 * 2^n + n^2 = \Omega(n^{100})$; $6 * 2^n + n^2 = \Omega(n^{50.2})$; $6 * 2^n + n^2 = \Omega(n^2)$; $6 * 2^n + n^2 = \Omega(n)$; and $6 * 2^n + n^2 = \Omega(1)$.

To see that $3n + 2 \neq \Omega(n^2)$, suppose that $3n + 2 = \Omega(n^2)$. Then positive c and n_0 exist such that $3n + 2 \geq cn^2$ for all $n \geq n_0$. So $cn^2/(3n + 2) \leq 1$ for all $n \geq n_0$. This relation cannot be true because its left side increases to infinity as n becomes large. ■

As in the case of the big oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. $g(n)$ is only a lower bound (up to a constant factor) on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So although we say that $3n + 3 = \Omega(n)$ and that $6 * 2^n + n^2 = \Omega(2^n)$, we almost never say that $3n + 3 = \Omega(1)$ or that $6 * 2^n + n^2 = \Omega(1)$, even though both these statements are correct.

Theorem 3.3 is the analog of Theorem 3.1 for the omega notation.

Theorem 3.3 If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof See Exercise 12. ■

Example 3.16 From Theorem 3.3, it follows that $3n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(n^2)$, and $100n^4 + 3500n^2 + 82n + 8 = \Omega(n^4)$. ■

Theorem 3.4 is the analog of Theorem 3.2, and it is usually easier to show $f(n) = \Omega(g(n))$ by using Theorem 3.4 than by using the definition of omega.

Theorem 3.4 [Omega ratio theorem] Let $f(n)$ and $g(n)$ be such that $\lim_{n \rightarrow \infty} g(n)/f(n)$ exists. $f(n) = \Omega(g(n))$ iff $\lim_{n \rightarrow \infty} g(n)/f(n) \leq c$ for some finite constant c .

Proof See Exercise 13. ■

Example 3.17 $3n + 2 = \Omega(n)$ as $\lim_{n \rightarrow \infty} n/(3n + 2) = 1/3$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $\lim_{n \rightarrow \infty} n^2/(10n^2 + 4n + 2) = 0.1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $\lim_{n \rightarrow \infty} 2^n/(6 * 2^n + n^2) = 1/6$. $6n^2 + 2 = \Omega(n)$ as $\lim_{n \rightarrow \infty} n/(6n^2 + 2) = 0$. $3n^2 + 5 \neq \Omega(n^3)$ as $\lim_{n \rightarrow \infty} n^3/(3n^2 + 5) = \infty$. ■

3.3.3 Theta Notation (Θ)

The theta notation is used when the function f can be bounded both from above and below by the same function g .

Definition 3.5 [Theta] $f(n) = \Theta(g(n))$ iff positive constants c_1 and c_2 and an n_0 exist such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all n , $n \geq n_0$. ■

When we write $f(n) = \Theta(g(n))$, we are saying that f lies between c_1 times the function g and c_2 times the function g except possibly when n is smaller than n_0 . Here c_1 and c_2 are positive constants. Thus g is both a lower and upper bound (up to a constant factor c) on the value of f for all suitably large n (i.e., $n \geq n_0$). Another way to view the theta notation is that it says $f(n)$ is both $\Omega(g(n))$ and $O(g(n))$.

Figure 3.6 illustrates what it means for a function $g(n)$ to both upper and lower bound (up to a constant factor) another function $f(n)$. There must exist a value m of n beyond which $f(n)$ lies between $c_1g(n)$ and $c_2g(n)$. The n_0 in the definition of theta could be any integer $\geq m$.

As in the case of the big oh and omega notations, we normally use only simple functional forms for g .

Example 3.18 From Examples 3.7, 3.8, 3.9, and 3.15, it follows that $3n + 2 = \Theta(n)$; $3n + 3 = \Theta(n)$; $100n + 6 = \Theta(n)$; $10n^2 + 4n + 2 = \Theta(n^2)$; $1000n^2 + 100n - 6 = \Theta(n^2)$; and $6 * 2^n + n^2 = \Theta(2^n)$.

$10 * \log_2 n + 4 = \Theta(\log_2 n)$ as $\log_2 n < 10 \log_2 n + 4 \leq 11 \log_2 n$ for $n \geq 16$. As remarked earlier, $\log_a n$ is $\log_b n$ times a constant, and we write $\Theta(\log_a n)$ simply as $\Theta(\log n)$.

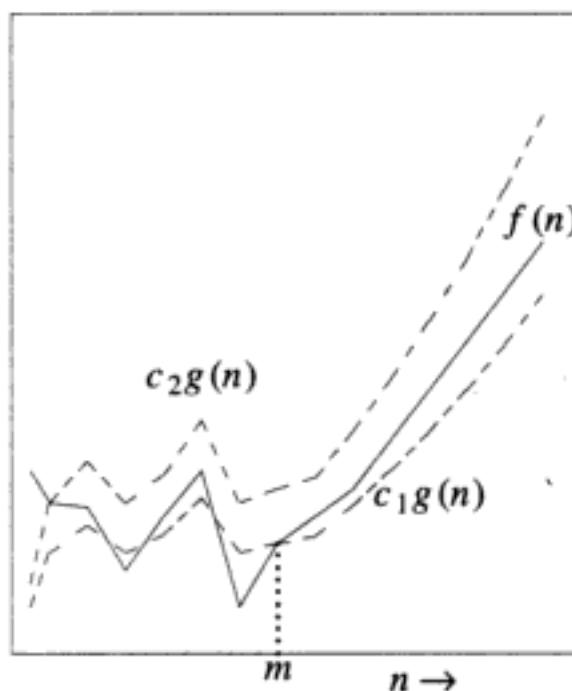


Figure 3.6 $g(n)$ is a lower and upper bound (up to a constant factor) on $f(n)$

In Example 3.12 we showed that $3n + 2 \neq O(1)$. So $3n + 2 \neq \Theta(1)$. Similarly, we may show that $3n + 3 \neq \Theta(1)$ and $100n + 6 \neq \Theta(1)$. Since $3n + 3 \neq \Omega(n^2)$, $3n + 3 \neq \Theta(n^2)$. Since $10n^2 + 4n + 2 \neq O(n)$, $10n^2 + 4n + 2 \neq \Theta(n)$. Also, since $10n^2 + 4n + 2 \neq O(1)$, it is not $\Theta(1)$.

Since $6 * 2^n + n^2$ is not $O(n^2)$, it is not $\Theta(n^2)$. Similarly, $6 * 2^n + n^2 \neq \Theta(n^{100})$; and $6 * 2^n + n^2 \neq \Theta(1)$. ■

As mentioned earlier it is common practice to use only g functions with a multiplicative factor of 1. We almost never say that $3n + 3 = O(3n)$ or $10 = O(100)$ or $10n^2 + 4n + 2 = \Omega(4 * n^2)$ or $6 * 2^n + n^2 = \Omega(6 * 2^n)$ or $6 * 2^n + n^2 = \Theta(4 * 2^n)$, even though each of these statements is true.

Theorem 3.5 If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof See Exercise 12. ■

Example 3.19 From Theorem 3.5 it follows that $3n + 2 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, and $100n^4 + 3500n^2 + 82n + 8 = \Theta(n^4)$. ■

Theorem 3.6 is the analog of Theorems 3.2 and 3.4.

Theorem 3.6 [Theta ratio theorem] Let $f(n)$ and $g(n)$ be such that $\lim_{n \rightarrow \infty} f(n)/g(n)$ and $\lim_{n \rightarrow \infty} g(n)/f(n)$ exist. $f(n) = \Theta(g(n))$ iff $\lim_{n \rightarrow \infty} f(n)/g(n) \leq c$ and $\lim_{n \rightarrow \infty} g(n)/f(n) \leq c$ for some finite constant c .

Proof See Exercise 13. ■

Example 3.20 $3n + 2 = \Theta(n)$ as $\lim_{n \rightarrow \infty} (3n + 2)/n = 3$ and $\lim_{n \rightarrow \infty} n/(3n + 2) = 1/3 < 3$; $10n^2 + 4n + 2 = \Theta(n^2)$ as $\lim_{n \rightarrow \infty} (10n^2 + 4n + 2)/n^2 = 10$; and $\lim_{n \rightarrow \infty} n^2/(10n^2 + 4n + 2) = 0.1 < 10$. $6*2^n + n^2 = \Theta(2^n)$ as $\lim_{n \rightarrow \infty} (6*2^n + n^2)/2^n = 6$ and $\lim_{n \rightarrow \infty} 2^n/(6*2^n + n^2) = 1/6 < 6$. $6n^2 + 2 \neq \Theta(n)$ as $\lim_{n \rightarrow \infty} (6n^2 + 2)/n = \infty$. ■

3.3.4 Little Oh Notation (o)

The little oh notation describes a strict upper bound on the asymptotic growth rate of the function f . Informally, $f(n)$ is little oh of $g(n)$ iff $f(n)$ is asymptotically smaller than $g(n)$ (recall that $f(n)$ is big oh of $g(n)$ iff $f(n)$ is asymptotically smaller than or equal to $g(n)$).

Definition 3.6 [Little oh] $f(n) = o(g(n))$ (read as “ f of n is little oh of g of n ”) iff $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$. ■

Example 3.21 [Little oh] $3n + 2 = o(n^2)$ as $3n + 2 = O(n^2)$ and $3n + 2 \neq \Omega(n^2)$. However, $3n + 2 \neq o(n)$. Similarly, $10n^2 + 4n + 2 = o(n^3)$, but is not $o(n^2)$. ■

The little oh notation is often used in step-count analyses. A step count of $3n + o(n)$ would mean that the step count is $3n$ plus terms that are asymptotically smaller than n . When performing such an analysis, one can ignore portions of the program that are known to contribute less than $\Theta(n)$ steps.

3.3.5 Properties

The following theorem is useful in computations involving asymptotic notation.

Theorem 3.7 These statements are true for every real number x , $x > 0$ and for every real ϵ , $\epsilon > 0$:

1. An n_0 exists such that $(\log n)^x < (\log n)^{x+\epsilon}$ for every n , $n \geq n_0$.
2. An n_0 exists such that $(\log n)^x < n^\epsilon$ for every n , $n \geq n_0$.
3. An n_0 exists such that $n^x < n^{x+\epsilon}$ for every n , $n \geq n_0$.
4. For every real y , an n_0 exists such that $n^x(\log n)^y < n^{x+\epsilon}$ for every n , $n \geq n_0$.
5. An n_0 exists such that $n^x < 2^n$ for every n , $n \geq n_0$.

Proof Follows from the definition of the individual functions. ■

Example 3.22 From Theorem 3.7 we obtain the following: $n^3 + n^2 \log n = \Theta(n^3)$; $2^n/n^2 = \Omega(n^k)$ for every natural number k ; $n^4 + n^{2.5} \log^{20} n = \Theta(n^4)$; $2^n n^4 \log^3 n + 2^n n^4 / \log n = \Theta(2^n n^4 \log^3 n)$. ■

Figure 3.7 lists some of the more useful identities involving the big oh, omega, and theta notations. In this table all symbols other than n are positive constants. Figure 3.8 lists some useful inference rules for sums and products.

	$f(n)$	Asymptotic
E1	c	$\oplus(1)$
E2	$\sum_{i=0}^k c_i n^i$	$\oplus(n^k)$
E3	$\sum_{i=1}^n i$	$\oplus(n^2)$
E4	$\sum_{i=1}^n i^2$	$\oplus(n^3)$
E5	$\sum_{i=1}^n i^k, k > 0$	$\oplus(n^{k+1})$
E6	$\sum_{i=0}^n r^i, r > 1$	$\oplus(r^n)$
E7	$n!$	$\oplus(\sqrt{n}(n/e)^n)$
E8	$\sum_{i=1}^n 1/i$	$\oplus(\log n)$

\oplus can be any one of O , Ω , and Θ

Figure 3.7 Asymptotic identities

Figures 3.7 and 3.8 prepare you to use asymptotic notation to describe the time complexity (or step count) of a program.

The definitions of O , Ω , Θ , and o can be extended to include functions of more than one variable. For example, $f(n, m) = O(g(n, m))$ iff positive constants c , n_0 , and m_0 exist such that $f(n, m) \leq cg(n, m)$ for all $n \geq n_0$ and all $m \geq m_0$.

EXERCISES

9. Show that the following equalities are correct, using the definitions of O , Ω , Θ , and o only. Do not use Theorems 3.1 through 3.6, or Figures 3.7 and 3.8.
 - (a) $5n^2 - 6n = \Theta(n^2)$.

- I1** $\{f(n) = \oplus(g(n))\} \rightarrow \sum_{n=a}^b f(n) = \oplus(\sum_{n=a}^b g(n)).$
- I2** $\{f_i(n) = \oplus(g_i(n)), 1 \leq i \leq k\} \rightarrow \sum_{i=1}^k f_i(n) = \oplus(\max_{1 \leq i \leq k} \{g_i(n)\}).$
- I3** $\{f_i(n) = \oplus(g_i(n)), 1 \leq i \leq k\} \rightarrow \prod_{i=1}^k f_i(n) = \oplus(\prod_{i=1}^k g_i(n)).$
- I4** $\{f_1(n) = O(g_1(n)), f_2(n) = \Theta(g_2(n))\} \rightarrow f_1(n) + f_2(n) = O(g_1(n) + g_2(n)).$
- I5** $\{f_1(n) = \Theta(g_1(n)), f_2(n) = \Omega(g_2(n))\} \rightarrow f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n)).$
- I6** $\{f_1(n) = O(g(n)), f_2(n) = \Theta(g(n))\} \rightarrow f_1(n) + f_2(n) = \Theta(g(n)).$

Figure 3.8 Inference rules for $\oplus \in \{O, \Omega, \Theta\}$

- (b) $n! = O(n^n).$
- (c) $2n^2 2^n + n \log n = \Theta(n^2 2^n).$
- (d) $\sum_{i=0}^n i^2 = \Theta(n^3).$
- (e) $\sum_{i=0}^n i^3 = \Theta(n^4).$
- (f) $n^{2^n} + 6 * 2^n = \Theta(n^{2^n}).$
- (g) $n^3 + 10^6 n^2 = \Theta(n^3).$
- (h) $6n^3 / (\log n + 1) = O(n^3).$
- (i) $n^{1.001} + n \log n = \Theta(n^{1.001}).$
- (j) $n^{k+\epsilon} + n^k \log n = \Theta(n^{k+\epsilon})$ for all k and ϵ , $k \geq 0$, and $\epsilon > 0$.
10. Do Exercise 9 using Theorems 3.2, 3.4, and 3.6.
11. Show that the following equalities are incorrect:
- $10n^2 + 9 = O(n).$
 - $n^2 \log n = \Theta(n^2).$
 - $n^2 / \log n = \Theta(n^2).$
 - $n^3 2^n + 6n^2 3^n = O(n^3 2^n).$
12. Prove Theorems 3.3 and 3.5.
13. Prove Theorems 3.4 and 3.6.
14. Prove that $f(n) = o(g(n))$ iff $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$.
15. Prove that equivalences E5 to E8 (Figure 3.7) are correct.

16. Prove the correctness of inference rules I1 to I6 (Figure 3.8).
17. Which of the following inferences are true? Why?
- $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$.
 - $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$.
 - $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$.
 - $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$.
 - $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$.
 - $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$.
 - $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$.
 - $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$.
 - $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$.

3.4 COMPLEXITY ANALYSIS EXAMPLES

In Section 3.2 we saw several examples in which we started with the step count of a program and then arrived at its asymptotic complexity. Actually, we can determine the asymptotic complexity quite easily without determining the exact step count. The procedure is to first determine the asymptotic complexity of each statement (or group of statements) in the program and then add up these complexities. Figures 3.9 to 3.12 determine the asymptotic complexity of several methods without performing an exact step-count analysis. These figures use the following fact that when $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then $f_1(n) + f_2(n) = \Theta(\max\{g_1(n), g_2(n)\})$.

Statement	s/e	Frequency	Total Steps
T sum(T a[], int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
T theSum = 0;	1	1	$\Theta(1)$
for (int i = 0; i < n; i++)	1	$n + 1$	$\Theta(n)$
theSum += a[i];	1	n	$\Theta(n)$
return theSum;	1	1	$\Theta(1)$
}	0	0	$\Theta(0)$

$$t_{\text{sum}}(n) = \Theta(\max\{g_i(n)\}) = \Theta(n)$$

Figure 3.9 Asymptotic complexity of `sum` (Program 1.30)

While the analyses of Figures 3.9 through 3.12 are actually carried out in terms of step counts, it is correct to interpret $t_P(n) = \Theta(g(n))$, $t_P(n) = O(g(n))$, or $t_P(n) = \Omega(g(n))$ as a statement about the computing time of program P because each step takes only $\Theta(1)$ time to execute.

Statement	s/e	Frequency	Total Steps
void transpose(T **a, int rows)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int i = 0; i < rows; i++)	1	rows + 1	$\Theta(rows)$
for (int j = i+1; j < rows; j++)	1	rows(rows + 1)/2	$\Theta(rows^2)$
swap(a[i][j], a[j][i]);	1	rows(rows - 1)/2	$\Theta(rows^2)$
}	0	0	$\Theta(0)$

$$t_{\text{transpose}}(\text{rows}) = \Theta(\text{rows}^2)$$

Figure 3.10 Asymptotic complexity of transpose (Program 2.19)

Statement	s/e	Frequency	Total Steps
void inef(T a[], T b[], int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int j = 0; j < n; j++)	1	n + 1	$\Theta(n)$
b[j] = sum(a, j + 1);	2j + 6	n	$\Theta(n^2)$
}	0	0	$\Theta(0)$

$$t_{\text{inef}}(n) = \Theta(n^2)$$

Figure 3.11 Asymptotic complexity of inef (Program 2.20)

Statement	s/e	Frequency	Total Steps
int sequentialSearch(T a[], int n, const T& x)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
int i;	1	1	$\Theta(1)$
for (i = 0; i < n && a[i] != x; i++);	1	$\Omega(1), O(n)$	$\Omega(1), O(n)$
if (i == n) return -1;	1	1	$\Theta(1)$
else return i;	1	$\Omega(0), O(1)$	$\Omega(0), O(1)$
}	0	0	$\Theta(0)$

$$t_{\text{sequentialSearch}}(n) = \Omega(1)$$

$$t_{\text{sequentialSearch}}(n) = O(n)$$

Figure 3.12 Asymptotic complexity of sequentialSearch (Program 2.1)

After you have had some experience using the table method, you will be in a position to arrive at the asymptotic complexity of a program by taking a more global approach. We elaborate on this method in the following examples.

Example 3.23 [Permutations] Consider the permutation generation code of Program 1.32. Assume that $m = n-1$. When $k = m$, the time taken is cn , where c is a constant. When $k < m$, the `else` clause is entered. At this time the `for` loop is entered

$m-k+1$ times. Each iteration of this loop takes $dt_{\text{permutations}}(k+1, m)$ time, where d is a constant. So $t_{\text{permutations}}(k, m) = d(m-k+1)t_{\text{permutations}}(k+1, m)$ when $k < m$. Using the substitution method, we obtain $t_{\text{permutations}}(0, m) = \Theta((m+1)*(m+1)!) = \Theta(n*n!)$. ■

Example 3.24 [Binary Search] Program 3.1 is a method to search a sorted array a for the element x . The STL algorithm `binary_Search` is quite similar. The variables `left` and `right` keep track of the two ends of the array segment to be searched. Initially we are to search between positions 0 and $n-1$. So `left` and `right` are, respectively, initialized to these values. We maintain the following invariant throughout:

x is one of $a[0:n-1]$ iff x is one of $a[\text{left}:\text{right}]$

```
template<class T>
int binarySearch(T a[], int n, const T& x)
{// Search a[0] <= a[1] <= ... <= a[n-1] for x.
// Return position if found; return -1 otherwise.
    int left = 0;                      // left end of segment
    int right = n - 1;                 // right end of segment
    while (left <= right) {
        int middle = (left + right)/2; // middle of segment
        if (x == a[middle]) return middle;
        if (x > a[middle]) left = middle + 1;
        else right = middle - 1;
    }
    return -1; // x not found
}
```

Program 3.1 Binary search

The search begins by comparing x with the element in the middle of the segment to be searched. If x equals this element, the search terminates. If x is smaller than this element, then we need only search the left half and so `right` is updated to `middle-1`. If x is bigger than the middle element, only the right half needs to be searched and `left` is updated to `middle+1`.

Each iteration of the `while` loop—except the last one—results in a decrease in the size of the segment of a that has to be searched by a factor of about 2. So this loop iterates $\Theta(\log n)$ times in the worst case. As each iteration takes $\Theta(1)$ time, the overall worst-case complexity is $\Theta(\log n)$. ■

Example 3.25 [Insertion Sort] Program 2.15 uses the insertion sort method to sort n elements. For each value of i , the innermost `for` loop has a worst-case complexity $\Theta(i)$. As a result, the worst-case time complexity of Program 2.15 is $\Theta(1 + 2 + 3 + \dots + n - 1) = \Theta(n^2)$. The best-case time complexity of Program 2.15 is $\Theta(n)$.

■

EXERCISE

18. Determine the asymptotic time complexity of the following methods. Set up a frequency table similar to Figures 3.9 through 3.12.
- `factorial` (Program 1.29).
 - `minmax` (Program 2.24).
 - `minmax` (Program 2.25).
 - `matrixAdd` (Program 2.21).
 - `squareMatrixMultiply` (Program 2.22).
 - `matrixMultiply` (Program 2.23).
 - `indexOfMax` (Program 1.37).
 - `polyEval` (Program 2.3).
 - `horner` (Program 2.4).
 - `rank` (Program 2.5).
 - `permutations` (Program 1.32).
 - `selectionSort` (Program 2.7).
 - `selectionSort` (Program 2.12).
 - `insertionSort` (Program 2.14).
 - `insertionSort` (Program 2.15).
 - `bubbleSort` (Program 2.9).
 - `bubbleSort` (Program 2.13).

3.5 PRACTICAL COMPLEXITIES

We have seen that the time complexity of a program is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. We can also use the complexity function to compare two programs P and Q that perform the same task. Assume that program P has complexity $\Theta(n)$ and that program Q has complexity $\Theta(n^2)$. We can assert that program P is faster than program Q is for “sufficiently

large" n . To see the validity of this assertion, observe that the actual computing time of P is bounded from above by cn for some constant c and for all n , $n \geq n_1$, while that of Q is bounded from below by dn^2 for some constant d and all n , $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, program P is faster than program Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

One should always be cautiously aware of the presence of the phrase *sufficiently large* in the assertion of the preceding discussion. When deciding which of the two programs to use, we must know whether the n we are dealing with is, in fact, sufficiently large. If program P actually runs in $10^6 n$ milliseconds while program Q runs in n^2 milliseconds and if we always have $n \leq 10^6$, then program Q is the one to use.

To get a feel for how the various functions grow with n , you should study Figures 3.13 and 3.14 very closely. These figures show that 2^n grows very rapidly with n . In fact, if a program needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately 1.1×10^{12} . On a computer performing 1,000,000,000 steps per second, this program would require about 18.3 minutes. If $n = 50$, the same program would run for about 13 days on this computer. When $n = 60$, about 310.56 years will be required to execute the program, and when $n = 100$, about 4×10^{13} years will be needed. We can conclude that the utility of programs with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 3.13 Value of various functions

Programs that have a complexity that is a high-degree polynomial are also of limited utility. For example, if a program needs n^{10} steps, then our 1,000,000,000 steps per second computer needs 10 seconds when $n = 10$; 3171 years when $n = 100$; and 3.17×10^{13} years when $n = 1000$. If the program's complexity had been n^3 steps instead, then the computer would need 1 second when $n = 1000$, 110.67 minutes when $n = 10,000$, and 11.57 days when $n = 100,000$.

Figure 3.15 gives the time that a 1,000,000,000 instructions per second computer needs to execute a program of complexity $f(n)$ instructions. One should note that currently only the fastest computers can execute about 1,000,000,000 instructions per second. From a practical standpoint, it is evident that for reasonably large n (say $n > 100$) only programs of small complexity (such as n , $n \log n$, n^2 , and n^3)

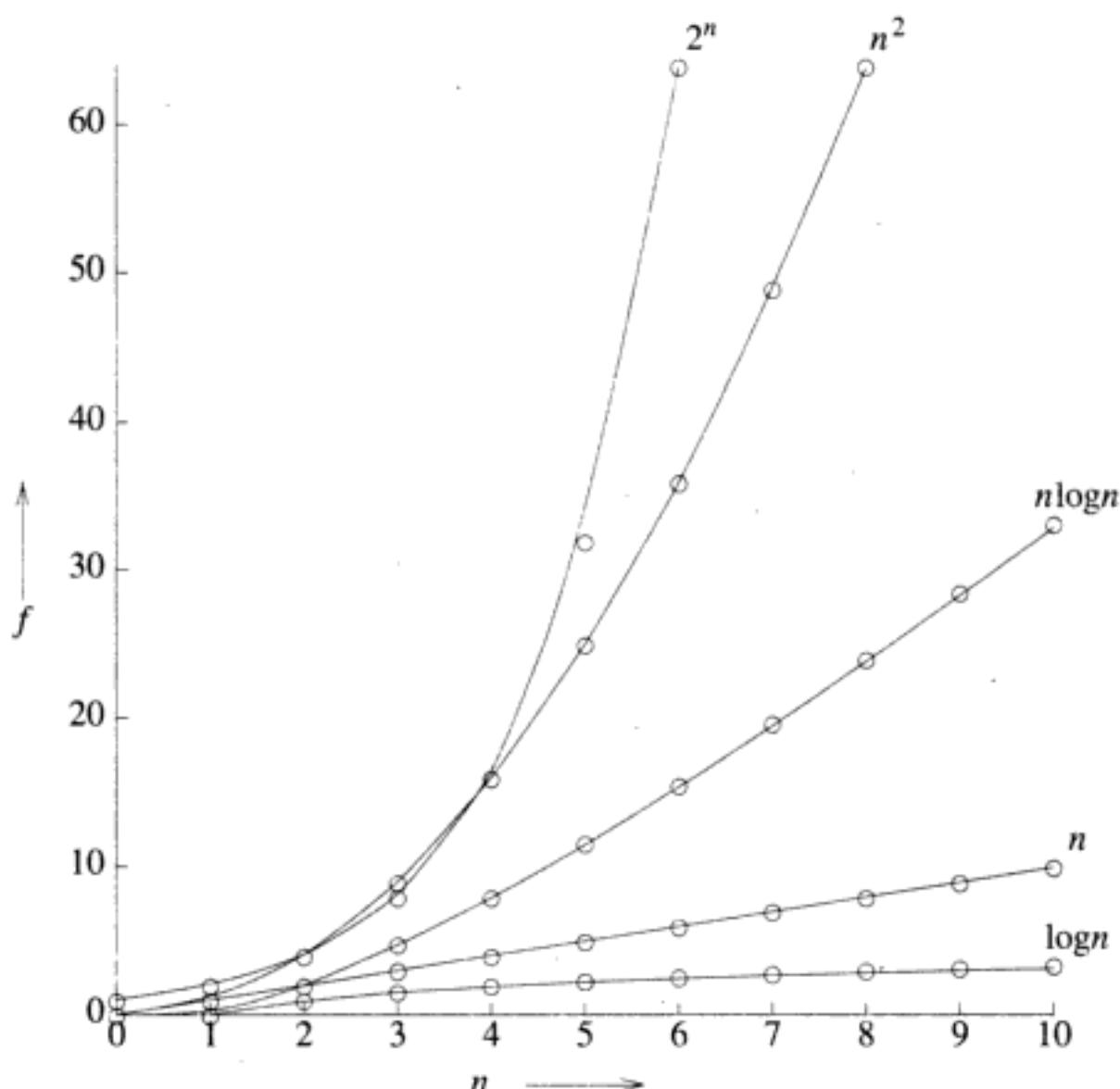


Figure 3.14 Plot of various functions

are feasible. Further, this is the case even if we could build a computer capable of executing 10^{12} instructions per second. In this case the computing times of Figure 3.15 would decrease by a factor of 1000. Now when $n = 100$, it would take 3.17 years to execute n^{10} instructions and $4 * 10^{10}$ years to execute 2^n instructions.

EXERCISES

19. Let A and B be two programs that perform the same task. Let $t_A(n)$ and $t_B(n)$, respectively, denote their run times. For each of the following pairs, find the range of n values for which program A is faster than program B .

n	$f(n)$						
	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10 s	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84 h	1 ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83 d	1 s
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56 ms	121 d	18 m
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25 ms	3.1 y	13 d
100	.10 μ s	.66 μ s	10 μ s	1 ms	100 ms	3171 y	$4 * 10^{13}$ y
10^3	1 μ s	9.96 μ s	1 ms	1 s	16.67 m	$3.17 * 10^{13}$ y	$32 * 10^{283}$ y
10^4	10 μ s	130 μ s	100 ms	16.67 m	115.7 d	$3.17 * 10^{23}$ y	
10^5	100 μ s	1.66 ms	10 s	11.57 d	3171 y	$3.17 * 10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17 * 10^7$ y	$3.17 * 10^{43}$ y	

μ s = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
 s = seconds; m = minutes; h = hours; d = days; y = years

Figure 3.15 Run times on a 1,000,000,000 instruction per second computer

- (a) $t_A(n) = 1000n$, $t_B(n) = 10n^2$.
 - (b) $t_A(n) = 2n^2$, $t_B(n) = n^3$.
 - (c) $t_A(n) = 2^n$, $t_B(n) = 100n$.
 - (d) $t_A(n) = 1000n \log_2 n$, $t_B(n) = n^2$.
20. Redo Figure 3.15 assuming a computer capable of doing 1 trillion instructions per second.
21. Suppose that using a certain program and computer, it is possible to solve problems of size up to $n = N$ in a “reasonable amount of time.” Create a table that shows the largest value of n for which solutions can be found in reasonable time using the same program and a computer that is x times as fast. Do this exercise for $x = 10, 100, 1000$, and 1,000,000 and $t_A(n) = n, n^2, n^3, n^5$, and 2^n .

3.6 REFERENCES AND SELECTED READINGS

The following books provide asymptotic analyses for several programs: *Fundamentals of Computer Algorithms* by E. Horowitz, S. Sahni, and S. Rajasekaran, W. H. Freeman and Co., New York, NY, 1998; *Introduction to Algorithms*, Second Edition, by T. Cormen, C. Leiserson, and R. Rivest, McGraw-Hill, New York, NY, 2002; and *Compared to What: An Introduction to the Analysis of Algorithms* by G. Rawlins, W. H. Freeman and Co., New York, NY, 1992.

CHAPTER 4

PERFORMANCE MEASUREMENT

BIRD'S-EYE VIEW

You can analyze and dissect all you like, but the proof of the pudding lies in the tasting. When you try to market an application code, your customer will want to know how many megabytes and seconds it's going to take to solve his/her problem on his/her computer. We can get a good handle on the memory requirements from the size of the compiled code and the size of the data space needed. The size of the data space is usually easy to figure out once you know what size instances the user is interested in solving. Determining the number of seconds the program will run requires you to actually perform experiments and measure run times. This chapter goes through the steps required to perform such an experiment.

The performance of your program depends not only on the number and type of operations you perform but also on the memory access pattern for the data and instructions in your program. Your computer has different kinds of memory—L1 cache, L2 cache, and main memory (for example)—and the time needed to access data from each is quite different. So a program with a large operation count and a small number of accesses to slow memory may take less time than a program with a small operation count and a large number of accesses to slow memory. This phenomenon is demonstrated using the matrix multiplication problem.

4.1 INTRODUCTION

Performance measurement is concerned with obtaining the actual space and time requirements of a program. As noted in earlier sections, these quantities are very dependent on the particular compiler and options used as well as on the specific computer on which the program is run. Unless otherwise stated, all performance values in this book were obtained using a 1.7 GHz Intel Pentium 4 PC with 512MB RAM and Microsoft Visual Studio .NET 2003. Time optimized code was generated using the statement

```
#pragma optimize("t", on)
```

We ignore the space and time needed for compilation because each program (after it has been fully debugged) will be compiled once and then executed several times. However, the space and time needed for compilation are important during program testing when more time may be spent on this task than in actually running the compiled code.

We do not explicitly consider measuring the run-time space requirements of a program for the following reasons:

- The size of the instruction and statically allocated data space is the size of the compiled code created by the compiler. This size may be determined using operating system commands to obtain the size of the file that contains the executable code.
- We can get a fairly accurate estimate of the recursion stack space and the space needed by dynamically allocated variables using the analytical methods of the earlier sections.

To obtain the execution (or run) time of a program, we need a clocking mechanism. In this book, we shall use the C++ function `clock()`, which measures time in ticks. The constant `CLOCKS_PER_SEC`, which is defined in the header file `time.h`, gives us the number of ticks in one second. This constant is used to convert from ticks to seconds. For our system, `CLOCKS_PER_SEC = 1000`. So, 1 tick equals 1 millisecond. Although more accurate time measurements are possible using system functions such as `QueryPerformanceCounter`, the C++ function `clock()` is adequate for our purposes.

Suppose we wish to measure the worst-case time requirements of function `insertionSort` (Program 2.15). First we need to

1. Decide on the values of n for which the times are to be obtained.
2. Determine, for each of the above values of n , the data that exhibit the worst-case behavior.

4.2 CHOOSING INSTANCE SIZE

We decide on which values of n to use according to two factors: the amount of timing we want to perform and what we expect to do with the times. Suppose we want to predict how long it will take, in the worst case, to sort an array \mathbf{a} of n objects using `insertionSort`. From Example 3.25 we know that the worst-case complexity of `insertionSort` is $\Theta(n^2)$; that is, it is quadratic in n . In theory, if we know the times for any three values of n , we can determine the quadratic function that describes the worst-case run time of `insertionSort` and we can obtain the time for all other values of n from this quadratic function. In practice, we need the times for more than three values of n for the following two reasons:

1. Asymptotic analysis tells us the behavior only for sufficiently large values of n . For smaller values of n , the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n .
2. Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve (quadratic in the case of `insertionSort`) because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, a program with asymptotic complexity $\Theta(n^2)$ can have an actual complexity that is $c_1 n^2 + c_2 n \log n + c_3 n + c_4$ —or any other function of n in which the highest order term is $c_1 n^2$ for some constant c_1 , $c_1 > 0$.

We expect the asymptotic behavior of Program 2.15 to begin for some n that is smaller than 100. So for $n > 100$ we will obtain the run time for just a few values. A reasonable choice is $n = 200, 300, 400, \dots, 1000$. There is nothing magical about this choice of values. We can just as well use $n = 500, 1000, 1500, \dots, 10,000$ or $n = 512, 1024, 2048, \dots, 2^{15}$. The latter choices will cost us more in terms of computer time and probably will not provide any better information about the run time of our method.

For n in the range $[0, 100]$, we will carry out a more refined measurement, as we aren't quite sure where the asymptotic behavior begins. Of course, if our measurements show that the quadratic behavior doesn't begin in this range, we will have to perform a more detailed measurement in the range $[100, 200]$ and so on until we detect the onset of this behavior. Times in the range $[0, 100]$ will be obtained in steps of 10 beginning at $n = 0$.

4.3 DEVELOPING THE TEST DATA

For many programs we can generate manually or by computer the data that exhibit the best- and worst-case time complexity. The average complexity, however, is usually quite difficult to demonstrate. For `insertionSort` the worst-case data for any n are a decreasing sequence such as $n, n - 1, n - 2, \dots, 1$. The best-case data

are a sorted sequence such as $0, 1, 2, \dots, n - 1$. It is difficult to envision the data that would cause `insertionSort` to exhibit its average behavior.

When we are unable to develop the data that exhibit the complexity we want to measure, we can pick the least (maximum, average) measured time from some randomly generated data as an estimate of the best (worst, average) behavior.

4.4 SETTING UP THE EXPERIMENT

Having selected the instance sizes and developed the test data, we are ready to write a program that will measure the desired run times. For our insertion sort example this program takes the form given in Program 4.1. The measured times are given in Figure 4.1.

```
int main()
{
    int a[1000], step = 10;
    double clocksPerMillis = double(CLOCKS_PER_SEC) / 1000;
                    // clock ticks per millisecond

    cout << "The worst-case time, in milliseconds, are" << endl;
    cout << "n \t Time" << endl;

    // times for n = 0, 10, 20, ..., 100, 200, 300, ..., 1000
    for (int n = 0; n <= 1000; n += step)
    {
        // initialize with worst-case data
        for (int i = 0; i < n; i++)
            a[i] = n - i;

        clock_t startTime = clock( );
        insertionSort(a, n);
        double elapsedMillis = (clock( ) - startTime) / clocksPerMillis;

        cout << n << '\t' << elapsedMillis << endl;

        if (n == 100) step = 100;
    }
    return 0;
}
```

Program 4.1 Program to obtain worst-case run times for insertion sort

n	Time	n	Time
0	0	100	0
10	0	200	0
20	0	300	0
30	0	400	0
40	0	500	0
50	0	600	0
60	0	700	0
70	0	800	15
80	0	900	0
90	0	1000	0

Times are in milliseconds

Figure 4.1 Times using Program 4.1

Figure 4.1 suggests that no time is needed to sort n elements for any of the tested values of n other than $n = 800$. Furthermore, to sort 800 numbers we need 15 milliseconds in the worst case while we can sort 1000 numbers in no time. This conclusion, of course, isn't true. The problem is that the time needed for our worst-case sorts is too small for `clock()` to measure. Although the C++ language does not specify the accuracy of `clock()`, let us assume that this function is accurate to within 100 ticks, which equals 100 milliseconds on our system. Therefore, if the method returns a time of t , the actual time lies between $\max\{0, t - 100\}$ and $t + 100$. The reported time (see Figure 4.1) for $n = 1000$ is 0 milliseconds. So the actual time could be anywhere between 0 and 100 milliseconds. If we wish our measurements to be accurate to within 10 percent, `clock() - startTime` should be at least 1000 ticks, which equals 1 second for our system. The times in Figure 4.1 do not meet this criterion.

To improve the accuracy of our measurements, we need to repeat the sort several times for each value of n . Since the sort changes the array `a`, we need to initialize this array before each sort. Program 4.2 gives the new timing program. Notice that now the measured time is the time to sort plus the time to initialize `a` and the overhead associated with the `while` loop. Figure 4.2 gives the measured times, and Figure 4.3 is a plot of these times.

We can determine the overhead associated with the `while` loop and the initialization of the array `a` by running Program 4.2 without the statement

```
insertionSort(a, n);
```

Figure 4.4 gives the output from this run for selected values of n . Subtracting the overhead time from the time per sort (Figure 4.2) gives us the worst-case time for

```
int main()
{
    int a[1000], step = 10;
    double clocksPerMillis = double(CLOCKS_PER_SEC) / 1000;
        // clock ticks per millisecond

    cout << "The worst-case time, in milliseconds, are" << endl;
    cout << "n \tRepetitions \t Total Ticks \tTime per Sort" << endl;

    // times for n = 0, 10, 20, ..., 100, 200, 300, ..., 1000
    for (int n = 0; n <= 1000; n += step)
    {
        // get time for size n
        long numberOfRepetitions = 0;
        clock_t startTime = clock( );
        do
        {
            numberOfRepetitions++;

            // initialize with worst-case data
            for (int i = 0; i < n; i++)
                a[i] = n - i;

            insertionSort(a, n);
        } while (clock( ) - startTime < 1000);
            // repeat until enough time has elapsed

        double elapsedMillis = (clock( ) - startTime) / clocksPerMillis;
        cout << n << '\t' << numberOfRepetitions << '\t' << elapsedMillis
            << '\t' << elapsedMillis / numberOfRepetitions
            << endl;

        if (n == 100) step = 100;
    }
    return 0;
}
```

Program 4.2 Program to obtain times with an accuracy of 10 percent

`insertionSort` as a function of `n`. Notice how for larger `n` the times of Figure 4.2 almost quadruple each time `n` is doubled. We expect this pattern, because the

n	Repetitions	Total Time	Time per Sort
0	6605842	1000	0.00015
10	2461486	1000	0.00041
20	1020396	1000	0.00098
30	585217	1000	0.00171
40	384720	1000	0.00260
50	262557	1000	0.00381
60	200216	1000	0.00499
70	150964	1000	0.00662
80	126457	1000	0.00791
90	99776	1000	0.01002
100	80252	1000	0.01246
200	20849	1000	0.04796
300	9527	1000	0.10497
400	5537	1000	0.18060
500	3576	1000	0.27964
600	2466	1000	0.40552
700	1870	1000	0.53476
800	1393	1000	0.71788
900	1156	1000	0.86505
1000	918	1000	1.08932

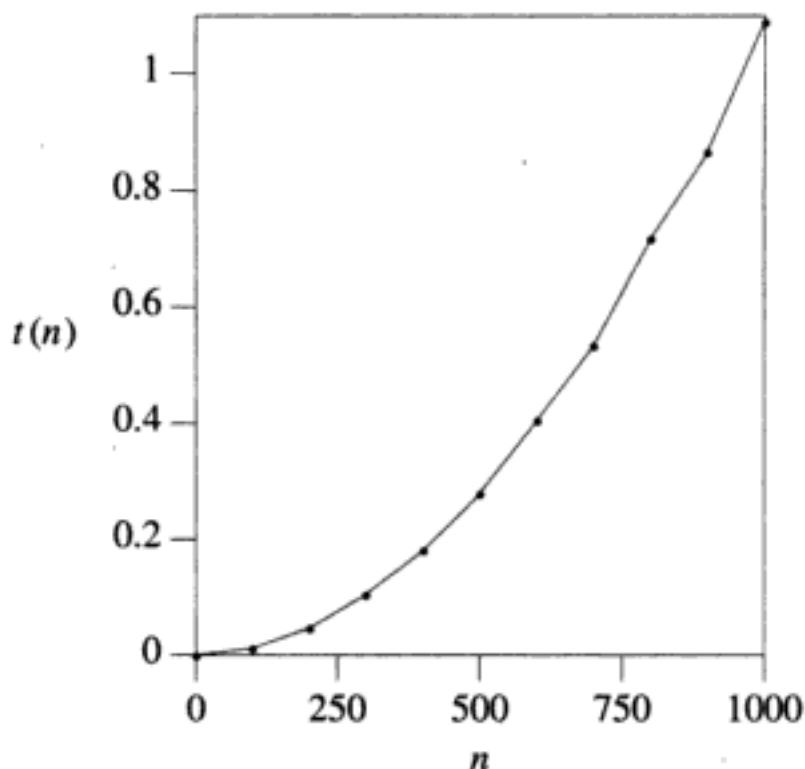
Times are in milliseconds

Figure 4.2 Output from Program 4.2

worst-case complexity is $\Theta(n^2)$.

EXERCISES

1. Why does Program 4.3 not measure run times to an accuracy of 10 percent?
2. Use Program 4.2 to obtain the worst-case run times for the two versions of insertion sort given in Programs 2.14 and 2.15. Use the same values of n as used in Program 4.2. Evaluate the relative merits of using the `insert` function versus incorporating the code for an `insert` directly into the `sort` function.
3. Use Program 4.2 to obtain the worst-case run times for the versions of bubble sort given in Programs 2.9 and 2.13. Use the same values of n as used in Program 4.2. However, you will need to verify that the worst-case data used by Program 4.2 is, in fact, worst-case data for the two bubble sort functions. Present your results as a table with three columns: n, Program 2.9, and

**Figure 4.3** Plot of worst-case insertion sort times

n	Repetitions	Total Time	Overhead
0	6588805	1000	0.00015
10	6129343	1000	0.00016
50	3014729	1000	0.00033
100	2985688	1000	0.00033
500	909538	1000	0.00110
1000	482291	1000	0.00207

Times are in milliseconds

Figure 4.4 Overhead in measurements of Figure 4.2

Program 2.13. What can you say about the worst-case performance of the two bubble sorts?

4. (a) Devise worst-case data for the two versions of selection sort given in Programs 2.7 and 2.12.
- (b) Use a suitably modified version of Program 4.2 to determine the worst-case times for the two selection sort functions. Use the same values of n

```
int main()
{
    long numberOfRepetitions = 0;
    clock_t elapsedTime = 0;
    do
    {
        numberOfRepetitions++;
        clock_t startTime = clock( );

        doSomething();

        elapsedTime += clock( ) - startTime;
    } while (elapsedTime < 1000);
    // repeat until enough time has elapsed

    cout << "Time is (in ticks) "
        << ((double) elapsedTime) / numberOfRepetitions
        << endl;
    return 0;
}
```

Program 4.3 Inaccurate way to time doSomething

- as used in Program 4.2.
- (c) Present your results as a single table with three columns: n , Program 2.7, and Program 2.12.
 - (d) What can you say about the worst-case performance of the two selection sorts?
5. This exercise compares the worst-case run times of insertion sort (Program 2.15) and the early-terminating versions of selection sort (Program 2.12) and bubble sort (Program 2.13). To level the playing field, rewrite Program 2.13 as a single function.
- (a) Devise data that show the worst-case behavior of each function.
 - (b) Using the data of (a) and the timing program of Program 4.2, obtain worst-case run times.
 - (c) Provide these times both as a single table with columns labeled n , selection sort, bubble sort, and insertion sort and as a single graph showing three curves (one for each function). The x -axis of the graph is labeled by n values, and the y -axis by time values.

- (d) What conclusions can you draw about the relative worst-case performance of the three sort functions?
 - (e) Measure the overheads for each value of n and report these in a table as in Figure 4.4. Subtract this overhead from the times obtained in (b) and present a new table of times and a new graph.
 - (f) Are there any changes to your conclusions about relative performance as a result of subtracting the overhead?
 - (g) Using the data you have obtained, estimate the worst-case time to sort 2000; 4000; and 10,000 numbers using each sort function.
6. Modify Program 4.2 so that it obtains an estimate of the average run time of `insertionSort` (Program 2.15). Do the following:
- (a) Sort a random permutation of the numbers $0, 1, \dots, n-1$ on each iteration of the `while` loop. This permutation is generated using a random permutation generator. In case you don't have such a function available, try to write one using a random number generator, or simply generate a random sequence of n numbers.
 - (b) Set the `while` loop so that at least 20 random permutations are sorted and so that at least 1 second has elapsed.
 - (c) Estimate the average sort time by dividing the elapsed time by the number of permutations sorted.
- Present the estimated average times as a table.
7. Use the strategy of Exercise 6 to estimate the average run times of the bubble sort functions given in Programs 2.9 and 2.13. Use the same values of n as in Program 4.2. Present your results as a table and as a graph.
8. Use the strategy of Exercise 6 to estimate the average run times of the selection sort functions given in Programs 2.7 and 2.12. Use the same values of n as in Program 4.2. Present your results as a table and as a graph.
9. Use the strategy of Exercise 6 to estimate and compare the average run times of the functions of Programs 2.12, 2.13, and 2.15. Use the same values of n as in Program 4.2. Present your results as a table and as a graph.
10. Devise experiments to determine the average time taken by sequential search (Program 2.1) and binary search (Program 3.1) to perform a successful search. Assume that each element of the array being searched is looked for with equal probability. Present your results as a table and as a graph.
11. Devise experiments to determine the worst-case time taken by sequential search (Program 2.1) and binary search (Program 3.1) to perform a successful search. Present your results as a table and as a graph.

12. Determine the run time of function `matrixAdd` (Program 2.21) for `rows` = 10, 20, 30, ..., 100. Present your measured times as a table and as a graph.
13. C++ has a sort function `sort(begin,end)` that may be used to sort the array `a[0:n-1]` using the invocation `sort(a, a+n)`. This sort function, which is defined in the header `algorithm`, uses a combination of the insertion sort, quick sort (Section 18.2.3) and heap sort (Section 12.6.1) methods. The complexity of the C++ function `sort` is $O(n \log n)$. Measure the time C++'s sort function takes on best- and worst-case insertion sort data. Compare these times with those for Program 2.15.
14. Determine the run time of function `transpose` (Program 2.19) for `rows` = 10, 20, 30, ..., 100. Present your measured times as a table and as a graph.
15. Determine the run time of function `squareMatrixMultiply` (Program 2.22) for `rows` = 10, 20, 30, ..., 100. Present your measured times as a table and as a graph.

4.5 YOUR CACHE AND YOU

4.5.1 A Simple Computer Model

Consider a simple computer model in which the computer's memory consists of an L1 (level 1) cache, an L2 cache, and main memory. Arithmetic and logical operations are performed by the arithmetic and logic unit (ALU) on data resident in registers (R). Figure 4.5 gives a block diagram for our simple computer model.

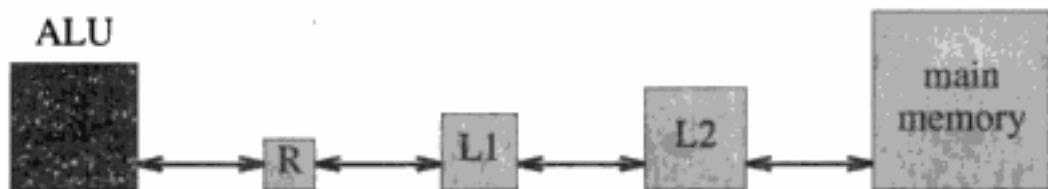


Figure 4.5 A simple computer model

Typically, the size of main memory is tens or hundreds of megabytes; L2 cache sizes are typically a fraction of a megabyte; L1 cache is usually in the tens of kilobytes; and the number of registers is between 8 and 32. When you start your program, all your data are in main memory.

To perform an arithmetic operation such as an add, in our computer model, the data to be added are first loaded from memory into registers, the data in the registers are added, and the result is written to memory.

Let one cycle be the length of time it takes to add data that are already in registers. The time needed to load data from L1 cache to a register is two cycles in our model. If the required data are not in L1 cache but are in L2 cache, we get an L1 cache miss and the required data are copied from L2 cache to L1 cache and the register in 10 cycles. When the required data are not in L2 cache either, we have an L2 cache miss and the required data are copied from main memory into L2 cache, L1 cache, and the register in 100 cycles. The write operation is counted as one cycle even when the data are written to main memory because we do not wait for the write to complete before proceeding to the next operation.

4.5.2 Effect of Cache Misses on Run Time

For our simple model, the statement `a = b + c` is compiled into the computer instructions

```
load a; load b; add; store c;
```

where the `load` operations load data into registers and the `store` operation writes the result of the `add` to memory. The `add` and the `store` together take two cycles. The two `loads` may take anywhere from 4 cycles to 200 cycles depending on whether we get no cache miss, L1 misses, or L2 misses. So the total time for the statement `a = b + c` varies from 6 cycles to 202 cycles. In practice, the variation in time is not as extreme because we can overlap the time spent on successive cache misses.

Suppose that we have two algorithms that perform the same task. The first algorithm does 2000 `adds` that require 4000 `load`, 2000 `add`, and 2000 `store` operations and the second algorithm does 1000 `adds`. The data access pattern for the first algorithm is such that 25 percent of the `loads` result in an L1 miss and another 25 percent result in an L2 miss. For our simplistic computer model, the time required by the first algorithm is $2000 * 2$ (for the 50 percent `loads` that cause no cache miss) + $1000 * 10$ (for the 25 percent `loads` that cause an L1 miss) + $1000 * 100$ (for the 25 percent `loads` that cause an L2 miss) + $2000 * 1$ (for the `adds`) + $2000 * 1$ (for the `stores`) = 118,000 cycles. If the second algorithm has 100 percent L2 misses, it will take $2000 * 100$ (L2 misses) + $1000 * 1$ (`adds`) + $1000 * 1$ (`stores`) = 202,000 cycles. So the second algorithm, which does half the work done by the first, actually takes 76 percent more time than is taken by the first algorithm.

Computers use a number of strategies (such as preloading data that will be needed in the near future into cache, and when a cache miss occurs, the needed data as well as data in some number of adjacent bytes are loaded into cache) to reduce the number of cache misses and hence reduce the run time of a program. These strategies are most effective when successive computer operations use adjacent bytes of main memory.

Although our discussion has focused on how cache is used for data, computers also use cache to reduce the time needed to access instructions.

4.5.3 Matrix Multiplication

This section is for the skeptics among you who do not believe that on a commercial computer, a program that performs more operations may actually take less time than another program that performs fewer operations. We are about to make a believer out of you.

Program 2.22 is the real program we start with. This program multiplies two square matrices that are represented as two-dimensional arrays. It performs the following computation:

$$c[i][j] = \sum_{k=1}^n a[i][k] * b[k][j], \quad 1 \leq i \leq n, \quad 1 \leq j \leq n \quad (4.1)$$

(You don't need to understand matrix multiplication to follow this demonstration. Matrix multiplication is motivated in Section 7.2.1.) Program 2.22 is a fairly standard piece of code that you can find in many books. Program 4.4 is an alternative code that produces the same two-dimensional array *c* as is produced by Program 2.22. We observe that Program 4.4 has two nested **for** loops that are not present in Program 2.22 and does more work than is done by Program 2.22 with respect to indexing into the array *c*. The remainder of the work is the same.

```
void fastSquareMatrixMultiply(int ** a, int ** b, int ** c, int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = 0;

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
}
```

Program 4.4 Less efficient way than Program 2.22 to multiply square matrices

You will notice that if you permute the order of the three nested **for** loops in Program 4.4, you do not affect the result array *c*. We refer to the loop order in Program 4.4 as *ijk* order. When we swap the second and third **for** loops, we get *ikj* order. In all, there are $3! = 6$ ways in which we can order the three nested **for** loops. All six orderings result in functions that perform exactly the same number of operations of each type. So you might think all six take the same time. Not so. By

changing the order of the loops, we change the data access pattern and so change the number of cache misses. This in turn affects the run time.

In **ijk** order, we access the elements of **a** and **c** by rows; the elements of **b** are accessed by column. Since elements in the same row are in adjacent memory and elements in the same column are far apart in memory, the accesses of **b** are likely to result in many L2 cache misses when the matrix size is too large for the three arrays to fit into L2 cache. In **ikj** order, the elements of **a**, **b**, and **c** are accessed by rows. Therefore, **ikj** order is likely to result in fewer L2 cache misses and so has the potential to take much less time than taken by **ijk** order.

Figure 4.6 gives the run time for Program 4.4 using **ijk** and **ikj** order as well as for Program 2.22. Figure 4.7 shows the normalized run times (i.e., the time taken by a method divided by the time taken by **ikj** order).

n	Program 2.22	Program 4.4	
	mult	ijk order	ikj order
500	1.5	2.6	0.9
1000	16.1	26.5	6.7
2000	719.8	844.6	54.2

Figure 4.6 Run times (in seconds) for matrix multiplication

What a surprise: **ikj** order runs much faster than both **ijk** order and Program 2.22! In fact, when $n = 500$, **ikj** order takes about 1/3rd the time taken by **ijk** order and about 1/2 that taken by Program 2.22. When $n = 1000$, these ratios are approximately 7/16 and 1/4; and when $n = 2000$, the ratios are approximately 1/13 and 1/16. Recall that **ikj** order does more work (as measured by the operation count) than is done by Program 2.22 and the same as is done by **ijk** order. Only the run time for **ikj** order grows at the $\Theta(n^3)$ rate predicted by an asymptotic analysis. The run time for **ijk** order and Program 2.22, for the tested values of n , is dominated by effects (such as cache misses) other than the operation count. Are you still skeptical?

The effect that memory hierarchy has on the performance of your code varies with the programming language, compiler, compiler options, and computer configuration. For example, when our matrix multiply codes were run on a 2.4 GHz Intel Pentium IV PC that has twice as much L2 cache as does the 1.7 GHz PC used in the experiment of Figures 4.6 and 4.7, the ratios for $n = 500$ were about 9/16 and 2/5. For $n = 1000$, the ratios were about 1/2 and 1/3; and for $n = 2000$, the ratios were about 1/4 and 1/5.

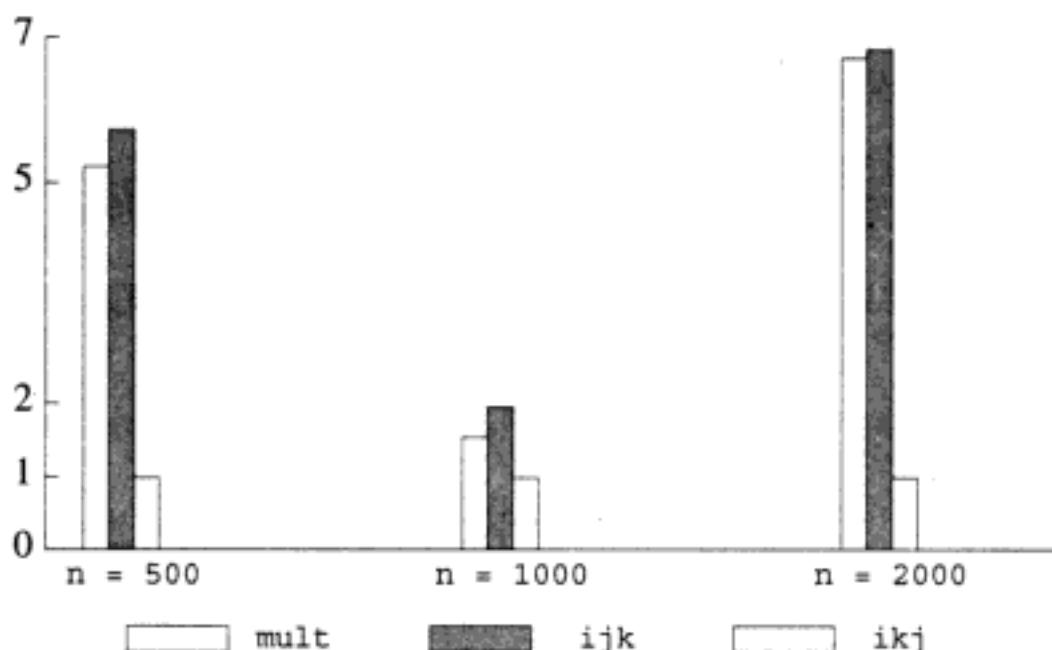


Figure 4.7 Normalized run times for matrix multiplication

EXERCISES

16. Repeat the experiment of Figure 4.6 using all six orderings of the three nested `for` loops of Program 4.4. Present your results as a table and as a bar chart.
17. In an alternative implementation of matrix multiplication, we first compute the transpose array $\text{bt}[j][k] = b[k][j]$. Equation 4.1 becomes

$$c[i][j] = \sum_{k=1}^n a[i][k] * \text{bt}[j][k], \quad 1 \leq i \leq m, \quad 1 \leq j \leq p \quad (4.2)$$

- (a) Write functions to compute the two-dimensional array c by first computing bt and then using Equation 4.2. You should have seven functions: one for each of the six permutations of the three nested `for` loops and one that corresponds to Program 2.22.
- (b) Measure the time taken by these seven functions for the cases $n = 500$, 1000, and 2000.
- (c) Present your results as a table and as a bar chart. Compare these times with those of Exercise 16.

18. Write a function to transpose an $n \times n$ array by blocks. That is, imagine that the array is partitioned into $k \times k$ subarrays (or blocks) and transpose one subarray at a time. Measure the run time of your transpose code for large n ; use $k = 2, 4, 8, 16, 32$, and 64 ; assume that n is a power of 2 . How does the performance of your code compare with that of the transpose code of Program 2.19. Can you explain the relative performance?

4.6 REFERENCES AND SELECTED READINGS

To learn more about how a cache works, see *Computer Organization and Design*, by J. Hennessy and D. Patterson, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998, Chapter 7.

CHAPTER 5

LINEAR LISTS—ARRAY REPRESENTATION

BIRD'S-EYE VIEW

We are now ready to begin the study of data structures, which continues through Chapter 16 of this book. Although Chapters 5 and 6 focus on the data structure *linear list*, their primary purpose is to introduce the different ways in which data may be represented or stored in a computer's memory as well as on a disk. In succeeding chapters we study the representation of other popular data structures such as matrices, stacks, queues, dictionaries, priority queues, tournament trees, search trees, and graphs.

The common data representation methods used by C++ programs are array based and linked (or pointer based). The data structure *linear list* is used to illustrate these representation methods. The current chapter develops the array representation of a linear list and Chapter 6 develops the linked representation of a linear list.

The STL's containers—`vector` and `list`—are roughly equivalent to our array and (doubly) linked representations of a linear list; the STL classes have many additional methods though. In our development of the array and linked representations of a linear list, we have used the same function/method names and similar signatures as used by the STL implementations. This approach will enable you to switch easily to the STL implementations.

In an array representation, elements are stored in an array; a mathematical formula is used to determine where to store each element. This formula gives the array index where the element resides. In the simplest cases the formula stores successive elements of a list in successive memory locations, and we obtain what is commonly known as the sequential representation of a list.

The data structure concepts introduced in this chapter are

- Abstract data types and their specification as C++ abstract classes.
- Linear lists.
- Changing array length (i.e., number of positions in the array) and array doubling.
- Array representation.
- Data structure iterators.

The C++ concepts developed in this chapter are

- Abstract classes.
- Iterators.

No new array applications are introduced in this chapter because several array applications were developed in Chapters 1 through 3.

5.1 DATA OBJECTS AND STRUCTURES

A **data object** is a set of *instances* or *values*. Some examples are

1. *boolean* = {*false*, *true*}
2. *digit* = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
3. *letter* = {A, B, C, ..., Z, a, b, ..., z}
4. *naturalNumber* = {0, 1, 2, ...}
5. *integer* = {0, ±1, ±2, ±3, ...}
6. *string* = {a, b, ..., aa, ab, ac, ...}

boolean, *digit*, *letter*, *naturalNumber*, *integer*, and *string* are data objects. *true* and *false* are the instances of *boolean*, while 0, 1, ..., and 9 are the instances of *digit*. We may regard the individual instances of a data object as being either **primitive** (or **atomic**) or composed of instances of another (possibly the same) data object. In the latter case we use the term **element** to refer to the individual components of an instance of an object.

For example, each instance of the data object *NaturalNumber* can be regarded as atomic. In this case we are not concerned with a further decomposition of the instances of this data object. Another view is to regard each instance of a *naturalNumber* as being composed of several instances of the data object *digit*. In this view the number 675 comprises the digits 6, 7, and 5 (in that order).

The data object *string* is the set of all possible string instances. Each instance of a string is composed of characters. Some examples of instances are *good*, *a trip to Hawaii, going down hill*, and *abcabcdabcde*. The first string has the four elements *g*, *o*, *o*, and *d* (in that order). Each element is an instance of the data object *Letter*.

The instances of a data object as well as the elements that constitute individual instances are generally related in some way. For example, the natural number 0 is the smallest natural number; 1 is the next; and 2 is the next. In the natural number 675, the most significant digit is 6, the next is 7, and 5 is the least significant digit. In the string *good*, *g* is the first letter, *o* the second and third, and *d* the last.

In addition to interrelationships, a set of operations (or functions) is generally associated with any data object. These operations may transform one instance of an object into another instance of that object, or into an instance of another data object, or do both these transformations. The operation could simply create a new instance without transforming the instances from which the new one is created. For example, the operation *add* defined on the natural numbers creates a new natural number that is the sum of the two numbers to be added; the two numbers that get added are unaltered.

A **data structure** is a data object together with the relationships that exist among the instances and among the individual elements that compose an instance. These relationships are provided by specifying the operations of interest.

When we study data structures, we are concerned with the representation of data objects (actually of the instances) as well as the implementation of the operations of interest for the data objects. The representation of each data object should facilitate an efficient¹ implementation of the operations.

The most frequently used data objects together with their frequently used operations are already implemented in C++ as primitive data types. The data objects *integer* (*int*) and *boolean* (*bool*), defined above, fall into this category. All other data objects can be represented using the primitive data types and the grouping ability provided by the class, array, and pointer features of C++. Many of the data objects we shall study in this text (for example, linear list, stack, queue, and priority queue) have been implemented as classes in the STL.

5.2 THE LINEAR LIST DATA STRUCTURE

Each instance of the data structure **linear list** (or **ordered list**) is an ordered collection of elements. Each instance is of the form $(e_0, e_1, \dots, e_{n-1})$ where n is a finite natural number; the e_i items are the elements of the list; the **index** of e_i is i ; and n is the list **length** or **size**. The elements may be viewed as atomic, as their individual structure is not relevant to the structure of the list. When $n = 0$, the list is **empty**. When $n > 0$, e_0 is the **zeroth** (or **front**) element and e_{n-1} is the **last** element of the list. We say that e_0 **comes before** (or precedes) e_1 , e_1 comes before e_2 , and so on. Other than this precedence relation, no other structure exists in a linear list.

Some examples of linear lists are (1) an alphabetized (i.e., ordered by name) list of students in a class; (2) a list of exam scores in nondecreasing order; (3) an alphabetized list of members of Congress; and (4) a list of gold-medal winners in the Olympics men's basketball event ordered by year. With these examples in mind, we see the need to perform the following operations on a linear list:

- Create a linear list.
- Destroy a linear list.
- Determine whether the list is empty.
- Determine the size of the list.
- Find the element with a given index.
- Find the index of a given element.
- Delete, erase or remove an element given its index.
- Insert a new element so that it has a given index.

¹The term *efficient* is used here in a very liberal sense. It includes performance efficiency as well as measures of the complexity of development and maintenance of associated software.

- Output the list elements in order, left to right.

5.2.1 The Abstract Data Type *linearList*

A linear list may be specified as an **abstract data type** (ADT) in which we provide a specification of the instances as well as of the operations that are to be performed (see ADT 5.1). The ADT specification is independent of any representation and programming language we have in mind. All representations of the ADT must satisfy the specification, and the specification becomes a way to validate the representation. In addition, all representations that satisfy the specification may be used interchangeably in applications of the data type. In ADT 5.1 we have omitted specifying operations to create and destroy instances of the data type. All ADT specifications implicitly include an operation to create an empty instance and, optionally, an operation to destroy an instance.

AbstractDataType *linearList*

{

instances

ordered finite collections of zero or more elements

operations

empty() : return **true** if the list is empty, **false** otherwise

size() : return the list size (i.e., number of elements in the list)

get(index): return the *index*th element of the list

indexOf(x): return the index of the first occurrence of *x* in the list,
return -1 if *x* is not in the list

erase(index): remove/delete the *index*th element, elements with higher index have their index reduced by 1

insert(index, x): insert *x* as the *index*th element, elements with index $\geq index$ have their index increased by 1

output(): output the list elements from left to right

}

ADT 5.1 Abstract data type specification of a linear list

5.2.2 The Abstract Class `LinearList`

C++ supports two types of classes—abstract and concrete. An abstract class is a class that contains a member function for which no implementation has been specified. Such a function, called a *pure virtual function*, is specified using the zero initializer as in

```
virtual int myPureVirtualFunction(int x) = 0;
```

A concrete class contains no pure virtual function. Only concrete classes may be instantiated. That is, we may create an instance or object only of a concrete class. However, we can create pointers to objects of an abstract class.

Rather than use the informal English approach to specify an ADT as in ADT 5.1, we may use a C++ abstract class as in Program 5.1.

```
template<class T>
class linearList
{
public:
    virtual ~linearList() {}
    virtual bool empty() const = 0;
        // return true iff list is empty
    virtual int size() const = 0;
        // return number of elements in list
    virtual T& get(int theIndex) const = 0;
        // return element whose index is theIndex
    virtual int indexOf(const T& theElement) const = 0;
        // return index of first occurrence of theElement
    virtual void erase(int theIndex) = 0;
        // remove the element whose index is theIndex
    virtual void insert(int theIndex, const T& theElement) = 0;
        // insert theElement so that its index is theIndex
    virtual void output(ostream& out) const = 0;
        // insert list into stream out
};
```

Program 5.1 Abstract class specification of a linear list

Although the specification of Program 5.1 is quite similar to that of ADT 5.1, this specification is programming-language dependent. In particular, many of the keywords we have used are defined only in C++. Since a class that derives from or extends an abstract class is itself abstract (and so cannot be instantiated) unless it provides an implementation for all pure virtual functions of the base class, by

requiring that every ADT implementation be derived from the abstract class for that ADT, we ensure a complete and consistent (i.e., with the same public functions) implementation of the ADT.

We provide a virtual destructor for our abstract class so that when a reference to a linear list goes out of scope, the default destructor for linear list isn't invoked; rather the destructor for the true data type of the referenced object is invoked.

EXERCISE

1. Let $L = (a, b, c, d)$ be a linear list. What is the result of each of the following operations?
 - (a) `empty()`
 - (b) `size()`
 - (c) `get(0), get(2), get(6), get(-3)`
 - (d) `indexOf(a), indexOf(c), indexOf(q)`
 - (e) `erase(0), erase(2), erase(3)`
 - (f) `insert(0, e), insert(2, f), insert(3, g), insert(4, h), insert(6, h), insert(-3, h)`

5.3 ARRAY REPRESENTATION

5.3.1 The Representation

In an **array representation**, we use an array to store the list elements. Although we can pack several list instances into a single array (see Section 5.5), it is easier to use a different array for each instance. Individual elements of an instance are located in the array using a mathematical formula.

Suppose we decide to use a one-dimensional **element** to store the elements of a linear list. The array **element** has positions (or locations) `element[0] ... element[arrayLength-1]`, where `arrayLength` is the length or capacity of the array. Each array position can be used to store a single list element. We need to map the elements of the list to positions in the array. Where does the zeroth element reside? Where does the last element reside? The most natural mapping uses the formula

$$\text{location}(i) = i \tag{5.1}$$

Equation 5.1 states that the i th element of the list (if it exists) is stored in position i of the array. Figure 5.1(a) shows how the list `[5, 2, 4, 8, 1]` is stored in the array **element** using the mapping of Equation 5.1. The length of the array is 10, and the size of the list is 5.

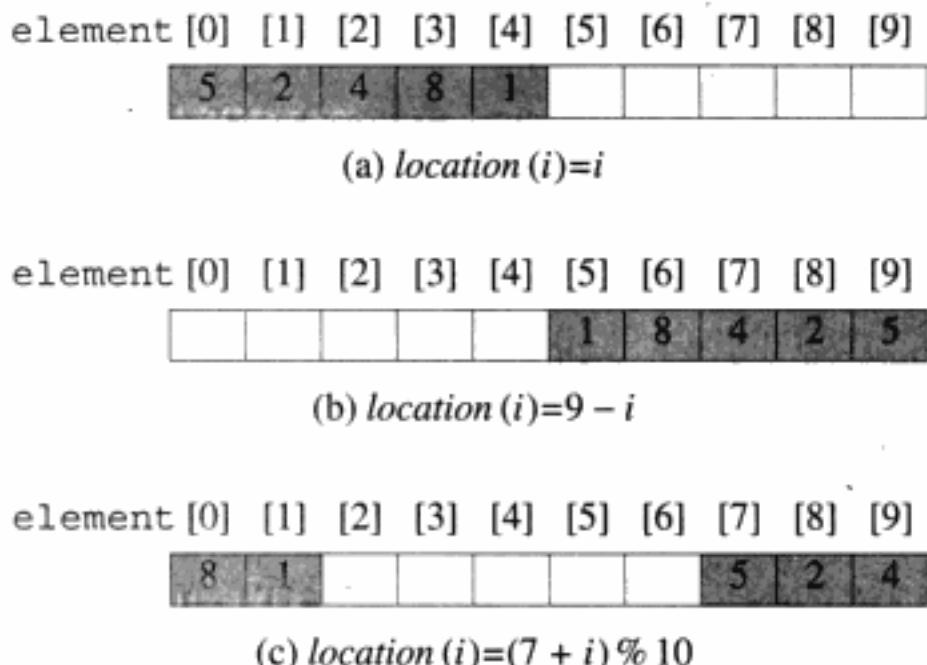


Figure 5.1 Different ways to map $[5, 2, 4, 8, 1]$ into a one-dimensional array

Although Equation 5.1 is a natural choice for a formula to map list elements into array positions, other choices are possible. For example, the formula

$$location(i) = \text{arrayLength} - i - 1 \quad (5.2)$$

stores the list elements backwards beginning at the right end of the array `element`, and the formula

$$location(i) = (location(0) + i)\%arrayLength \quad (5.3)$$

stores elements beginning at any position in the array and wraps around to the front of the array, if necessary, to store the remaining elements. Figure 5.1(b) shows how the list $[5, 2, 4, 8, 1]$ is stored when Equation 5.2 is used, and Figure 5.1(c) shows how this list is stored using Equation 5.3 and $location(0) = 7$. Equation 5.3 is used in Chapter 9 to map a queue into a one-dimensional array.

In our array representation of a linear list, we use a one-dimensional array `element` that holds the list elements as per Equation 5.1, a variable `listSize` that keeps track of the number of elements currently in the list, and a variable `arrayLength` that keeps track of the capacity of the array `element`. We may remove element e_i from the list by moving elements to its right down by 1. For example, to remove the element $e_1 = 2$ from the list of Figure 5.1(a), we have to move the elements $e_2 = 4$, $e_3 = 8$, and $e_4 = 1$, which are to the right of e_1 , to positions 1, 2,

and 3 of the array `element`. Figure 5.2(a) shows the result. The shaded elements were moved.

element [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
5	4	8	1						

(a) 2 removed from `element [1]`, `listSize = 4`

element [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
5	4	7	8	1					

(b) 7 inserted at `element [2]`, `listSize = 5`

Figure 5.2 Removing and inserting an element

To insert an element so that it becomes element i of a list, we must first move the existing element e_i (if any) and all elements to its right one position right and then put the new element into position i of the array. For example, to insert 7 as the second element of the list of Figure 5.2(a), we first move elements e_2 and e_3 to the right by 1 and then put 7 into position 2 of the array. Figure 5.2(b) shows the result. The shaded elements were moved.

Before we can write an array class that implements the ADT *linearList*, we must decide on the data type of the array `element` and the length of this array. By making our linear list class a template class, we avoid having to make the first decision. For the second decision, we note that the array `element` must be large enough to hold the maximum number of elements that might be in the list at any time. This maximum number is often difficult to estimate. To overcome this hurdle, we can ask the user to provide an estimate and then dynamically increase the length of the array `element` in case the user underestimated.

5.3.2 Changing the Length of a One-Dimensional Array

To increase or decrease the length of a one-dimensional array `a` that contains elements in positions `a[0:n-1]`, we first define an array of the new length, then copy the `n` elements from `a` to the new array, and finally change the value of `a` so that it references the new array. Program 5.2 gives the method `changeLength1D`, which performs these three tasks.

It takes $\Theta(1)$ time to create an array of length m . Notice that the invocation of `new` to create the new array may cause a `bad_alloc` exception to be thrown. If `new` is successful in creating the new array, Program 5.2 spends $\Theta(n)$ time copying elements from the source array into the destination array. Therefore, the complexity of Program 5.2 is $O(n) = O(n)$.

```
template<class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
    if (newLength < 0)
        throw illegalParameterValue("new length must be >= 0");

    T* temp = new T[newLength];           // new array
    int number = min(oldLength, newLength); // number to copy
    copy(a, a + number, temp);
    delete [] a;                         // deallocate old memory
    a = temp;
}
```

Program 5.2 Changing the length of a one-dimensional array

When an array is used to represent a data structure whose size increases dynamically, the array length is often doubled whenever the array becomes full. This process is referred to as **array doubling**. When array doubling is used, the total time spent changing the array length is (in an asymptotic sense) no more than the total time spent inserting elements into the data structure (see Theorem 5.1).

5.3.3 The Class `arrayList`

Class Definition for `arrayList`

We define a C++ class `arrayList` that implements the ADT *linearList* using Equation 5.1. Program 5.3 gives the class header, the data members, and the function/method prototypes. Since `arrayList` is to be a concrete class that extends the abstract class `linearList`, it must provide an implementation of all the methods of the abstract class `linearList`. The class `arrayList` may, however, contain methods that are not declared in `linearList`. Our class, for example, contains the methods `capacity` and `checkIndex` that are not declared in `linearList`. The method `capacity`, gives the current length of the array `element` while the method `checkIndex` verifies that a specified element index is in the permissible range 0 through `listSize - 1`.

Constructor and Copy Constructor for `arrayList`

Program 5.4 gives the class constructor as well as the copy constructor. The constructor creates an array whose length is `initialCapacity`; the default value for this length is 10. Also, it sets `arrayLength` to `initialCapacity` and `listSize` to 0. The copy constructor makes a copy or clone of an object. This constructor is invoked, for example, when an object is passed by value to a function as well as

```

template<class T>
class arrayList : public linearList<T>
{
public:
    // constructor, copy constructor and destructor
    arrayList(int initialCapacity = 10);
    arrayList(const arrayList<T>&);
    ~arrayList() {delete [] element;}

    // ADT methods
    bool empty() const {return listSize == 0;}
    int size() const {return listSize;}
    T& get(int theIndex) const;
    int indexOf(const T& theElement) const;
    void erase(int theIndex);
    void insert(int theIndex, const T& theElement);
    void output(ostream& out) const;

    // additional method
    int capacity() const {return arrayLength;}

protected:
    void checkIndex(int theIndex) const;
        // throw illegalIndex if theIndex invalid
    T* element;           // 1D array to hold list elements
    int arrayLength;      // capacity of the 1D array
    int listSize;         // number of elements in list
};

```

Program 5.3 Class definition for arrayList

when a function returns an object. Our code for this constructor employs the STL algorithm `copy` (see Section 1.8).

Program 5.4 also gives the code for the `empty`, `size`, and `capacity` functions. If we assume that the complexity of the `new` operator is $O(1)$, we see that the complexity of the constructor is $O(1)$ when `T` is a primitive data type of C++ (e.g., `int`, `float`, and so on). When `T` is a user-defined data type, the complexity of the constructor is $O(\text{initialCapacity})$. This is so because when the array `element` is created, the constructor for the user-defined data type `T` is invoked for each position of the array. The complexity of `empty`, `size`, and `capacity` methods is $O(1)$ and that of the copy constructor is $O(n)$, where n is the size of the list that is to be

```
template<class T>
arrayList<T>::arrayList(int initialCapacity)
{// Constructor.
    if (initialCapacity < 1)
        {ostringstream s;
         s << "Initial capacity = " << initialCapacity << " Must be > 0";
         throw illegalParameterValue(s.str());
        }
    arrayLength = initialCapacity;
    element = new T[arrayLength];
    listSize = 0;
}

template<class T>
arrayList<T>::arrayList(const arrayList<T>& theList)
{// Copy constructor.
    arrayLength = theList.arrayLength;
    listSize = theList.listSize;
    element = new T[arrayLength];
    copy(theList.element, theList.element + listSize, element);
}
```

Program 5.4 Constructors for arrayList

copied.

Instantiating arrayList

Linear lists that are represented as arrays may be created/instantiated using statements similar to those given below.

```
// create two linear lists with initial capacity 100
linearList *x = (linearList) new arrayList<int>(100);
arrayList<double> y(100);

// create a linear list with the default initial capacity
arrayList<char> z;

// create a linear list that is a copy of the list y
arrayList<double> w(y);
```

Elementary Methods of arrayList

Program 5.5 gives the implementation of the `checkIndex`, `get` and `indexOf` methods. The code for the `indexOf` method uses the STL function `find` that searches a range for the first occurrence of a matching element.

```

template<class T>
void arrayList<T>::checkIndex(int theIndex) const
{// Verify that theIndex is between 0 and listSize - 1.
    if (theIndex < 0 || theIndex >= listSize)
    {ostringstream s;
        s << "index = " << theIndex << " size = " << listSize;
        throw illegalIndex(s.str());
    }
}

template<class T>
T& arrayList<T>::get(int theIndex) const
{// Return element whose index is theIndex.
// Throw illegalIndex exception if no such element.
    checkIndex(theIndex);
    return element[theIndex];
}

template<class T>
int arrayList<T>::indexOf(const T& theElement) const
{// Return index of first occurrence of theElement.
// Return -1 if theElement not in list.

    // search for theElement
    int theIndex = (int) (find(element, element + listSize, theElement)
                           - element);

    // check if theElement was found
    if (theIndex == listSize)
        // not found
        return -1;
    else return theIndex;
}

```

Program 5.5 `checkIndex`, `get` and `indexOf`

The complexity of `checkIndex` and `get` is $\Theta(1)$ and that of `indexOf` is $O(\max\{\text{listSize}, 1\})$. For simplicity, we will often write complexities of this latter form as $O(\text{listSize})$.

Removing an Element

To remove or delete the `theIndex`th element from a list, we need to first ascertain that the list contains an element with this index and then delete this element. If the list does not have a `theIndex`th element, an exception occurs because the ADT `linearList` (ADT 5.1) doesn't tell us what to do at this time. Therefore, we throw an exception of type `illegalIndex`.

When there is a `theIndex`th element, we can perform the deletion by using the `copy` algorithm to move elements `theIndex+1`, `theIndex+2`, ..., `listSize-1` down (left) one position and reducing the value of `listSize` by 1. Function `erase` (Program 5.6) implements the delete/remove operation.

```
template<class T>
void arrayList<T>::erase(int theIndex)
{// Delete the element whose index is theIndex.
// Throw illegalIndex exception if no such element.
    checkIndex(theIndex);

    // valid index, shift elements with higher index
    copy(element + theIndex + 1, element + listSize,
         element + theIndex);

    element[--listSize].~T(); // invoke destructor
}
```

Program 5.6 Remove the `theIndex`th element

When there is no `theIndex`th element, an exception is thrown and the time taken by `erase` is $\Theta(1)$. When the list has a `theIndex`th element, `listSize-theIndex` elements are moved, taking $\Theta(\text{listSize}-\text{theIndex})$ time (assuming each element move takes $O(1)$ time). Hence the overall complexity is $O(\text{listSize}-\text{theIndex})$.

Inserting an Element

To insert a new element as the `theIndex`th element in the list, we need to first move elements `theIndex` through `listSize-1` one position up (right), then insert the new element in position `theIndex`, and finally increment `listSize` by 1. This upward move of elements is accomplished using the `copy_backward` STL function rather than the `copy` function. The `copy_backward` function moves elements beginning

with the rightmost one that is to be moved. Program 5.7 gives the complete C++ code to insert an element. As you can see, the method doubles the length of the array `element` in case the array has no space to accommodate the new element that is to be inserted.

```
template<class T>
void arrayList<T>::insert(int theIndex, const T& theElement)
{// Insert theElement so that its index is theIndex.
    if (theIndex < 0 || theIndex > listSize)
        // invalid index
        ostringstream s;
        s << "index = " << theIndex << " size = " << listSize;
        throw illegalIndex(s.str());
}

// valid index, make sure we have space
if (listSize == arrayLength)
    // no space, double capacity
    changeLength1D(element, arrayLength, 2 * arrayLength);
    arrayLength *= 2;
}

// shift elements right one position
copy_backward(element + theIndex, element + listSize,
              element + listSize + 1);

element[theIndex] = theElement;

listSize++;
}
```

Program 5.7 Insert theElement as theIndexth element

It takes $\Theta(1)$ time to determine whether an exception is to be thrown, $\Theta(\text{arrayLength}) = \Theta(\text{listSize})$ time to double the array length if this doubling is necessary, and $\Theta(\text{listSize}-\text{theIndex})$ time to shift elements. Therefore, the total time taken by `insert` is $O(\text{listSize})$.

Why do we double the array length in Program 5.7 and not simply increase the length by 1 or 2 (say)? Although increasing the array length by 1 or 2 every time does not affect the worst-case complexity of an insert (this worst-case complexity remains $\Theta(\text{listSize})$), increasing array length in this way can affect the asymptotic complexity of a sequence of inserts. Suppose we start with an empty list with initial

capacity 1 and perform $n = 2^k + 1$ inserts. Assume that the inserts are performed at the end of the list. Therefore, no insert requires a shift of previously inserted elements and the time required to make the n inserts is $\Theta(n)$ plus the time spent increasing the array length. When the array length is always increased by 1, the time spent increasing the array length is $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$. Therefore, the total time needed for the n inserts is $\Theta(n^2)$.

If we double the array length as is done in Program 5.7, the total time spent changing the array length is $\Theta(\sum_{i=0}^k 2^i) = \Theta(2^{k+1} - 1) = \Theta(n)$. Therefore, the complexity of the n inserts is $\Theta(n)$. In fact, a simple generalization of this analysis shows that when the array length is always increased by a multiplicative factor (from `arrayLength` to `c*arrayLength`, where $c > 1$ is a constant), the total time spent increasing the array length is $O(\text{number of inserts})$ even if `erase` and other operations are mixed in with the insert operations. This analysis leads to Theorem 5.1.

Theorem 5.1 *If we always increase the array length by a constant factor (which is 2 in Program 5.7), the time spent on any sequence of linear list operations increases by at most a constant factor when compared to the time taken for the same set of operations under the assumption that the initial capacity is not an underestimate (note that when this assumption is valid, no time is spent increasing the array length).*

The Function `output` and Overloading `<<`

Program 5.8 gives the code for `output`. The complexity of this code is $O(\text{listSize})$ under the assumption that the time to needed to insert a single element into the output stream is $O(1)$. Program 5.8 also gives the code to overload the stream insertion operator `<<`.

```
template<class T>
void arrayList<T>::output(ostream& out) const
{// Put the list into the stream out.
    copy(element, element + listSize, ostream_iterator<T>(cout, " "));
}

// overload <<
template <class T>
ostream& operator<<(ostream& out, const arrayList<T>& x)
{x.output(out); return out;}
```

Program 5.8 Inserting a linear list into an output stream

Decreasing the Length of element

Although our implementation of a linear list increases the length of `element` as needed, it never reduces its length. Therefore, an array linear list whose element array has grown to a length of 1,000,000 (say) will hold on to this much array space until the linear list is destroyed. This status continues even though the list may never again have more than 10 elements in it.

To enable the linear list to free some of the array space when the list size becomes small, we can modify the method `erase` so that it reduces the array length to $\max\{\text{initialCapacity}, \text{length}/2\}$ whenever `size < length/4` (say). This strategy is considered in Exercise 20.

Using the Class `arrayList`

A sample `main` method and the generated output can be found on the Web site for this book.

5.3.4 Iterators in C++

An **iterator** is a pointer to an element of an object (for example, a pointer to an element of an array). As the name suggests, an iterator permits you to go (or iterate) through the elements of the object one by one. Program 5.9 shows how to use a pointer `y` to an array element to iterate through the array's elements. The datatype of the pointer `y` is `int*`, which indicates that `y` points to elements of type `int`. In the `for` loop header, `y` is initialized to point to the first element in the array `x[]` (technically, the variable `x` is a pointer to the first element of the array). The expression `y++` increments the pointer so that it advances to the next element of the array. Similarly, `x + 3` is a pointer 3 positions from `x`; that is, it points one position past the last element `x[2]` of the array. So in the `for` loop of Program 5.9 the pointer (or iterator) `y` iterates through elements in the range $[x, x + 3]$. The expression `*y` dereferences the pointer `y` so as to get the element pointed to by `y`. The program outputs `x[0:2]`.

The following code is equivalent to the `for` loop of Program 5.9.

```
for (int i = 0; i != 3; i++)
    cout << x[i] << " ";
```

Although you may find this code more transparent than that of Program 5.9, the code of Program 5.9 is generalized easily to output the elements of any object for which an iterator is defined. The code

```
for (iterator i = start; i != end; i++)
    cout << *i << " ";
```

outputs all elements in the range $[\text{start}, \text{end}]$. In this code `iterator` is the datatype of the iterator, `start` is the iterator value for the first element in the range and

```
int main()
{
    int x[3] = {0, 1, 2};

    // use a pointer y to iterate through the array x
    for (int* y = x; y != x + 3; y++)
        cout << *y << " ";
    cout << endl;
    return 0;
}
```

Program 5.9 Using an array iterator

`end` is the value the iterator has when incremented one past the last element to be output.

The concept of an iterator is fundamental to writing generic code in C++. Program 5.10 gives a possible code for the STL `copy` function, for example. This code may be used to copy elements of any object that has an iterator for which the operators `!=`, `*`, and `++` (postincrement) as well as the dereferenced assignment (`*to =`) are defined. Different generic codes we write require our iterator to have different capabilities. For example, the `copy_backward` function requires us to decrement the value of the iterator.

```
template <class iterator>
void copy(iterator start, iterator end, iterator to)
{// copy from [start, end) to [to, to + end - start)
    while (start != end)
        {*to = *start; start++; to++;}
}
```

Program 5.10 Possible code for STL `copy` function

To simplify iterator development and categorization of generic iterator-based codes, the C++ STL defines five categories of iterators: input, output, forward, bidirectional and random access. All iterators support the equality operators `==` and `!=` as well as the dereference operator `*`. Input iterators additionally provide read access to the elements pointed at and support the pre- and post-increment operator `++`. Output iterators provide write access to the elements and also permit iterator advancement via the `++` operator. Forward iterators may be advanced using the increment operator `++` while bidirectional iterators may be incremented

as well as decremented (--). Random access iterators are the most general. They permit pointer jumps by arbitrary amounts as well as pointer arithmetic. C++ array iterators such as `y` in Program 5.9 are random access iterators.

5.3.5 An Iterator for `arrayList`

We shall define a C++ class `iterator` that will serve as a bidirectional iterator for `arrayList`. This class will, itself, be a public member of the class `arrayList`. Additionally, we shall add two public methods `begin()` and `end()` to `arrayList`. These methods, respectively, return iterators whose value is a pointer to the first element of the list (i.e., `element[0]`) and a pointer to one position past the last element (i.e., `element[listSize]`). The code for these two methods of `arrayList` is

```
class iterator;
iterator begin() {return iterator(element);}
iterator end() {return iterator(element + listSize);}
```

Program 5.11 gives the code for the class `iterator`. The five `typedef` statements are required by C++ to recognize our iterator class as a bidirectional iterator and to generate proper code for STL algorithms that employ bidirectional iterators. The complexity of each method of our iterator class is $\Theta(1)$.

An instance of our list iterator may be created and initialized using a statement such as

```
arrayList<int>::iterator x = y.begin();
```

where `y` is of type `arrayList`. With the addition of the iterator to our linear list class, we can use STL algorithms to perform tasks that require only the capabilities of a bidirectional iterator. For example, we can reverse the elements in a list `y` using the STL function `reverse` and we can sum the list elements using the STL function `accumulate`. The code for these two tasks is

```
reverse(y.begin(), y.end());
int sum = accumulate(y.begin(), y.end(), 0);
```

However, we cannot use the STL algorithm `sort` as this algorithm requires a random access iterator.

EXERCISES

- Let $L = (a, b, c, d, e)$ be a linear list that is represented in an array `element` using Equation 5.1. Assume that `arrayLength = 10`. Draw figures similar to Figure 5.2 showing the contents of the array `element` and the value of `listSize` following each operation in the operation sequence: initial state, `insert(0, f)`, `insert(3, g)`, `insert(7, h)`, `erase(0)`, `erase(4)`.

```
class iterator
{
    public:
        // typedefs required by C++ for a bidirectional iterator
        typedef bidirectional_iterator_tag iterator_category;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T* pointer;
        typedef T& reference;

        // constructor
        iterator(T* thePosition = 0) {position = thePosition;}

        // dereferencing operators
        T& operator*() const {return *position;}
        T* operator->() const {return &*position;}

        // increment
        iterator& operator++()    // preincrement
            {++position; return *this;}
        iterator operator++(int) // postincrement
            {iterator old = *this;
             ++position;
             return old;
            }

        // decrement
        iterator& operator--()    // predecrement
            {--position; return *this;}
        iterator operator--(int) // postdecrement
            {iterator old = *this;
             --position;
             return old;
            }

        // equality testing
        bool operator!=(const iterator right) const
            {return position != right.position;}
        bool operator==(const iterator right) const
            {return position == right.position;}
    protected:
        T* position; // pointer to a list element
};
```

Program 5.11 An iterator for the class arrayList

3. Write a function `changeLength2D` to change the length of a two-dimensional array. You must allow for a change in both dimensions of the array. Test your code.
4. To the class `arrayList` add a constructor that allows you to specify the amount by which the list capacity (or array length) is to be increased whenever array resizing is needed. When no capacity increment is specified, array doubling is done. Modify `insert` to work in this way. Test your code.
5. Write the method `arrayList<T>::trimToSize`, which makes the array length equal to `max{listSize,1}`. What is the complexity of your method? Test your code.
6. Write the method `arrayList<T>::setSize`, which makes the list size equal to the specified size. If the original list size was less than the new one, `NULL` elements are added, and if the original size was more than the new one, the extra elements are removed. What is the complexity of your method? Test your code.
7. Overload the operator `[]` so that the expression `x[i]` returns a reference to the `i`th element of the list. If the list doesn't have an `i`th element, an `illegalIndex` exception is to be thrown. The statements `x[i] = y` and `y = x[i]` should work as expected. Test your code.
8. Overload the operator `==` so that the expression `x == y` returns true iff the two array lists `x` are `y` are equal (i.e., the `i`th elements of both lists are equal for all `i`). Test your code.
9. Overload the operator `!=` so that the expression `x != y` returns true iff the two array lists `x` are `y` are not equal (see Exercise 8). Test your code.
10. Overload the operator `<` so that the expression `x < y` returns true iff the array list `x` is lexically smaller than the array list `y` (see Exercise 8). Test your code.
11. Write the method `arrayList<T>::push_back`, which inserts `theElement` at the right end of the list. Do not use the `insert` method. What is the time complexity of your method? Test your code.
12. Write the method `arrayList<T>::pop_back`, which erases the element at the right end of the list. Do not use the `erase` method. What is the time complexity of your method? Test your code.
13. Write the method `arrayList<T>::swap(theList)`, which swaps the elements of the lists `*this` and `theList`. What is the time complexity of your method? Test your code.

14. Write the method `arrayList<T>::reserve(theCapacity)`, which changes the capacity of the list to the larger of its current capacity and `theCapacity`. Test your code.
15. Write the method `arrayList<T>::set(theIndex, theElement)`, which replaces the element whose index is `theIndex` with `theElement`. Throw an exception in case `theIndex` is out of range. You should return the old element with the specified index. Test your code.
16. Write the method `arrayList<T>::clear`, which makes the list empty. What is the complexity of your method? Test your code.
17. Write the method `arrayList<T>::removeRange`, which removes all elements in the specified index range. What is the complexity of your method? Test your code.
18. Write the method `arrayList<T>::lastIndexOf`, which returns the index of the right-most occurrence of the specified object. A `-1` is returned in case the specified object is not in the list. What is the complexity of your method? Test your code.
19. Prove Theorem 5.1.
20. A shortcoming of the class `arrayList` (Program 5.1) is that it never decreases the length of the array `element`.
 - (a) Write a new version of this class so that if, following a deletion, the list size drops below `arrayLength/4`, a smaller array of length `max{arrayLength/2, initialCapacity}` is allocated and the elements are copied from the old array into the new one.
 - (b) (Optional) Consider any sequence of n linear list operations beginning with an empty list. Suppose that the total step count is $f(n)$ when the initial capacity equals or exceeds the maximum list size. Show that if we start with an initial capacity of 1 and use array resizing during inserts and removes as described above and in Section 5.3, the new step count is at most $cf(n)$ for some constant c .
21. Prove a theorem analogous to Theorem 5.1 for the case when array length is increased by a constant factor $c > 1$ whenever the array gets full and is reduced by the factor c whenever array occupancy falls below $1/(2c)$ (subject, of course, to the constraint that array length never falls below its initial length).
22. (a) Write the method `arrayList<T>::reverse`, which reverses the order of the elements in the list. The reversal is to be done in place (i.e., within the array `element` and without the creation of a new array). Note that before the reversal, the k th element (if it exists) of the list is in `element[k]`; following the reversal the k th element is in `element[listSize-k-1]`. Do not use the STL function `reverse`.

- (b) The complexity of your method should be linear in `listSize`. Show that this is the case.
- (c) Test the correctness of your code, using your own test data.
- (d) Now write another in-place method to reverse an object of type `arrayList`. This method is not a member of `arrayList` and should not access the data members of `arrayList`. Rather, your method should use the member methods of `arrayList` to produce the reversed list.
- (e) What is the time complexity of your new method?
- (f) Compare the run-time performance of the two reversal methods using linear lists of size 1000; 5000; and 10,000.
23. (a) Write the method `arrayList<T>::leftShift(i)` that shifts the list elements left by `i` positions. If `x = [0, 1, 2, 3, 4]`, then `x.leftShift(2)` results in `x = [2, 3, 4]`.
- (b) What is the time complexity of your method?
- (c) Test your code.
24. In a circular shift operation, the elements of a linear list are rotated clockwise by a given amount. For example, when the elements of `x = [0, 1, 2, 3, 4]` are shifted circularly by 2, the result is `x = [2, 3, 4, 0, 1]`.
- (a) Describe how you can perform a circular shift using three reversal operations. Each reversal may reverse a portion of the list or reverse the entire list.
- (b) Write the method `arrayList<T>::circularShift(i)`, which performs a circular shift by `i` positions. The complexity of your method should be linear in the list length.
- (c) Test your code.
25. The invocation `x.half()` eliminates every other element of `x`. So if `x.size()` is initially 7 and `x.element[] = [2, 13, 4, 5, 17, 8, 29]`, then following the execution of `x.half()`, `x.size()` is 4 and `x.element[] = [2, 4, 17, 29]`. If `x.size()` is initially 4 and `x.element[] = [2, 13, 4, 5]`, then following the execution of `x.half()`, `x.size()` is 2 and `x.element[] = [2, 4]`. If `x` is initially empty, then it is empty following the execution of `x.half()`.
- (a) Write code for the method `arrayList<T>::half()`. You should not use any of the other methods of `arrayList`. The complexity of your code should be $O(\text{size})$.
- (b) Show that the complexity of your code is, in fact, $O(\text{listSize})$.
- (c) Test your code.

26. Write a function equivalent to the method `half` of Exercise 25. Your function should not be a member of `arrayList` and should not access any of the data members of this class either. Rather, accomplish your task by using public methods of `arrayList`. What is the complexity of your method? Test your code.
27. Extend the iterator class `arrayList::iterator` (Program 5.11) so that it is a random access iterator. Test your iterator by using the STL `sort` function to sort the elements of a linear list.
28. Let `a` and `b` be two objects of type `arrayList`.
 - (a) Write the method `arrayList<T>::meld(a,b)`, which creates a linear list that contains elements alternately from `a` and `b`, beginning with the zeroth element of `a`. If you run out of elements in one list, then append the remaining elements of the other list to the list being created. The invocation `c.meld(a,b)` should make `c` the melded list. The complexity of your code should be linear in the sizes of the two input lists.
 - (b) Show that the complexity of your code is linear in the sum of the sizes of `a` and `b`.
 - (c) Test your code.
29. Let `a` and `b` be objects of type `arrayList`. Assume that the elements of `a` and `b` are in sorted order (i.e., nondecreasing from left to right).
 - (a) Write the method `arrayList<T>::merge(a,b)`, which creates a new sorted linear list that contains all the elements in `a` and `b`. The merged list is assigned to the invoking object `*this`. Do not use the STL function `merge`.
 - (b) What is the complexity of your method?
 - (c) Test your code.
30. (a) Write the method `arrayList<T>::split(a,b)`, which creates two linear lists `a` and `b`. `a` contains the elements of `*this` that have an even index, and `b` contains the remaining elements.
 - (b) What is the complexity of your method?
 - (c) Test your code.
31. Suppose that we are to represent a linear list using Equation 5.3. Rather than store the list size explicitly, we keep variables `first` and `last` that give the locations of the first and last elements of the list.
 - (a) Develop a class similar to `arrayList` for this representation. Name your class `circularArrayList`. Write code for all methods. You can make the `erase` and `insert` codes more efficient by properly choosing to move either elements to the left or right of the removed/inserted element.

- (b) What is the time complexity of each of your methods?
 - (c) Test your code.
32. Write a bidirectional iterator for `circularArrayList` class of Exercise 31.
33. Do Exercise 22 using Equation 5.3 instead of 5.1.
34. Do Exercise 28 using Equation 5.3 instead of 5.1.
35. Do Exercise 29 using Equation 5.3 instead of 5.1.
36. Do Exercise 30 using Equation 5.3 instead of 5.1.

5.4 VECTOR REPRESENTATION

The STL provides a class `vector` that uses an array and provides all of the functionality of the class `arrayList` (plus many additional methods). The length of the array used to implement a `vector` is dynamically increased as needed. Typically an insertion into a full vector will result in increasing the vector capacity by the larger of 1 and 50% of current capacity. The class `vector` doesn't have a constructor equivalent to that of `arrayList`; nor does it have methods with names `get`, `indexOf`, and `output`. However, `vector` has the methods `empty` and `size` that are equivalent to the corresponding methods of `arrayList`. Although `vector` has the methods `erase` and `insert` that respectively delete and add an element, the `vector` methods need to know the memory address (rather than element index) where the operation is to be performed. Another difference between `vector` and `arrayList` is that the two classes throw different types of exceptions when something goes wrong. To account for these differences, we define a class `vectorList` that uses a `vector` to represent a linear list and whose methods have the same signatures and behavior as those of `linearList`. Consequently, the classes `arrayList` and `vectorList` may be used interchangeably.

Programs 5.12 through 5.14 give the codes for some of the methods of the class `vectorList`.

EXERCISES

37. Write code for the method `vectorList<T>::half` (see Exercise 25). The complexity of your code should be linear in the size of the list. Test your code.
38. Write code for the method `vectorList<T>::meld(a,b)` (see Exercise 28). The complexity of your code should be linear in the sizes of the two input lists. Test your code.

```
template<class T>
class vectorList : public linearList<T>
{
public:
    // constructor, copy constructor and destructor
    vectorList(int initialCapacity = 10);
    vectorList(const vectorList<T>&);
    ~vectorList() {delete element;}

    // ADT methods
    bool empty() const {return element->empty();}
    int size() const {return (int) element->size();}
    T& get(int theIndex) const;
    int indexOf(const T& theElement) const;
    void erase(int theIndex);
    void insert(int theIndex, const T& theElement);
    void output(ostream& out) const;

    // additional method
    int capacity() const {return (int) element->capacity();}

    // iterators to start and end of list
    typedef typename vector<T>::iterator iterator;
    iterator begin() {return element->begin();}
    iterator end() {return element->end();}

protected: // additional members of vectorList
    void checkIndex(int theIndex) const;
    vector<T>* element; // vector to hold list elements
};
```

Program 5.12 An array linear list implemented using a vector

39. Write code for the method `vectorList<T>::merge(a,b)` (see Exercise 29).
Test your code.
40. Write code for the method `vectorList<T>::split(a,b)` (see Exercise 30).
Test your code.

```
template<class T>
vectorList<T>::vectorList(int initialCapacity)
{// Constructor.
    if (initialCapacity < 1)
        ostringstream s;
        s << "Initial capacity = " << initialCapacity << " Must be > 0";
        throw illegalParameterValue(s.str());
}

element = new vector<T>;
    // create an empty vector with capacity 0
element->reserve(initialCapacity);
    // increase vector capacity from 0 to initialCapacity
}

template<class T>
vectorList<T>::vectorList(const vectorList<T>& theList)
{// Copy constructor.
    element = new vector<T>(*theList.element);
}
```

Program 5.13 Constructors for vectorList

5.5 MULTIPLE LISTS IN A SINGLE ARRAY

Before accepting the array representation of a linear list, let us reflect on its merits. Certainly, the operations to be performed on a linear list can be implemented as very simple C++ methods. The methods `indexOf`, `remove`, and `add` have a worst complexity that is linear in the size of the individual list. We might regard this complexity as quite satisfactory. (In Chapter 15 we will see representations that allow us to perform these operations even faster.)

A negative aspect of the array representation is its inefficient use of space. Consider the following situation. We are to maintain three lists. We know that the three lists together will never have more than 4097 elements in them at any time. However, it is quite possible for a particular list to have 4097 elements at one time and for another list to have 4097 elements at another time. If we create three instances of `arrayList`, each with an initial array length of 4097, we will need space for a total of 12,291 elements even though we will never have more than 4097 elements at any time. However, using an initial length of 4097 for each array ensures that array resizing will not be required and our program will run as fast as possible. On the other hand, if we create three arrays with initial length 1, then when the length of one of these arrays is to increase from 4096 to 4097, we will need to first

```
template<class T>
void vectorList<T>::erase(int theIndex)
{// Delete the element whose index is theIndex.
// Throw illegalIndex exception if no such element.
    checkIndex(theIndex);
    element->erase(begin() + theIndex);
}

template<class T>
void vectorList<T>::insert(int theIndex, const T& theElement)
{// Insert theElement so that its index is theIndex.
    if (theIndex < 0 || theIndex > size())
// invalid index
    {
        ostringstream s;
        s << "index = " << theIndex << " size = " << size();
        throw illegalIndex(s.str());
    }

    element->insert(element->begin() + theIndex, theElement);
    // may throw an uncaught exception if insufficient
    // memory to resize vector
}
```

Program 5.14 Delete and insert for `vectorList`

create an array of length 8192 and copy 4096 elements into the new array. During the copy, both the 4096 and 8192 length arrays are needed. Therefore, space for at least 12,288 elements is needed.

In many applications of linear lists, the amount of memory used is not an issue because our computer has enough memory for the application to run to completion using the single-list-in-a-single-array representation. However, in applications that use very large lists, the list representations of this chapter may cause the application to fail (for insufficient memory) even though the total number of elements we have is small enough that all elements can be accommodated in the available memory. The application fails because excess memory has been allocated to a particular array or because array doubling fails.

One way to overcome this space requirement problem is to buy more memory. Another possibility is to map all of our lists into a single array `element` whose length is the maximum possible. In addition, we use two other arrays, `front` and `last`, to index into the array `list`. Figure 5.3 shows three lists represented in the single array `element`. We adopt the convention that the lists are numbered 1 through `m`.

there are m lists and that $\text{front}[i]$ is actually one less than the actual position of the zeroth element in list i . This convention on $\text{front}[i]$ makes it easier to use the representation. $\text{last}[i]$ is the actual position of the last element in list i . Notice that with this convention, $\text{last}[i] > \text{front}[i]$ whenever the i th list is not empty. We shall have $\text{front}[i] = \text{last}[i]$ whenever list i is empty. So in the example of Figure 5.3, list 2 is empty. The lists are represented in the array in the order 1, 2, 3, \dots , m from left to right.

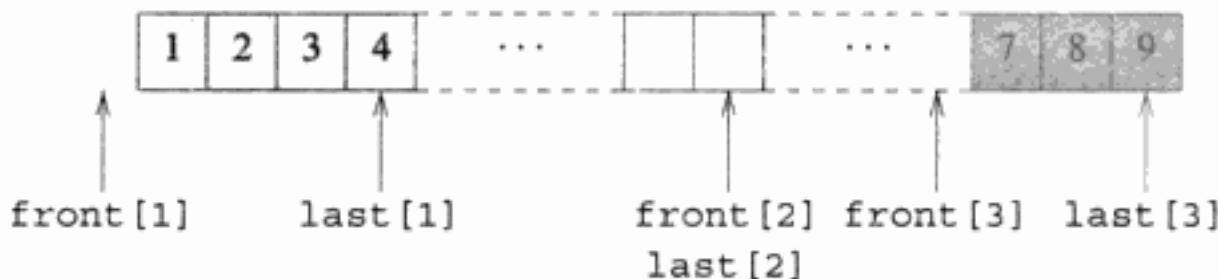


Figure 5.3 Three lists in a single array

To avoid having to handle the first and last lists differently from others, we define two boundary lists 0 and $m+1$ with $\text{front}[0] = \text{last}[0] = -1$ and $\text{front}[m+1] = \text{last}[m+1] = \text{list.length-1}$. To insert an element as the index^{th} element of list i , we need to first create space for the new element. If $\text{last}[i] = \text{front}[i+1]$, then there is no space between lists i and $i+1$ and we cannot move elements index through the last one up one position. At this time we can check whether it is possible to move elements 0 through $\text{index}-1$ of the i th list one position down by checking the relation $\text{last}[i-1] < \text{front}[i]$. If this relation does not hold, then we need to either shift some of the lists 1 through $i-1$ down or some of the lists $i+1$ through m up and create space for list i to grow. This shifting is possible when the total number of elements in all the lists is fewer than list.length .

Figure 5.4 is a pseudo-C++ version of the method to insert an element into list i . This pseudocode may be refined into compilable C++ code.

Although representing several lists in a single array uses space more efficiently than using a separate array for each list, insertions take more time in the worst case. In fact, a single insertion could require us to move as many as arrayLength-1 elements, where arrayLength is the length of the array list . The multiple-list-in-a-single-array representation is also quite cumbersome to implement. A much simpler solution whose space requirement equals that for the elements in all the lists plus that for one pointer per element is the subject of the next chapter.

EXERCISES

41. Refine Figure 5.4 into a C++ method and test its correctness.

```
void insert(int i, int index, Object element)
{ // Insert y as the index'th element in list i.
    size = last[i] - front[i]; // number of elements in list i
    if (index < 0 || index > size)
        throw an exception;
    // Is there space on the right?
    Find the least j, j ≥ i such that last[j] < front[j+1];
    If such a j exists, then move lists i+1 through j and elements index through
    the last one of list i up one position and insert element into list i;
    This move should update appropriate last and first values;

    // Is there space on the left?
    If no j was found above, then find the largest j, j < i such that
    last[j] < front[j+1];
    If such a j is found, then move lists j through i-1
    and elements 1 through index-1 of list i one position left and insert element;
    This move should update appropriate last and first values;

    // Success?
    if (no j was found above) throw an exception;
}
```

Figure 5.4 Pseudocode to insert an element in the many lists per array representation

42. Write a C++ method to insert an element as the *index*th element in list *i*. Assume that a single array represents *m* lists. If you have to move lists to accommodate the new element, your should first determine the amount of available space and then move the lists so that each has about the same amount of space available for future growth. Test your code.
43. Write a C++ method to remove the *index*th element from list *i*. Assume that a single array represents *m* lists. Test the correctness of your code by compiling and executing it.

5.6 PERFORMANCE MEASUREMENT

In this chapter we have developed two array classes that implement the data structure *linear list*—*arrayList* and *vectorList*. Both classes are equally good as far as their space complexity is concerned. Even though both classes offer the same asymptotic time complexity, their actual run times are likely to be different.

To obtain the actual run times, we must design an experiment. We wish to measure the time taken by the operations `get`, `indexOf`, `erase`, and `insert`. For the `get` and `indexOf` operations, we measure the total time required for the sequences `get(i)`, $0 \leq i < \text{listSize}$ and `indexOf(ei)`, $0 \leq i < \text{listSize}$, where e_i is the i th element of the list. Figure 5.5 gives the measured times for `listSize = 50,000`, and Figure 5.6 shows these times as bar graphs.

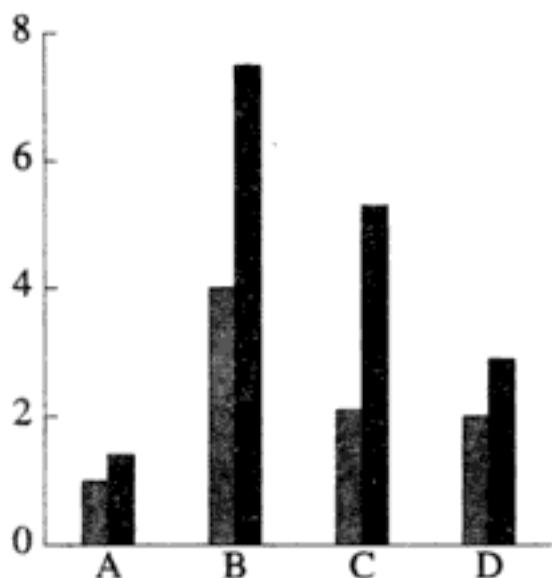
operation	arrayList	vectorList
<code>get</code>	1.0 ms	1.4 ms
<code>indexOf</code>	2.3 s	2.3 s
best-case inserts	4.0/2.1 ms	7.5/5.3 ms
average inserts	1.5/1.5 s	1.5/1.5 s
worst-case inserts	2.5/2.5 s	2.7/2.5 s
best-case erases	2.0 ms	2.9 ms
average erases	1.5 s	1.5 s
worst-case erases	2.5 s	2.4 s

Times for 50,000 operations

Figure 5.5 Time taken by different array linear list implementations

For the `insert` operation, we do a sequence of $n = 50,000$ inserts beginning with an empty list and report the total time for the 50,000 inserts. The best case for the insert sequence is when each new element is inserted at the right end of the list; the worst case is when each new element is inserted at the left end. To estimate average behavior, we do the inserts at randomly generated positions of the list. Figure 5.5 gives the insert times in the format TA/TB , where TA is the time when the list is constructed with the default initial capacity of 10 and TB is the time when the initial capacity is 50,000. For best-case inserts, array doubling increases run time by about 90 percent for `arrayList` when compared with the case when no array resizing is done. Increasing array size by a factor of 50 percent whenever array resizing is needed increases the run time of a best-case insert in `vectorList` by about 42 percent when compared to the case when no array resizing is done. The total time spent resizing the element array was 1.9 ms in the case of `arrayList` and 2.2 ms in the case of `vectorList`. For the average and worst-case tests, the array resizing time is a negligible part of the total cost. This result is to be expected because both array doubling and increasing array size by 50 percent add $\Theta(n)$ to the cost of n inserts; the cost of n best-case inserts is $\Theta(n)$, and the cost of n average- and worst-case inserts is $\Theta(n^2)$.

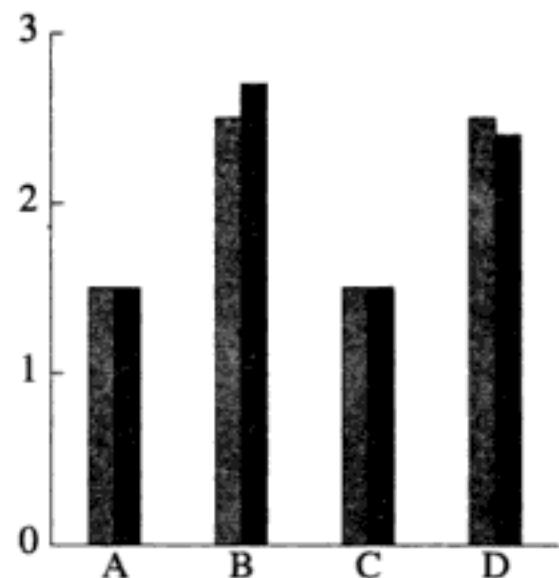
Notice the colossal increase in run time from the best-case inserts to the worst-case inserts—the run time for `arrayList` jumped from 4.0 ms to 2.5 seconds, a 625-fold increase. This 625-fold increase isn't altogether surprising given that n best-case inserts take $\Theta(n)$ time, whereas n worst-case inserts take $\Theta(n^2)$ time. If



A = `get`
 B = `inserts with array resizing`
 C = `inserts with no array resizing`
 D = `erases`

`arrayList`

(a) Best times in milliseconds



A = `average inserts`
 B = `worst-case inserts`
 C = `average erases`
 D = `worst-case erases`

`vectorList`

(b) Average and worst times in seconds

Figure 5.6 Plot of run times

the constant factors in the best-case and worst-case time expressions are the same (and they are not), we would expect to see the time go up by a factor of almost $n = 50,000$.

The average insert time is approximately half the worst-case insert time. This result is to be expected because, on average, half the elements have to be moved during an insert; in the worst-case, all elements are moved.

For the `erase` operation, we start with a list that has $n = 50,000$ elements and do a sequence of n removes. The best case for the remove sequence is when each remove operation removes the element at the right end of the list; the worst case is when each remove operation removes the element at the left end. To estimate average behavior, we do the removes from randomly generated positions of the list.

For the `get` operation as well as for best-case inserts and erases, `arrayList` is considerably faster than `vectorList`. However, for the `indexOf` as well as for average- and worst-case inserts and erases the two classes perform about the same. This result is to be expected, because of the significantly larger overheads associated with the `vector` class relative to those associated with an array. For $O(1)$ time operations such as `get` and best-case inserts and erases, this overhead dominates

the run time and causes `vectorList` to perform poorly. But for the $O(n)$ operations such as `indexOf` and average- and worst-case inserts and deletes, the overhead is dwarfed by the time spent searching for or moving elements.

So which class should you use? If your primary operation is `get` or if you are doing `inserts` and `erases` primarily from the right end of the list (as is the case for the stack data structure of Chapter 8), use `arrayList`. For other applications of a linear list, either `arrayList` or `vectorList` may be used. But wait; we have yet to see other linear list implementations. These might be even faster!

EXERCISE

44. Develop the class `arrayListNoSTL`, which implements a linear list using an array. However, unlike the class `arrayList`, the class `arrayListNoSTL` doesn't use any STL function. So, for example, STL functions such as `copy`, `copy_backward`, and `find` should not be used. Repeat the experiment described in this section obtaining run times for `arrayList`, `vectorList`, and `arrayListNoSTL`. Present your results in both tabular and bar chart forms.

5.7 REFERENCES AND SELECTED READINGS

Additional material on data structures in C++ may be found in the texts *C++ Plus Data Structures*, Third Edition, by N. Dale, Jones and Bartlett, Sudbury, MA, 2003; *Data Structures and Algorithm Analysis in C++*, Second Edition, by M. Weiss, Addison-Wesley, Menlo Park, CA, 1998; *Data Structures and Algorithms in C++* by M. Goodrich, R. Tamassia, and D. Mount, John Wiley & Sons, New York, NY, 2002; and *Fundamentals of Data Structures in C++* by E. Horowitz, S. Sahni and D. Mehta, Computer Science Press, New York, NY, 1995.

CHAPTER 6

LINEAR LISTS—LINKED REPRESENTATION

BIRD'S-EYE VIEW

The array representation of a linear list is so natural that you may think there is no other reasonable way to represent a linear list. This chapter will dispel any such thought you may have.

In a linked representation, the elements of a list may be stored in any arbitrary set of memory locations. Each element has an explicit pointer or link (the terms *pointer* and *link* are synonyms) that tells us the location (i.e., the address) of the next element in the list.

In an array representation, the element addresses are determined by using a mathematical formula; and in a linked representation, the element addresses are distributed across the list elements.

The data structure concepts introduced in this chapter are

- Linked representation.
- Chains, circular lists, and doubly linked lists.
- Header nodes.

The STL container class `list` uses a doubly linked circular list with a header node to represent its instances. The methods of `list` have the same signatures

and behavior as do those of the class `vector`. Hence the `list` methods `erase` and `insert` don't have the same signatures as required by our ADT `linearList`. However, as was the case for `vector`, `list` may be used in the development of a concrete linear list class that derives from the abstract class `linearList`.

The applications developed in this chapter are bin sort (also known as bucket sort), radix sort, convex hulls, and the union-find problem. Bin sort, radix sort and the union-find problem use chains; the convex hull application uses a doubly linked list. Using either a bin sort or a radix sort, you can sort n elements in $O(n)$ time provided the keys are in an "appropriate range." Although the sort methods developed in Chapter 2 take $O(n^2)$ time, they do not require the keys to lie in a particular range. Bin sort and radix sort are considerably faster than the sort methods of Chapter 2 when the keys lie in an appropriate range. The union-find problem illustrates how linked lists may be built using integers as pointers.

6.1 SINGLY LINKED LISTS AND CHAINS

6.1.1 The Representation

In a linked representation each element of an instance of a data object is represented in a cell or node. The nodes, however, need not be components of an array, and no formula is used to locate individual elements. Instead, each node keeps explicit information about the location of other relevant nodes. This explicit information about the location of another node is called a **link or pointer**.

Let $L = (e_0, e_1, \dots, e_{n-1})$ be a linear list. In one possible linked representation for this list, each element e_i is represented in a separate node. Each node has exactly one link field that is used to locate the next element in the linear list. So the node for e_i links to that for e_{i+1} , $0 \leq i < n - 1$. The node for e_{n-1} has no node to link to and so its link field is **NULL**. The variable **firstNode** locates the first node in the representation. Figure 6.1 shows the linked representation of the list $L = (e_0, e_1, \dots, e_{n-1})$. Links are shown as arrows. To locate the element e_2 , for example, we must start at **firstNode**; follow the pointer in **firstNode** to the next node; follow one more pointer to get to the node with e_2 . In general, to locate the element with index *theIndex*, we must follow a sequence of *theIndex* pointers beginning at **firstNode**.

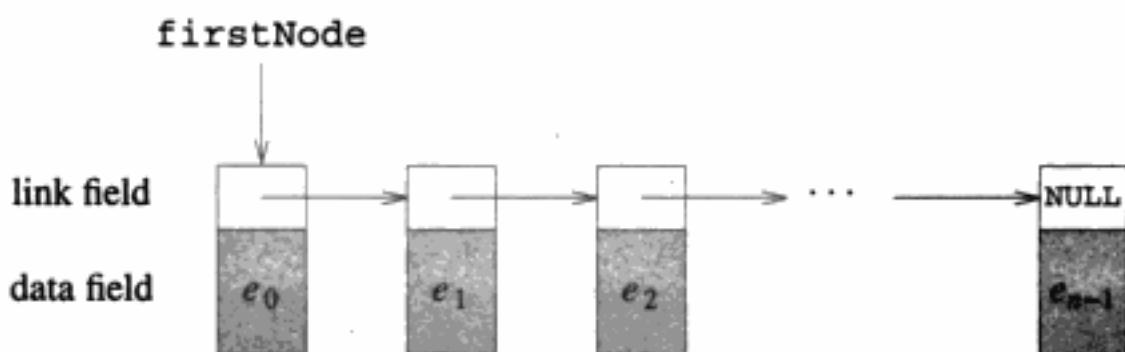


Figure 6.1 Linked representation of a linear list

Since each node in the linked representation of Figure 6.1 has exactly one link, the structure of this figure is called a **singly linked list**. Since the nodes are ordered from left to right with each node (other than the last one) linking to the next, and the last node has a **NULL** link, the structure is also called a **chain**.

To remove the element e_2 whose index is 2 from the chain of Figure 6.2, we do the following (note that e_2 is in the third node of the chain):

- Locate the second node (i.e., the node with e_1) in the chain.
- Link the second node to the fourth node.

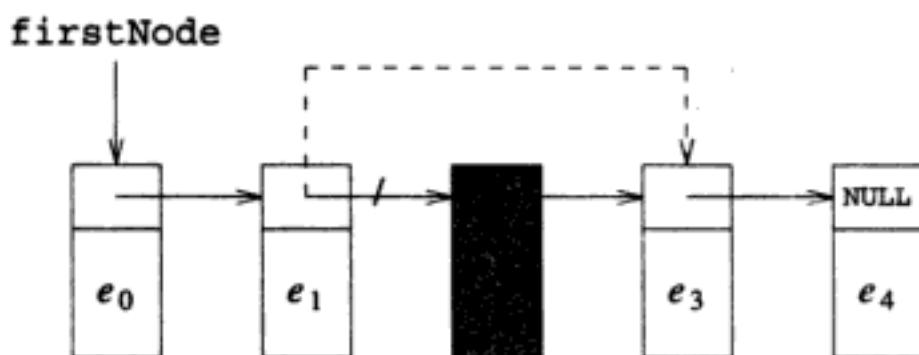


Figure 6.2 Removing e_2 from a 5-node chain

Notice that the removal of the third node from Figure 6.2 automatically decrements the index of the succeeding nodes by 1 (i.e., what were previously the fourth and fifth nodes of the chain become the third and fourth nodes). Because the nodes on a chain are always defined to be those nodes that can be reached following a sequence of pointers beginning at `firstNode`, we do not bother to change the pointer in the removed node (i.e., the former third node of the chain). Since the removed node is no longer reachable from `firstNode`, it is no longer part of the chain.

To insert a new element as the `index`th element in a chain, we need to first locate the `index - 1`th element and then insert a new node just after it. Figure 6.3 shows the link changes needed for the two cases $\text{index} = 0$ and $0 < \text{index} \leq \text{listSize}$. Solid pointers exist prior to the insert, and those shown as broken (or dashed) lines exist following the insert.

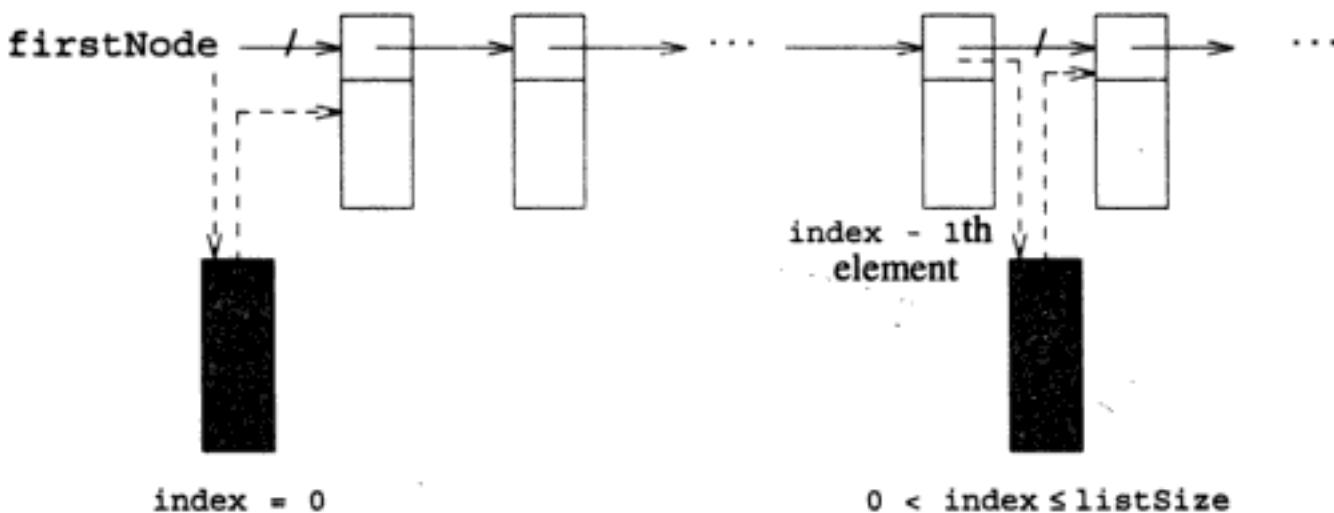


Figure 6.3 Insertion into a chain

6.1.2 The Struct `chainNode`

To represent a linear list as a chain, we define a struct `chainNode` and a class `chain`. Program 6.1 gives the struct `chainNode`, which defines the data type of the nodes used in Figure 6.1. The data member `element` holds a list element and is the data field of the node; the data member `next` holds a pointer to the next node in the chain and is the node's link field.

```
template <class T>
struct chainNode
{
    // data members
    T element;
    chainNode<T> *next;

    // methods
    chainNode() {}
    chainNode(const T& element)
        {this->element = element;}
    chainNode(const T& element, chainNode<T>* next)
        {this->element = element;
         this->next = next;}
};
```

Program 6.1 Struct definition for a chain node

Notice that two of the constructors of `chainNode` use the syntax `this->element` and `this->next` to access the data members of the constructed instance. This syntax is necessary to distinguish between the data members of the constructed instance and the formal parameters of the constructors because the data members and formal parameters have the same names.

6.1.3 The Class `chain`

Header, `empty`, and `size`

The class `chain` implements a linear list as a singly linked list of nodes in which the last node has the pointer `NULL`; that is, it implements a linear list as a chain of nodes. Program 6.2 gives the class header, data members and code for the `empty`, and `size` methods.

The data members are `firstNode` and `listSize`. `firstNode` is a pointer to the first node (i.e., the node for the zeroth element of the list) in the chain. When the chain has no first node, that is, when the chain is empty, `firstNode` is `NULL`.

```

template<class T>
class chain : public linearList<T>
{
public:
    // constructor, copy constructor and destructor
    chain(int initialCapacity = 10);
    chain(const chain<T>&);
    ~chain();

    // ADT methods
    bool empty() const {return listSize == 0;}
    int size() const {return listSize;}
    T& get(int theIndex) const;
    int indexOf(const T& theElement) const;
    void erase(int theIndex);
    void insert(int theIndex, const T& theElement);
    void output(ostream& out) const;

protected:
    void checkIndex(int theIndex) const;
        // throw illegalIndex if theIndex invalid
    chainNode<T>* firstNode; // pointer to first node in chain
    int listSize;           // number of elements in list
};

```

Program 6.2 Header for the class *chain*

listSize gives the number of elements in the list, which equals the number of nodes in the chain.

Constructor and copy constructor

Program 6.3 gives the constructor and copy constructor for *chain*.

To create an empty chain, we need merely set the first node pointer *firstNode* to *NULL*. Unlike the case when a linear list is represented by an array, we do not allocate space for the expected maximum number of elements at the time the chain is created. Therefore, the user does not need to specify an estimate of this maximum size or provide an initial capacity for the list. Nevertheless, we have provided a constructor which has *initialCapacity* as a formal parameter in order to be compatible with the class *arrayList*. In particular, an application can create an array of type *linearList* and initialize the array components using either form of constructor as is shown below.

```
template<class T>
chain<T>::chain(int initialCapacity)
{// Constructor.
    if (initialCapacity < 1)
        ostringstream s;
        s << "Initial capacity = " << initialCapacity << " Must be > 0";
        throw illegalParameterValue(s.str());
    }
    firstNode = NULL;
    listSize = 0;
}

template<class T>
chain<T>::chain(const chain<T>& theList)
{// Copy constructor.
    listSize = theList.listSize;

    if (listSize == 0)
        // theList is empty
        firstNode = NULL;
        return;
    }

    // non-empty list
    chainNode<T>* sourceNode = theList.firstNode;
        // node in theList to copy from
    firstNode = new chainNode<T>(sourceNode->element);
        // copy first element of theList
    sourceNode = sourceNode->next;
    chainNode<T>* targetNode = firstNode;
        // current last node in *this
    while (sourceNode != NULL)
        // copy remaining elements
        targetNode->next = new chainNode<T>(sourceNode->element);
        targetNode = targetNode->next;
        sourceNode = sourceNode->next;
    }
    targetNode->next = NULL; // end the chain
}
```

Program 6.3 Constructor and copy constructor for chain

```
linearList<int>* list[10];
list[0] = new arrayList<int>(20);
list[1] = new arrayList<int>();
list[2] = new chain<int>(5);
list[3] = new chain<int>;
```

The complexity of the constructor is $\Theta(1)$. The copy constructor makes a clone of the chain `theChain` by copying the nodes of `theChain` one node at a time. The complexity of the copy constructor is $O(\text{theList.listSize})$.

The destructor

Program 6.4 gives the destructor for `chain`. The destructor deletes the nodes of the chain one by one. The strategy used in our destructor code is to repeatedly delete the first node in the chain until the remaining chain has no first node. Note that we must save the pointer to the second node in a variable such as `nextNode` before we delete the first node. The complexity of the destructor is $O(\text{listSize})$.

```
template<class T>
chain<T>::~chain()
{// Chain destructor. Delete all nodes in chain.
    while (firstNode != NULL)
        {// delete firstNode
            chainNode<T>* nextNode = firstNode->next;
            delete firstNode;
            firstNode = nextNode;
        }
}
```

Program 6.4 Destructor for `chain`

The method `get`

When an array representation is used, we locate a list element by evaluating a (usually simple) formula. To find the `theIndex`th element of a chain, however, we must start at the first node and follow the `next` links until we reach the desired node; that is, we must follow `theIndex` number of links. *We cannot access the desired node by performing arithmetic on the value of `firstNode`.* Program 6.5 gives the code. The method `checkIndex` is the same as that defined for `arrayList`. The complexity of `chain<T>::get` is $O(\text{theIndex})$, while that of `arrayList<T>::get` is $\Theta(1)$.

```
template<class T>
T& chain<T>::get(int theIndex) const
{// Return element whose index is theIndex.
// Throw illegalIndex exception if no such element.
    checkIndex(theIndex);

    // move to desired node
    chainNode<T>* currentNode = firstNode;
    for (int i = 0; i < theIndex; i++)
        currentNode = currentNode->next;

    return currentNode->element;
}
```

Program 6.5 Method to return the `theIndex`th element

The method `indexOf`

Program 6.6 gives the code for the method `chain<T>::indexOf`. This code differs from the code for `arrayList<T>::indexOf` primarily in the mechanism used to go from one list element to the next. In the case of an array list, we go from one element to the next by performing some arithmetic on the location of the current, element (when Equation 5.1 is used, we add 1 to the current location to get to the next location). When a chain is used, the only way to go from one node to the next is to follow the link or pointer in the current node. The complexity of `chain<T>::indexOf` is $O(\text{listSize})$.

The method `erase`

Program 6.7 gives the code for the `erase` operation. There are three cases to consider:

- `theIndex < 0` or `theIndex ≥ listSize`. In this case the `erase` operation fails because there is no `theIndex`th element to erase. This case implicitly covers the case when the chain is empty.
- The zeroth element is to be erased from a nonempty chain.
- An element other than the zeroth element is to be erased.

To get a feel for Program 6.7, manually try it on an initially empty list as well as on lists that contain at least one node. In addition, try out values of `theIndex` such as `theIndex < 0`, `theIndex = 0` (erase the zeroth element), `theIndex = listSize-1` (erase the last element), `theIndex ≥ listSize`, and `0 < theIndex < listSize-1` (erase an interior element).

```
template<class T>
int chain<T>::indexOf(const T& theElement) const
{// Return index of first occurrence of theElement.
// Return -1 if theElement not in list.

    // search the chain for theElement
    chainNode<T>* currentNode = firstNode;
    int index = 0; // index of currentNode
    while (currentNode != NULL &&
           currentNode->element != theElement)
    {
        // move to next node
        currentNode = currentNode->next;
        index++;
    }

    // make sure we found matching element
    if (currentNode == NULL)
        return -1;
    else
        return index;
}
```

Program 6.6 Method to return the index of the first occurrence of theElement

The complexity of `chain<T>::erase` is $O(\text{theIndex})$, whereas the complexity of `arrayList<T>::erase` is $O(\text{listSize}-\text{theIndex})$. Therefore, the linked implementation of a linear list is expected to perform better than the array implementation for erases that are done from near the front of the list.

The method insert

Insertion and erasing work in a similar way. To insert a new element as the `theEnd`th one in a chain, we need to first locate the `theIndex-1`th element and then insert a new node just after it. Program 6.8 gives the code. Its complexity is $O(\text{theIndex})$.

Outputting a chain

Program 6.9 gives the code for the `output` method as well as for the overloading of the stream insertion operator `<<`. The code for `chain<T>::output` differs from that for `arrayList<T>::output` primarily in its use of the `next` pointer to go from

```
template<class T>
void chain<T>::erase(int theIndex)
{// Delete the element whose index is theIndex.
// Throw illegalIndex exception if no such element.
checkIndex(theIndex);

// valid index, locate node with element to delete
chainNode<T>* deleteNode;
if (theIndex == 0)
{// remove first node from chain
    deleteNode = firstNode;
    firstNode = firstNode->next;
}
else
{ // use p to get to predecessor of desired node
    chainNode<T>* p = firstNode;
    for (int i = 0; i < theIndex - 1; i++)
        p = p->next;

    deleteNode = p->next;
    p->next = p->next->next; // remove deleteNode from chain
}
listSize--;
delete deleteNode;
}
```

Program 6.7 Erase the theIndexth element

one node to the next. The complexity of `chain<T>::output` is the same as that of `arrayList<T>::output`, $O(\text{listSize})$.

The member class iterator

By using `next` pointers, we can efficiently move from a node to its successor node in the chain. However, in a chain, there is no efficient way to move from a node to its predecessor. Therefore, for a chain, we define only a forward iterator. Recall that for an `arrayList` we defined a bidirectional iterator that enabled us to move from any list element to both its successor element and its predecessor element in $O(1)$ time. Program 6.10 gives the code for some of the methods of the forward iterator for a chain. The complete code may be found at the Web site for this book.

The methods `chain<T>::begin` and `chain<T>::end` are defined as

```

template<class T>
void chain<T>::insert(int theIndex, const T& theElement)
// Insert theElement so that its index is theIndex.
{
    if (theIndex < 0 || theIndex > listSize)
        // invalid index
        ostringstream s;
        s << "index = " << theIndex << " size = " << listSize;
        throw illegalIndex(s.str());
}

if (theIndex == 0)
    // insert at front
    firstNode = new chainNode<T>(theElement, firstNode);
else
{ // find predecessor of new element
    chainNode<T>* p = firstNode;
    for (int i = 0; i < theIndex - 1; i++)
        p = p->next;

    // insert after p
    p->next = new chainNode<T>(theElement, p->next);
}
listSize++;
}

```

Program 6.8 Insert theElement as the theIndexth element of the chain

```

iterator begin() {return iterator(firstNode);}
iterator end() {return iterator(NULL);}

```

The difference in run times between using the `get` method and the `iterator` method to access the linear list elements in left-to-right order is quite dramatic when the list is represented as a chain. The time needed to access the i th element using `get` is $\Theta(i)$. Therefore, the `get` method to examine the list elements one at a time takes $\Theta(\text{listSize}^2)$ time, whereas the `iterator` method takes only $\Theta(\text{listSize})$ time.

6.1.4 Extensions to the ADT *linearList*

In some applications of linear lists, we wish to perform operations in addition to those that are part of the abstract data type *linearList* (ADT 5.1). So it is useful to

```
template<class T>
void chain<T>::output(ostream& out) const
{// Put the list into the stream out.
    for (chainNode<T>* currentNode = firstNode;
          currentNode != NULL;
          currentNode = currentNode->next)
        out << currentNode->element << " ";
}

// overload <<
template <class T>
ostream& operator<<(ostream& out, const chain<T>& x)
{x.output(out); return out;}
```

Program 6.9 The method output

extend the ADT to include additional functions such as *clear* (remove all elements from the list) and *push_back(theElement)* (insert *theElement* at the end of the list). Program 6.11 gives the abstract class that corresponds to the extended ADT.

6.1.5 The Class `extendedChain`

We will develop a class `extendedChain` that provides a linked implementation of the abstract class `extendedLinearList`. The easiest way to arrive at the class `extendedChain` is to derive it from `chain`.

To efficiently insert an element at the end of a chain, we add a new data member `lastNode` that points to the last node in the chain. Using this pointer, we can append an element to a chain in $\Theta(1)$ time. However, the addition of this new data member requires us to make changes in the implementation of the methods `erase` and `insert` because these methods may change the last node in the chain. When these methods change the last node, they must also update the new data member `lastNode`. Therefore, the class `extendedChain` will declare the data member `lastNode`; provide modified implementations of the methods `erase` and `insert`; define the remaining pure virtual functions of `linearList` as invocations of the corresponding methods in the class `chain`; and provide implementations for the new methods `clear` and `push_back`.

Program 6.12 gives the code for the methods `clear` and `push_back`. The full code for `extendedChain` may be found at the Web site for this book.

```
class iterator
{
public:
    // typedefs required by C++ for a forward iterator omitted

    // constructor
    iterator(chainNode<T>* theNode = NULL)
        {node = theNode; }

    // dereferencing operators
    T& operator*() const {return node->element;}
    T* operator->() const {return &node->element;}

    // increment
    iterator& operator++() // preincrement
        {node = node->next; return *this;}
    iterator operator++(int) // postincrement
        {iterator old = *this;
         node = node->next;
         return old;
        }

    // equality testing
    bool operator!=(const iterator right) const
        {return node != right.node;}
    bool operator==(const iterator right) const
        {return node == right.node;}
protected:
    chainNode<T>* node;
};
```

Program 6.10 The class `chain<T>::iterator`

6.1.6 Performance Measurement

Memory Comparison

In an array implementation of a linear list, we typically use array doubling when the array gets full and array halving when the array occupancy falls below 25 percent (note, however, that the STL container class `vector` increases array length by a factor of 1.5 and never decreases array length; our array-list classes use array doubling and do not decrease array length when usage falls below 25 percent of

```
template<class T>
class extendedLinearList : linearList<T>
{
public:
    virtual ~extendedLinearList() {}
    virtual void clear() = 0;
        // empty the list
    virtual void push_back(const T& theElement) = 0;
        // insert theElement at end of list
};
```

Program 6.11 Abstract class for extended linear list

```
template<class T>
void extendedChain<T>::clear()
{// Delete all nodes in chain.
    while (firstNode != NULL)
        {// delete firstNode
            chainNode<T>* nextNode = firstNode->next;
            delete firstNode;
            firstNode = nextNode;
        }
    listSize = 0;
}

template<class T>
void extendedChain<T>::push_back(const T& theElement)
{// Insert theElement at the end of the chain.
    chainNode<T>* newNode = new chainNode<T>(theElement, NULL);
    if (firstNode == NULL)
        // chain is empty
        firstNode = lastNode = newNode;
    else
        { // attach next to lastNode
            lastNode->next = newNode;
            lastNode = newNode;
        }
    listSize++;
}
```

Program 6.12 The methods clear and push_back of extendedChain<T>

array capacity). Therefore, a linear list with n elements may reside in an array whose length is between n and $4n$. So space for between n and $4n$ elements is needed. When a chain is used, exactly n nodes, each with two fields, are allotted to the list. Therefore, the chain representation uses space for n elements and n pointers. Let s be the number of bytes required by an element and assume that a pointer requires 4 bytes. Ignoring the space required by class data members such as `size` and `firstNode`, the array representation of a linear list requires between ns and $4ns$ bytes while a singly-linked representation requires $n(s + 4)$ bytes. For most applications this difference in the space requirements will not be a deciding factor in selecting the representation to use.

Run-Time Comparison

For the time requirements we expect `chain<T>::get` to be much slower than `ArrayList<T>::get`, because the complexity of `chain<T>::get` is $O(\text{listSize})$, whereas the complexity of the `arrayList<T>::get` is $\Theta(1)$. This expectation is borne out by experiment. We constructed a 50,000-node chain by making successive inserts at the left end of an initially empty chain. Then we measured the total time required to perform the 50,000 get operations `get(i)`, $0 \leq i < 50000$. `chain<T>::get` took 13.2 seconds to perform this sequence of operations, while `arrayList<T>::get` took 1.0 ms—not a commendable showing for the `chain` class. Things do not get any better when we compare the times for the `indexOf`, `insert` and `erase` methods.

Figures 6.4 and 6.5 show the times taken by `arrayList` and `chain` for operation sequences as described in Section 5.6. The time for 50,000 `indexOf` operations using `chain<T>::indexOf` is approximately 6 times that when `arrayList<T>::indexOf` is used. The average-case insert and erase times for the class `chain` are about 33 and 46 times that for the corresponding methods of `arrayList`.

operation	arrayList	chain
<code>get</code>	1.0 ms	13.2 s
<code>indexOf</code>	2.3 s	13.0 s
best-case inserts	2.1 ms	45.1 ms
average inserts	1.5 s	49.3 s
worst-case inserts	2.5 s	12.9 s
best-case erases	2.0 ms	2.1 ms
average erases	1.5 s	68.8 s
worst-case erases	2.5 s	12.9 s

Times for 50,000 operations

Figure 6.4 Time taken by different linear list implementations

Even though the insert and erase sequences (`insert` and `erase` from the right

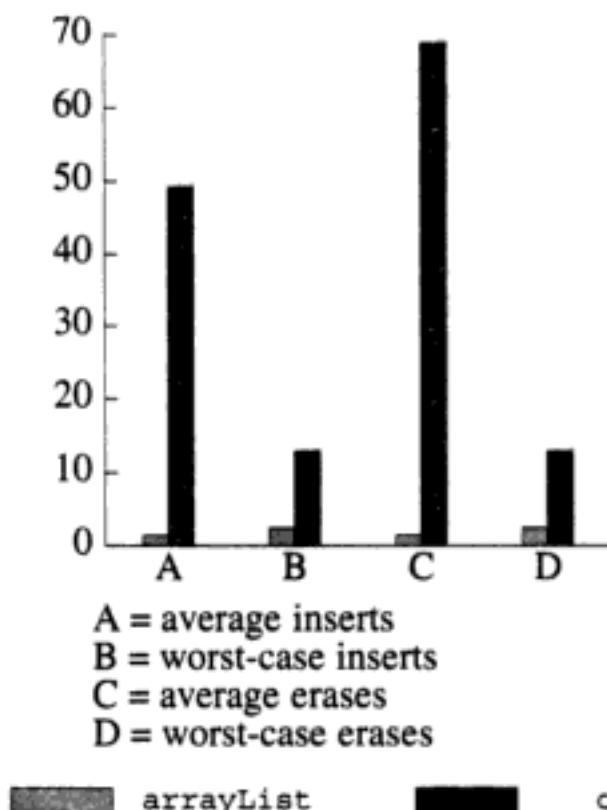


Figure 6.5 Average- and worst-case times, in seconds, for 50,000 operations

end of a chain) used in our worst-case test for chains cause our methods to do the maximum work, these sequences do not ensure maximum run time because of the cache effect (Section 4.5). In fact, you will notice that the measured worst-case times are smaller than the measured average times!

The average-case insert times were obtained by inserting into random positions of the linear list. Consequently, nodes that are adjacent in the chain are randomly located in memory. Hence when you move down the chain from left to right, you need to access random memory locations. This results in many cache misses. The same is true for the average-case erase experiment, which uses a randomly constructed chain. For the worst-case experiment, successive inserts are made at the right end of the chain. Since, for our experimental setup, successive calls to `new` return nodes that are adjacent in memory, nodes that are adjacent in the chain also are adjacent in memory. So when you move down the chain from left to right, you access adjacent memory; a memory access pattern that is favored by the cache management scheme. So the number of cache misses is reduced. This reduction in cache misses results in the anomalous run time measurements for worst-case inserts and erases relative to the average times.

The reported times for `chain<T>::get` and `chain<T>::indexOf` were obtained from a chain constructed by making inserts at the front of the chain (best-case

inserts). Since, in our experimental setup, successive calls to `new` return nodes that are adjacent in memory, adjacent nodes in the chain used to measure the times for the `get` and `indexOf` sequences occupy adjacent memory. So expect that performing the same sequence of `get` and `indexOf` operations in a chain created by making random inserts would take much more time. In fact, in a separate experiment, we determined this time to be 167 seconds and 165 seconds, respectively. So the time needed for our sequence of 50,000 `gets` in a randomly constructed chain is about 13 times that in the chain constructed for best-case inserts. This ratio is about the same for the `indexOf` operation. The linked representation gets a thumbs down as far as standard linear list operations are concerned!

Note that cache effects did not play a role in our comparison of best-case, average, and worst-case times for array representation (Section 5.6) because in all tests, the array elements are accessed from left to right and an array occupies a contiguous block of memory.

Are Pointers Any Good?

You are probably wondering whether you have just wasted a lot of time studying pointers. In Chapter 15 we develop balanced binary tree structures such as AVL and red-black trees. Indexed versions of these structures (e.g., the indexed AVL tree) may be used to represent a linear list (see Exercise 15.20). These structures use pointers and knock the socks off of `arrayList` as far as worst-case inserts and erases are concerned.

Despite the poor showing of chains in the linear list timing experiments we conducted, chains are more efficient than array linear list representations in several linear list applications. Section 6.5 gives a few of these applications. These applications require us to combine multiple lists into one or to remove and insert elements when the node just before the node to be removed or inserted is known because of other work done on the chain.

Two chains may be combined into one by linking the last node of one chain to the first node of the second. If we know both the first and last nodes of a chain, this combining is done in $O(1)$ time. To combine two array linear lists into one, we must copy the second into the array used by the first. This copying takes $\Theta(\text{size of second list})$ time. When we know the “just before node,” the remove and insert operations of a chain run in $O(1)$ time; the complexity of these operations remains $O(\text{list size})$ when an array representation is used.

EXERCISES

1. Let $L = (a, b, c, d, e)$ be a linear list that is represented as a chain. Draw figures similar to Figure 6.1 showing the chain following each operation in the operation sequence: initial state, `insert(0, f)`, `insert(3, g)`, `insert(7, h)`, `erase(0)`, `erase(4)`.

2. Write code for the method `chain<T>::setSize(int theSize)` that makes the list size equal to `theSize`. If the original list size was less than the new one, `NULL` elements are added at the right end, and if the original size was more than the new one, the extra elements are removed from the right end. What is the complexity of your method? Test your code.
3. Write code for the method `chain<T>::set(theIndex, theElement)` that sets the element whose index is `theIndex` to `theElement`. Throw an exception in case `theIndex` is out of range. What is the complexity of your method? Test your code.
4. Write code for the method `chain<T>::removeRange(fromIndex, toIndex)` that removes all elements in the specified index range. What is the complexity of your method? Test your code.
5. Write code for the method `chain<T>::lastIndexOf(theElement)` that returns the index of the rightmost occurrence of `theElement`. A `-1` is returned in case `theElement` is not in the list. What is the complexity of your method? Test your code.
6. Overload the operator `[]` so that the expression `x[i]` returns a reference to the `i`th element of the chain `x`. If the chain doesn't have an `i`th element, an `illegalIndex` exception is to be thrown. The statements `x[i] = y` and `y = x[i]` should work as expected. Test your code.
7. Overload the operator `==` so that the expression `x == y` returns true iff the two chains `x` are `y` are equal (i.e., the `i`th elements of both chains are equal for all `i`). Test your code.
8. Overload the operator `!=` so that the expression `x != y` returns true iff the two chains `x` are `y` are not equal (see Exercise 7). Test your code.
9. Overload the operator `<` so that the expression `x < y` returns true iff the chain `x` is lexically smaller than the chain `y` (see Exercise 7). Test your code.
10. Write code for the method `chain<T>::swap(theChain)`, which swaps the elements of the chains `*this` and `theChain`. What is the time complexity of your method? Test your code.
11. Write a method to convert an array linear list into a chain. Your method is a member of neither `arrayList` nor `chain`. Use the `get` method of `arrayList` and the `insert` method of `chain`. What is the time complexity of your method? Test the correctness of your code.
12. Write a method to convert a linear list that is an instance of `chain` into an equivalent list that is an instance of `arrayList`. Your method is a member of neither `arrayList` nor `chain`.

- (a) First use only the `get` and `listSize` methods of `chain` and the `insert` method of `arrayList`. What is the time complexity of your method? Test the correctness of your code.
 - (b) Now use a chain iterator. What is the time complexity of the new code? Test your code using your own test data.
13. Add methods to `chain` to convert an `arrayList` to a `chain` and vice versa. Specifically, write a method `fromList(theList)` to convert the array linear list `theList` into a chain and another method `toList(theList)` to convert the chain `*this` into an array linear list `theList`. What is the time complexity of each method? Test the correctness of your code.
14. (a) Write code for the method `chain<T>::leftShift(i)` that shifts the list elements left by `i` positions. If `l = [0, 1, 2, 3, 4]`, then `l.leftShift(2)` results in `l = [2, 3, 4]`.
- (b) What is the time complexity of your method?
- (c) Test your code.
15. (a) Write code for the method `chain<T>::reverse`, which reverses the order of the elements in `*this`. Do the reversal in-place and do not allocate any new nodes.
- (b) What is the complexity of your method?
- (c) Test the correctness of your method by compiling and then executing the code. Use your own test data.
16. Write a nonmember method to reverse a chain. Use the member methods of `chain` to accomplish the reversal. What is the complexity of your method? Test the correctness of your method.
17. Let `a` and `b` be of type `extendedChain`.
- (a) Write a method `meld` to create a new extended chain `c` that contains elements alternately from `a` and `b`, beginning with the first element of `a`. If you run out of elements in one of `a` and `b`, then append the remaining elements of the other extended chain to `c`. The complexity of your code should be linear in the lengths of the `a` and `b`. Note that `meld` is not a member method of the class `extendedChain`.
 - (b) Show that your code has linear complexity.
 - (c) Test the correctness of your method by compiling and then executing the code. Use your own test data.
18. Write code for the method `chain<T>::meld`. This method is similar to the method `meld` of Exercise 17. However, `a` and `b` as well as the result are of type `chain<T>`. You should use the same physical nodes used by the chains `a` and `b` to create the resulting melded chain. Following a call to `meld`, the input chains `a` and `b` are empty.

- (a) Write the code for `meld`. The complexity of your code should be linear in the lengths of initial chains.
 - (b) Show that your code has linear complexity.
 - (c) Test the correctness of your code by compiling and then executing the code. Use your own test data.
19. Let `a` and `b` be of type `extendedChain`. Assume that the elements of `a` and `b` are of a type for which the relational operators `<`, `>`, `<=`, `>=`, `==`, and `!=` are defined. Further, assume that both `a` and `b` are in sorted order (i.e., nondecreasing from left to right).
- (a) Write a nonmember method `merge` to create a new sorted linear list `c` that contains all the elements in `a` and `b`.
 - (b) What is the complexity of your method?
 - (c) Test the correctness of your method by compiling and then executing the code. Use your own test data.
20. Redo Exercise 19 but this time write code for the method `chain<T>::merge`. You should use the same nodes as the two input chains use. Following the merge, the input chains are empty.
21. Let `c` be of type `extendedChain`.
- (a) Write a nonmember method `split` to create two extended chains `a` and `b`. `a` contains all elements in odd positions of `c`, and `b` contains the remaining elements. Your method should not change the extended chain `c`.
 - (b) What is the complexity of your code?
 - (c) Test the correctness of your method by compiling and then executing the code. Use your own test data.
22. Write code for the method `chain<T>::split` that is similar to the method `split` of Exercise 21. However, the new method `split` destroys the input chain `*this` and uses its nodes to construct the chains `a` and `b`.
23. In a circular shift operation, the elements of a linear list are rotated clockwise by a given amount. For example, when the elements of `L = [0, 1, 2, 3, 4]` are shifted circularly by 2, the result is `L = [2, 3, 4, 0, 1]`.
- (a) Write code for the method `extendedChain<T>::circularShift(i)`, which performs a circular shift by `i` positions.
 - (b) Test your code.

24. Let `theChain` be a chain. Suppose that as we move to the right, we reverse the direction of the chain pointers; therefore, when we are at node `p`, the chain is split into two chains. One is a chain that begins at `p` and goes to the last node of `theChain`. The other begins at the node `l` that precedes `p` in `theChain` and goes back to the first node of `theChain`. Initially, `p = theChain.firstNode` and `l = NULL`.
- (a) Draw a chain with six nodes and show the configuration when `p` is at the third node and `l` is at the second.
 - (b) Develop the class `moveLeftAndRightOnAChain`. The class constructor initializes the data members `l` and `p`. The public methods of `moveLeftAndRightOnAChain` are `moveRight`—move `l` and `p` one node right, `moveLeft`—move `l` and `p` one node left, `currentElement`—return the element at node `p`, and `previousElement`—return the element at node `l`.
 - (c) Test your codes using suitable data.
25. Use the ideas of Exercise 24 to obtain a new version of the class `chain` of Program 6.2. The new version should permit you to move back and forth on a chain efficiently and to perform the methods of `linearList` even though the chain may be split into two chains as described in Exercise 24. For this version, add the data members `l` and `p` as in Exercise 24 and add the following public methods:
- (a) `reset`—Set `p` to `firstNode` and `l` to `NULL`.
 - (b) `current()`—Return the element pointed to by `p`; throw an exception if the operation fails.
 - (c) `atEnd`—Return `true` if `p` is at the last element of the list; return `false` otherwise.
 - (d) `atFront`—Return `true` if `p` is at the first element of the list; return `false` otherwise.
 - (e) `moveToNext`—Move `p` and `l` one position right; throw an exception if the operation fails.
 - (f) `moveToPrevious`—Move `p` and `l` one position back; throw an exception if the operation fails.
- To implement the `insert`, `erase`, and `indexOf` methods efficiently, it will be useful to have another data member `currentElement` that gives you the index of the element to which `p` points (i.e., is it element 0, 1, 2, etc., of the list?). Test the correctness of your code using suitable test data.
26. Write code for the method `chain<T>::insertionSort`, which uses insertion sort (see Program 2.15) to reorder the chain elements into nondecreasing order.

Do not create new nodes or delete old ones. You may assume that the elements being sorted are of type for which the relational operators ($<$, $>$, etc.) are defined.

- (a) What is the worst-case time complexity of your method? How much time does your method need if the elements are already in sorted order?
 - (b) Test the correctness of your method by compiling and then executing the code. Use your own test data.
27. Do Exercise 26 for the following sort methods (see Chapter 2 for descriptions):
- (a) Bubble sort.
 - (b) Selection sort.
 - (c) Count or rank sort.

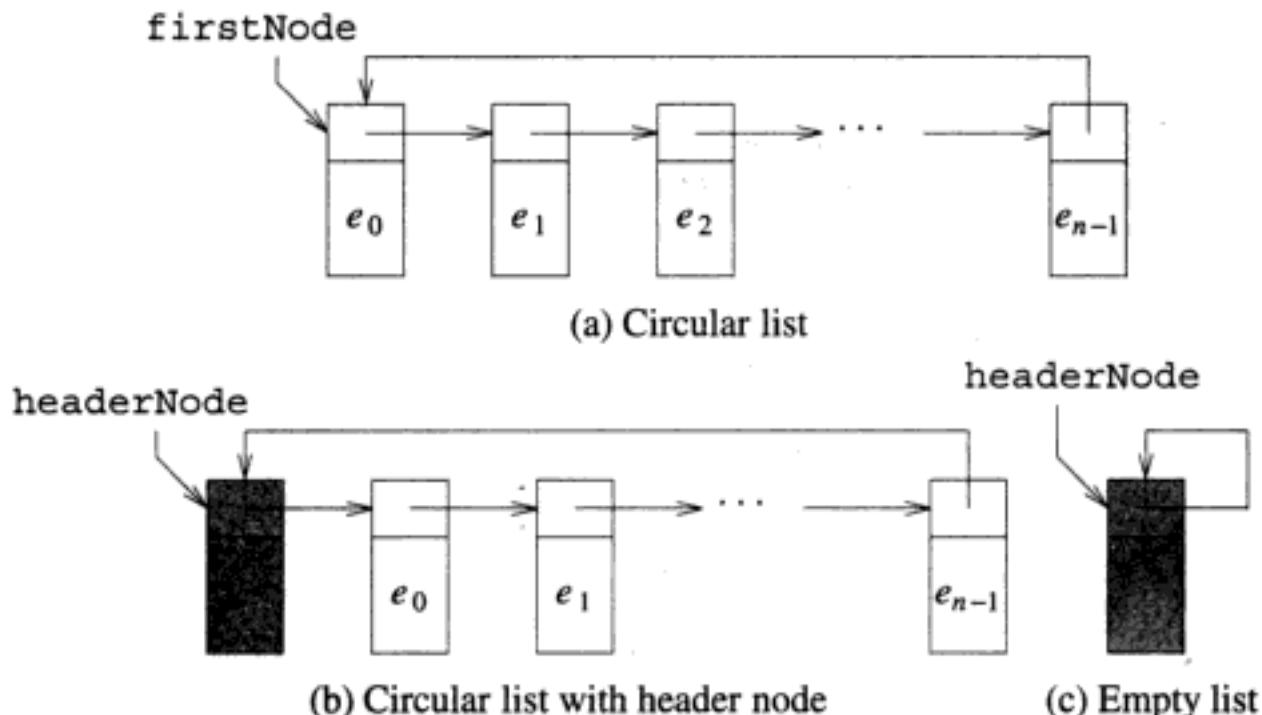
6.2 CIRCULAR LISTS AND HEADER NODES

Application codes that result from the use of chains can often be simplified and made to run faster by doing one or both of the following: (1) represent the linear list as a **singly linked circular list** (or simply **circular list**), rather than as a chain, and (2) add an additional node, called the **header node**, at the front. A circular list is obtained from a chain by linking the last node back to the first as in Figure 6.6(a). Figure 6.6(b) shows a nonempty circular list with a header node, and Figure 6.6(c) shows an empty circular list with a header node.

The use of header nodes is a very common practice when linked lists are used, as their presence generally leads to simpler and faster programs. Program 6.13 gives the constructor and `indexOf` methods for the class `circularListWithHeader`, which represents a linear list as a circular list with a header node. The constructor creates the configuration for an empty list (Figure 6.6(c)). The complexity of the constructor is $\Theta(1)$ and that of `indexOf` is $O(\text{listSize})$. Although `chain<T>::indexOf` and `circularListWithHeader<T>::indexOf` have the same complexity, the code for the latter method is slightly simpler. Since `circularListWithHeader<T>::indexOf` avoids the check `currentNode != NULL` that is made by `chain<T>::indexOf` on each iteration of its `while` loop, `circularListWithHeader<T>::indexOf` will run slightly faster than `chain<T>::indexOf` except possibly when we are looking for an element near the left end of the chain.

EXERCISES

28. Compare the run-time performance of the `indexOf` methods of Programs 6.6 and 6.13. Do this for both worst-case and average run times using linear lists of size 100; 1000; 10,000; and 100,000. Present your times in tabular form and in graph form.

**Figure 6.6** Circular linked lists

29. Develop the class `circularList`. Objects of this type are circular linked lists, as in Figure 6.6, except the lists do not have a header node. You must implement all the methods defined for the classes `chain` (Section 6.1.3) and `extendedChain` (Section 6.1.5). What is the time complexity of each method? Test the correctness of your code.
30. Do Exercise 15 using circular lists instead of chains.
31. Do Exercise 16 using circular lists instead of chains.
32. Do Exercise 17 using circular lists instead of chains.
33. Do Exercise 19 using circular lists instead of chains.
34. Do Exercise 20 using circular lists instead of chains.
35. Do Exercise 21 using circular lists instead of chains.
36. Do Exercise 22 using circular lists instead of chains.
37. Let `x` point to an arbitrary node in a circular list.
 - (a) Write a method to remove the element in node `x`. *Hint:* Since we do not know which node precedes `x`, it is difficult to remove the node `x` from the list; however, to remove the element in `x`, it is sufficient to replace the

```
template<class T>
circularListWithHeader<T>::circularListWithHeader()
{// Constructor.
    headerNode = new chainNode<T>();
    headerNode->next = headerNode;
    listSize = 0;
}

template<class T>
int circularListWithHeader<T>::indexOf(const T& theElement) const
{// Return index of first occurrence of theElement.
// Return -1 if theElement not in list.

    // Put theElement in header node
    headerNode->element = theElement;

    // search the chain for theElement
    chainNode<T>* currentNode = headerNode->next;
    int index = 0; // index of currentNode
    while (currentNode->element != theElement)
    {
        // move to next node
        currentNode = currentNode->next;
        index++;
    }

    // make sure we found matching element
    if (currentNode == headerNode)
        return -1;
    else
        return index;
}
```

Program 6.13 Searching a circular linked list that has a header node

data field (i.e., `element`) of `x` by the data field of the node `y` that follows it and then remove the node `y`. When the element in the last node is removed, the element in the first node becomes the last element.

- (b) What is the complexity of your method?
- (c) Test the correctness of your method by compiling and then executing the code. Use your own test data.

38. Complete the class `circularListWithHeader` by writing code for the remaining methods of `extendedLinearList`. What is the time complexity of each method? Test the correctness of your code.
39. Do Exercises 15 and 16 using circular lists with header nodes instead of chains.
40. Do Exercises 17 and 18 using circular lists with header nodes instead of chains.
41. Do Exercises 19 and 20 using circular lists with header nodes instead of chains.
42. Do Exercises 21 and 22 using circular lists with header nodes instead of chains.

6.3 DOUBLY LINKED LISTS

For most applications of linear lists, the chain and/or circular list representations are adequate. However, in some applications it is useful to have a pointer from each element to both the next and previous elements. A **doubly linked list** is an ordered sequence of nodes in which each node has two pointers: `next` and `previous`. The `previous` pointer points to the node (if any) on the left, and the `next` pointer points to the node (if any) on the right. Figure 6.7 shows the doubly linked list representation of the linear list (1, 2, 3, 4).

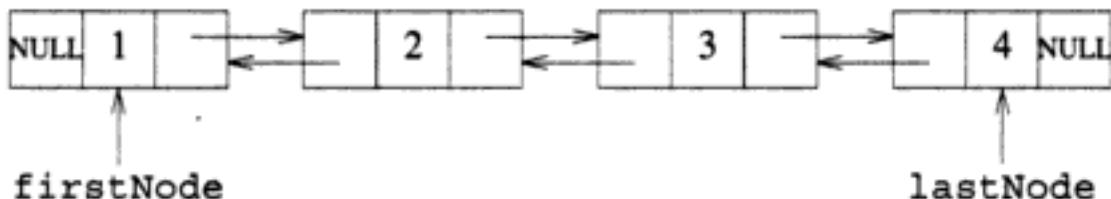


Figure 6.7 A doubly linked list

When defining the class `doublyLinkedList`, we use two data members `firstNode` and `lastNode` that, respectively, point to the left-most and right-most nodes of the doubly linked list (see Figure 6.7). A doubly linked list with just one element or node `p` has `firstNode = lastNode = p`, whereas `firstNode = lastNode = NULL` for an empty doubly linked list. These conventions are similar to those used for an extended chain (Program 6.12). When a doubly linked list is used, we find the `index`th element by moving from left to right when `index < listSize/2` and from right to left otherwise. Exercise 43 asks you to develop the code for the class `doublyLinkedList`.

We can enhance doubly linked lists by adding a header node at the left and/or right ends and by making them circular lists. In a nonempty circular doubly linked list, `firstNode.previous` is a pointer to the right-most node (i.e., `firstNode.previous = lastNode`), and `lastNode.next` is a pointer to the left-most node. So

we can dispense with one of the variables `firstNode` and `lastNode` and simply keep track of the list using the remaining variable.

EXERCISES

43. Develop the class `doublyLinkedList`. Objects of this type are doubly linked lists with no header node. You must implement all the methods defined for the class `extendedChain` (Section 6.1.5). What is the time complexity of each method? Test the correctness of your code.
44. Write a method to join two doubly linked lists into a single doubly linked list. In a join the elements of the second list are appended to the end of those of the first list; the join is destructive in the sense that following the join, the second list becomes empty. Test your code.
45. Do Exercises 15 and 16 using doubly linked lists instead of chains.
46. Do Exercises 17 and 18 using doubly linked lists instead of chains.
47. Do Exercises 19 and 20 using doubly linked lists instead of chains.
48. Do Exercises 21 and 22 using doubly linked lists instead of chains.
49. Develop the class `doubleCircularList`. Objects of this type are doubly linked circular lists with no header node. You must implement all the methods defined for the class `extendedChain` (Section 6.1.5). What is the time complexity of each method? Test the correctness of your code.
50. Do Exercises 15 and 16 using doubly linked circular lists.
51. Do Exercise 44 using doubly linked circular lists.
52. Do Exercises 17 and 18 using doubly linked circular lists.
53. Do Exercises 19 and 20 using doubly linked circular lists.
54. Do Exercises 21 and 22 using doubly linked circular lists.
55. Do Exercise 49 using a header node for the doubly linked circular list. Compare the run time of your class with that of an equivalent class that uses the STL container class `list` much in the same way that `vectorList` (Program 5.12) uses a vector to implement an array linear list. Perform an experiment similar to that done in Section 6.1.6.
56. Do Exercises 15 and 16 using doubly linked circular lists with header nodes.
57. Do Exercise 44 using doubly linked circular lists with header nodes.
58. Do Exercises 17 and 18 using doubly linked circular lists with header nodes.

59. Do Exercises 19 and 20 using doubly linked circular lists with header nodes.
60. Do Exercises 21 and 22 using doubly linked circular lists with header nodes.
61. For the doubly linked circular list with header node class of Exercise 55, develop a bidirectional iterator. Test the correctness of your code using suitable test data.

6.4 GLOSSARY OF LINKED LIST TERMS

This chapter introduced the following important concepts:

- *chain*. A chain is a singly linked list of nodes. Let x be a chain. x is empty iff $x.\text{firstNode} = \text{NULL}$. If x is not empty, then $x.\text{firstNode}$ points to the first node in the chain. The first node links to the second; the second to the third; and so on. The link (i.e., `next`) field of the last node is `NULL`.
- *Singly linked circular list*. This type of list differs from a chain only in that now the last node links back to the first. When the circular list x is empty, $x.\text{firstNode} = \text{NULL}$.
- *Header node*. A header node is an additional node introduced into a linked list. The use of this additional node generally results in simpler programs, as we can often avoid treating the empty list as a special case. When a header node is used, every list (including the empty list) contains at least one node. (i.e., the header node).
- *Doubly linked list*. A doubly linked list consists of nodes ordered from left to right. Nodes are linked from left to right using a pointer field (say) `next`. The right-most node has this field set to `NULL`. Nodes are also linked from right to left using a pointer field (say) `previous`. The left-most node has this field set to `NULL`.
- *Circular doubly linked list*. This type of list differs from a doubly linked list only in that now the left-most node uses its `previous` field to point to the right-most node and the right-most node uses its `next` field to point to the left-most node.

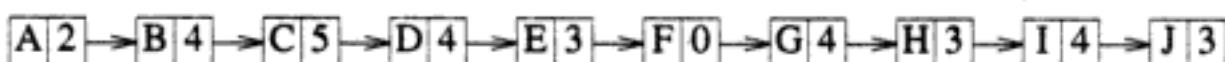
6.5 APPLICATIONS

6.5.1 Bin Sort

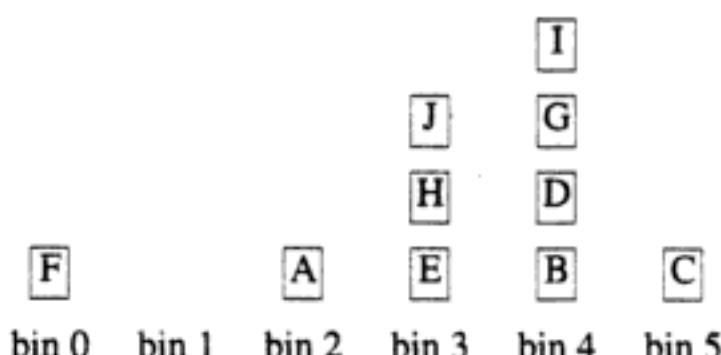
Suppose that a chain is used to maintain a list of students in a class. Each node has fields for the student's name, Social Security number, score on each assignment and test, and weighted aggregate score of all assignments and tests. Assume that all

scores are integers in the range 0 through 100. We are to sort the nodes in order of the aggregate score. This sort takes $O(n^2)$ time (n is the number of students in the class) if we use one of the sort methods of Chapter 2. A faster way to accomplish the sort is to use **bin sort**. In a bin sort the nodes are placed into bins, each bin containing nodes with the same score. Then we combine the bins to create a sorted chain.

Figure 6.8(a) shows a sample chain with 10 nodes. This figure shows only the name and score fields of each node. The first field is the name, and the second is the score. For simplicity, we assume that each name is a single character and that the scores are in the range 0 through 5.



(a) Input chain



(b) Nodes in bins



(c) Sorted chain

Figure 6.8 Bin sort example

We will need six bins, one for each of the possible score values 0 through 5. Figure 6.8(b) shows the 10 nodes distributed into bins by score. We can obtain this distribution by moving down the chain and examining the nodes one at a time. When a node is examined, it is placed into the bin that corresponds to its score. So the first node is placed into bin 2, the second into bin 4, and so forth. Now if we collect the nodes from the bins, beginning with those in bin 0, we will have a sorted list as shown in Figure 6.8(c).

To implement the bins, we note that each bin is a linear list of nodes. The number of nodes in a bin may vary from 0 to as many as n . Before we begin the

node distribution step, all bins are empty.

For bin sort we need to be able to (1) move down the input chain deleting nodes from this chain and adding them to the list for the appropriate bin and (2) collect and concatenate lists from the bins into a single sorted chain. If the input chain is of type `chain` (Program 6.2), we can do (1) by successively deleting the first element from the chain and inserting it as the first element in the appropriate bin list; we can do (2) by deleting the elements from the bins (beginning with the last bin) and inserting them at the front of an initially empty chain.

Program 6.14 gives a possible struct definition for student records. Our intent is to use chains of type `chain<studentRecord>`. In a realistic situation, `studentRecord` would contain several additional data members. The operators `!=` and `<<` have been overloaded, as these operators are used by the class `chain`.

```
struct studentRecord
{
    int score;
    string* name;
    int operator !=(studentRecord x) const
        {return (score != x.score);}
};

ostream& operator<<(ostream& out, const studentRecord& x)
    {out << x.score << ' ' << *x.name << endl; return out;}
```

Program 6.14 Possible struct for bin sort chain elements

An alternative to overloading `!=` is to provide a conversion from the type `studentRecord` to a numeric type that can be used for comparison and other purposes. For example, we can overload the type conversion operator `int()` as shown in Program 6.15. Operators such as the arithmetic and relational operators `+`, `/`, `<=` and `!=` that are not explicitly defined on the type `studentRecord` now can complete successfully by first performing a conversion to the type `int`. This solution is somewhat more general than our earlier one in which we explicitly overloaded the operator `!=`, as now the code works even when the class `chain` includes methods that perform other operations on `this->element`.

We can combine both overloading approaches so that type conversion to `int` occurs only when the operator will fail without the type conversion. So we may, for example, use the definition of Program 6.16. Type conversion to `int` now will take place only for operators other than `!=` and `<<`.

Program 6.17 gives the code for the bin sort method. This code uses a chain for each bin. Although we could have represented each bin as an array list, we have used a chain because we plan to develop another bin sort method that is a member

```
struct studentRecord
{
    int score;
    string* name;

    operator int() const {return score;}
        // type conversion from studentRecord to int
};

ostream& operator<<(ostream& out, const studentRecord& x)
{out << x.score << ' ' << *x.name << endl; return out;}
```

Program 6.15 An alternative definition of studentRecord

```
struct studentRecord
{
    int score;
    string* name;

    int operator !=(studentRecord x) const
    {return (score != x.score || name != x.name);}
    operator int() const {return score;}
};

ostream& operator<<(ostream& out, const studentRecord& x)
{out << x.score << ' ' << *x.name << endl; return out;}
```

Program 6.16 Yet another definition of studentRecord

of chain. In this new method, it is more efficient to use chains rather than array lists because the input and output for the sort is a chain.

For the complexity analysis, we first note that binSort (Program 6.17) could terminate prematurely because of an exception. For example, the statement

```
bin = new chain<studentRecord> [range + 1];
```

could fail for lack of sufficient memory. If this statement fails, the method terminates in $\Theta(1)$ time. Assume that no exception occurs while the method executes. Now the first for loop takes $\Theta(\text{range})$ time. Each get, insert and erase performed in the remaining two for loops takes $\Theta(1)$ time. Therefore, the complexity of the second for loop is $\Theta(n)$ where n is the size of the input chain, the complexity of the third

```
void binSort(chain<studentRecord>& theChain, int range)
{// Sort by score.

    // initialize the bins
    chain<studentRecord> *bin;
    bin = new chain<studentRecord> [range + 1];

    // distribute student records from theChain to bins
    int numberofElements = theChain.size();
    for (int i = 1; i <= numberofElements; i++)
    {
        studentRecord x = theChain.get(0);
        theChain.erase(0);
        bin[x.score].insert(0,x);
    }

    // collect elements from bins
    for (int j = range; j >= 0; j--)
        while (!bin[j].empty())
        {
            studentRecord x = bin[j].get(0);
            bin[j].erase(0);
            theChain.insert(0,x);
        }

    delete [] bin;
}
```

Program 6.17 Bin sort using the methods of chain

for loop is $\Theta(n+range)$, and the overall complexity of `binSort` (when no exception is thrown) is $\Theta(n+range)$. Accounting for the possibility of an exception or error, the overall complexity is $O(n+range)$.

Bin Sort as a Method of a chain

Efficiency-conscious readers have probably noticed that we can avoid much of the work done by the method `binSort` (Program 6.17) by developing `binSort` as a method of `chain`. This approach enables us to avoid the calls to `new` made by the invocations of `insert` in Program 6.17; the calls to `delete` made by the invocations of `erase` also may be avoided. Further, by keeping track of the front and end of each bin chain, we can concatenate the bin chains in the “collection phase,” as shown in

Program 6.18.

The chain for each bin begins with the node at the bottom of the bin and goes to the node at the top of the bin. Each chain has two pointers, `bottom` and `top`, to it. `bottom[theBin]` points to the node at the bottom of bin `theBin`, while `top[theBin]` points to the node at the top of this bin. The initial configuration of empty bins is represented by `bottom[theBin] = NULL` for all bins. As chain nodes are examined, they are added to the top of the required bin (first `for` loop of Program 6.18). The second `for` loop examines the bins beginning with bin 0 and concatenates the chains in the nonempty bins to form the sorted chain.

For the time complexity of `binSort`, assume that no exception is thrown. The creation and initialization of the arrays `bottom` and `top` as well as the second `for` loop take $\Theta(\text{range})$ time, and the first `for` loop takes $\Theta(n)$ time. Allowing for the possibility that an exception or error is thrown, the overall complexity is $O(n+\text{range})$.

Notice that `binSort` (Program 6.18) does not change the relative order of elements that have the same score. For example, suppose that E, G, and H all have the score 3 and that E comes before G, which comes before H in the input chain. In the sorted chain, too, E comes before G, which comes before H. A sort method that preserves the relative order of elements with the same value is called a **stable sort**.

6.5.2 Radix Sort

The bin sort method of Section 6.5.1 may be extended to sort, in $\Theta(n)$ time, n integers in the range 0 through $n^c - 1$ where $c \geq 0$ is an integer constant. Notice that if we use `binSort` with `range = n^c` , the sort complexity will be $\Theta(n + \text{range}) = \Theta(n^c)$. Instead of using `binSort` directly on the numbers to be sorted, we will decompose these numbers using some radix r . For example, the number 928 decomposes into the digits 9, 2, and 8 using the radix 10 (i.e., $928 = 9*10^2 + 2*10^1 + 8*10^0$). The most significant digit is 9, and the least significant digit is 8; the ones digit is 8, the tens digit is 2, and the hundreds digit is 9. The number 3725 has the radix 10 decomposition 3, 7, 2, and 5; using the radix 60 instead, the decomposition is 1, 2, and 5 (i.e., $(3725)_{10} = (125)_{60}$). In a **radix sort** we decompose the numbers into digits using some radix r and then sort by digits.

Example 6.1 Suppose that we are sorting 10 integers in the range 0 through 999. If we use `binSort` with `range = 1000`, then the bin initialization takes 1000 steps, the node distribution takes 10 steps, and collecting from the bins takes 1000 steps. The total step count is 2010. Another approach is

1. Use `binSort` to sort the 10 numbers by their least significant digit (i.e., the ones digit). Since each digit ranges from 0 through 9, we have `range = 10`. Figure 6.9(a) shows a sample 10-number chain, and Figure 6.9(b) shows the chain sorted by least significant digit.

```

template<class T>
void chain<T>::binSort(int range)
{// Sort the nodes in the chain.
    // create and initialize the bins
    chainNode<T> **bottom, **top;
    bottom = new chainNode<T>* [range + 1];
    top = new chainNode<T>* [range + 1];
    for (int b = 0; b <= range; b++)
        bottom[b] = NULL;

    // distribute to bins
    for (; firstNode != NULL; firstNode = firstNode->next)
        {// add firstNode to proper bin
            int theBin = firstNode->element; // type conversion to int.
            if (bottom[theBin] == NULL) // bin is empty
                bottom[theBin] = top[theBin] = firstNode;
            else
                {// bin not empty
                    top[theBin]->next = firstNode;
                    top[theBin] = firstNode;
                }
        }

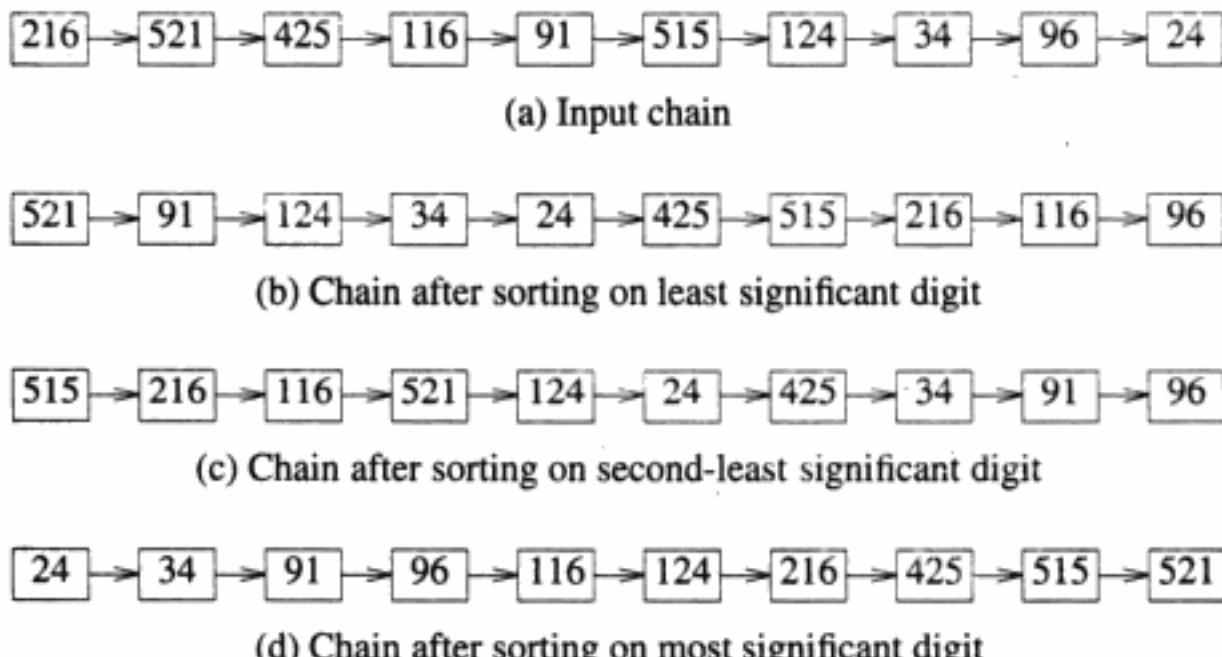
    // collect from bins into sorted chain
    chainNode<T> *y = NULL;
    for (int theBin = 0; theBin <= range; theBin++)
        if (bottom[theBin] != NULL)
            {// bin not empty
                if (y == NULL) // first nonempty bin
                    firstNode = bottom[theBin];
                else // not first nonempty bin
                    y->next = bottom[theBin];
                y = top[theBin];
            }
    if (y != NULL)
        y->next = NULL;

    delete [] bottom;
    delete [] top;
}

```

Program 6.18 Bin sort as a method of chain

2. Use `binSort` to sort the chain from (1) by the next digit (i.e., the tens digit). Again, `range = 10`. Since bin sort is a stable sort, elements that have the same second digit remain sorted by the least significant digit. As a result, the chain is now sorted by the two least significant digits. Figure 6.9(c) shows our chain following this sort.
 3. Use `binSort` to sort the chain from (2) by the next digit (i.e., the hundreds digit). (For numbers smaller than 100, the hundreds digit is 0.) Since the sort on the hundreds digit is stable, elements with the same hundreds digit remain sorted on the remaining two digits. As a result, the chain is sorted on the three least significant digits. Figure 6.9(d) shows the chain following this sort.
-

**Figure 6.9** Radix sort with $r = 10$ and $d = 3$

The preceding sorting scheme describes a radix 10 sort. The numbers to be sorted are decomposed into their decimal (or base 10) digits, and the numbers are sorted on these digits. Since each number has at most three digits, three sort passes are made. Each sort pass uses a bin sort with `range = 10`. In each of these three bin sorts, we spend 10 steps in initializing the bins, 10 in distributing the records, and 10 in bin collection. The total number of steps is 90, which is less than when the 10 numbers are sorted using a single bin sort with `range = 1000`. The single bin sort scheme is really a radix sort with $r = 1000$. ■

Example 6.2 Suppose that 1000 integers in the range 0 through $10^6 - 1$ are to be sorted. Using a radix of $r = 10^6$ corresponds to using `binSort` directly on the numbers and takes 10^6 steps to initialize the bins, 1000 steps to distribute the numbers into bins, and another 10^6 to collect from the bins. The total number of steps is therefore 2,001,000. With $r = 1000$, the sort proceeds as follows:

1. Sort using the three least significant decimal digits of each number and use `range = 1000`.
2. Sort the result of (1) using the next three decimal digits of each number.

Each of the preceding sorts takes 3000 steps, so the sort is accomplished in a total of 6000 steps. When $r = 100$ is used, three bin sorts on pairs of decimal digits are performed. Each of these sorts takes 1200 steps, and the total number of steps needed for the sort becomes 3600. If we use $r = 10$, six bin sorts will be done, one on each decimal digit. The total number of steps will be $6(10 + 1000 + 10) = 6120$. For our example we expect radix sort with $r = 100$ to be most efficient. ■

We can decompose a number into digits by using the division and mod operators. If we are performing a radix 10 decomposition, then the radix 10 digits may be computed (from least significant to most significant) using the following expressions:

$$x \% 10; \quad (x \% 100) / 10; \quad (x \% 1000) / 100; \quad \dots$$

When $r = 100$, these expressions become

$$x \% 100; \quad (x \% 10000) / 100; \quad (x \% 1000000) / 10000; \quad \dots$$

For a general radix r , the expressions are

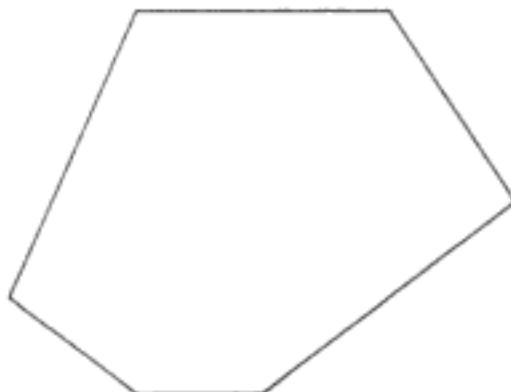
$$x \% r; \quad (x \% r^2) / r; \quad (x \% r^3) / r^2; \quad \dots$$

When we use the radix $r = n$ to decompose n integers in the range 0 through $n^c - 1$, the number of digits is c . So the n numbers can be sorted using c bin sort passes with `range = n`. The time needed for the sort is $\Theta(cn) = \Theta(n)$, as c is a constant.

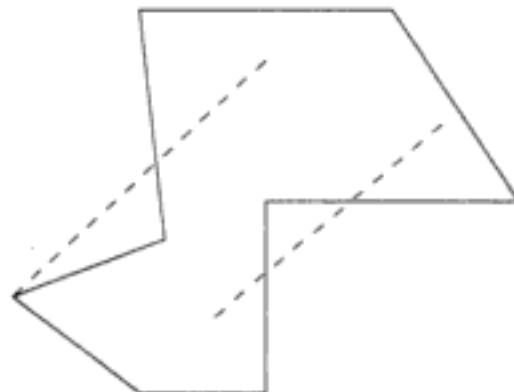
6.5.3 Convex Hull

A **Polygon** is a closed planar figure with three or more straight edges. The polygon of Figure 6.10(a) has six edges, and that of Figure 6.10(b) has eight. A polygon **contains** all points that are either on its edges or inside the region it encloses. A polygon is **convex** iff all line segments that join two points on or in the polygon

include no point that is outside the polygon. The polygon of Figure 6.10(a) is convex, while that of Figure 6.10(b) is not. Figure 6.10(b) shows two line segments (broken lines) whose endpoints are on or in the polygon. Both of these segments contain points that are outside the polygon.



(a) Convex polygon



(b) Nonconvex polygon

Figure 6.10 Convex and nonconvex polygons

The **convex hull** of a set S of points in the plane is the smallest convex polygon that contains all these points. The vertices (i.e., corners) of this polygon are the **extreme points** of S . Figure 6.11 shows 13 points in the plane. The convex hull is the polygon defined by the solid lines. The extreme points have been identified by circles. When all points of S lie on a straight line (i.e., they are collinear), we have a degenerate case for which the convex hull is defined to be the smallest straight line that includes all the points.

The problem of finding the convex hull of a set of points in the plane is a fundamental problem in computational geometry. The solutions to several other problems in computational geometry (e.g., find the smallest rectangle that encloses a set of points in the plane) require the computation of the convex hull. In addition, the convex hull finds application in image processing and statistics.

Suppose we pick a point X in the interior of the convex hull of S and draw a vertical line downwards from X (Figure 6.12(a)). Exercise 67 describes how we can select the point X . Let a_i be the (polar) angle made by this line and the line from X to the i th point of S . a_i is measured by going counterclockwise from a point on the vertical line to the line from X to the i th point. Figure 6.12(a) shows a_2 . Now let us arrange the points of S into nondecreasing order of a_i . Points with the same polar angle are ordered by distance from X . In Figure 6.12(a) the points have been numbered 1 through 13 in the stated order.

A counterclockwise sweep of the vertical line downwards from X encounters the extreme points of S in order of the polar angle a_i . If u , v , and w are three consecutive extreme points in counterclockwise order, then the counterclockwise

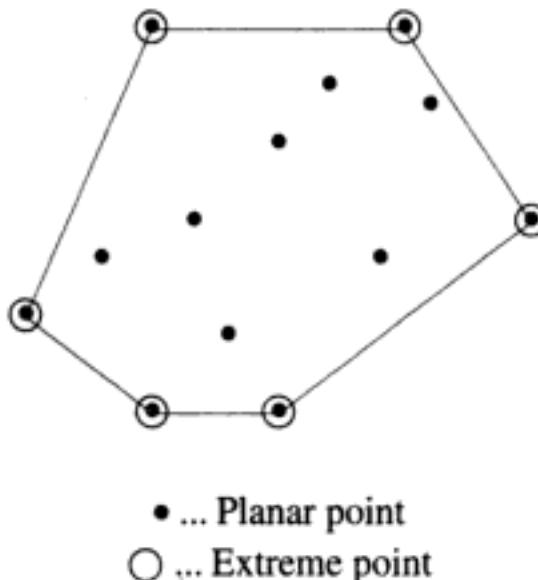


Figure 6.11 Convex hull of planar points

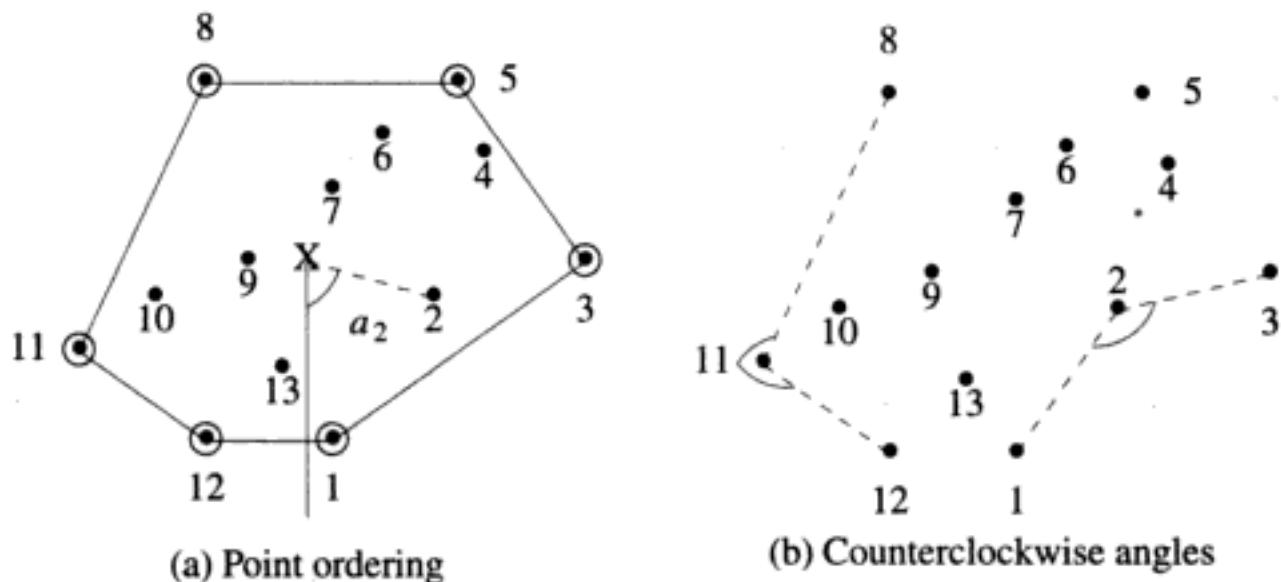


Figure 6.12 Identifying extreme points

angle made by the line segments from u to v and w to v is more than 180 degrees. (Figure 6.12(b) shows the counterclockwise angle made by points 8, 11, and 12.) When the counterclockwise angle made by three consecutive points in the polar order is less than or equal to 180 degrees, then the second of these points is not an extreme point. Notice that when the angle made by u , v , and w is less than 180 degrees, if we walk from u to v to w , we make a right turn at v . When we

walk counterclockwise around a convex polygon, all our turns are left turns. The observations made so far result in the algorithm of Figure 6.13, which finds the extreme points and convex hull of S .

Step 1: [Handle degenerate cases]

If S has fewer than three points, return S .

If all points lie on a straight line, compute the endpoints of the smallest line that includes all points of S and return these two points.

Step 2: [Sort by polar angle]

Find a point X that is inside the convex hull of S .

Sort S by polar angle and within polar angle by distance from X .

Create a doubly linked circular list of points using the above order.

Let `right` link to the next point in the order and `left` link to the previous point.

Step 3: [Eliminate nonextreme points]

Let p be the point that has the smallest y -coordinate (break a tie, if any, by selecting the one with largest x -coordinate).

```
for (x = p, rx = point to the right of x; x != rx; )
```

```
{
```

```
    rrx = point to the right of rx;
```

```
    if (angle formed by x, rx, and rrx is ≤ 180 degrees)
```

```
{
```

```
        delete rx from the list;
```

```
        rx = x; x = point on left of rx;
```

```
}
```

```
    else {x = rx; rx = rrx;}
```

```
}
```

Figure 6.13 Pseudocode to find the convex hull of S

Step 1 of the algorithm handles the degenerate cases when the number of points in S is 0 or 1, as well as when all points of S are collinear. This step can be done in $O(n)$ time where n is the number of points in S . For the collinearity test, we select any two points and compute the equation of the line through them. Next we examine the remaining $n - 2$ points and determine whether they lie on this line. During this process we can also determine the endpoints of the shortest line that includes all points in case they are collinear.

In step 2 the points are ordered by polar angle and collected into a doubly linked list because in step 3 we will be eliminating points that are not extreme points and also moving backwards on the list. Both operations are straightforward in a doubly

linked list. Exercise 67 asks you to explore the use of a singly linked list. Because of the sort, this step takes $O(n^2)$ time if we use any of the sorts from Chapter 2. In Chapters 9 and 14, we will see that we can sort in $O(n \log n)$ time. As a result, the complexity of step 2 is counted as $O(n \log n)$.

In step 3 we repeatedly examine sets of three consecutive points in counterclockwise order and check whether the angle they make is less than or equal to 180 degrees. If it is, then the middle point rx is not an extreme point and is eliminated. If the angle exceeds 180 degrees, rx may or may not be an extreme point and we advance x to the next vertex rx . When the **for** loop is exited, every point x on the doubly linked circular list satisfies the property that the angle made by x , rx , and rrx exceeds 180 degrees. Hence all of these points are extreme points. By going around the list using the **right** fields, we traverse the boundary of the convex hull in counterclockwise order. We begin at the point with lowest y , as this point must be in the convex hull.

For the complexity of step 3, we note that following each angle check in the **for** loop either (1) a vertex rx is eliminated and x is moved back one node on the list or (2) x is moved forward on the list. Since the number of eliminated vertices is $O(n)$, x can be moved back at most a total of $O(n)$ nodes. Hence we can be in case (2) only $O(n)$ times. So the **for** loop is iterated $O(n)$ times. Since an angle check takes $\Theta(1)$ time, the complexity of step 3 is $O(n)$. As a result, we can find the convex hull of n points in $O(n \log n)$ time.

6.5.4 Union-Find Problem

Equivalence Classes

Suppose we have a set $U = 1, 2, \dots, n$ of n elements and a set $R = (i_1, j_1), (i_2, j_2), \dots, (i_r, j_r)$ of r relations. The relation R is an **equivalence relation** iff the following conditions are true:

- $(a, a) \in R$ for all $a \in U$ (the relation is reflexive).
- $(a, b) \in R$ iff $(b, a) \in R$ (the relation is symmetric).
- $(a, b) \in R$ and $(b, c) \in R$ imply that $(a, c) \in R$ (the relation is transitive).

Often when we specify an equivalence relation R , we omit some of the pairs in R . The omitted pairs may be obtained by applying the reflexive, symmetric, and transitive properties of an equivalence relation.

Example 6.3 Suppose $n = 14$ and $R = \{(1,11), (7,11), (2,12), (12,8), (11,12), (3,13), (4,13), (13,14), (14,9), (5,14), (6,10)\}$. We have omitted all pairs of the form (a, a) because these pairs are implied by the reflexive property. Similarly, we have omitted all symmetric pairs. Since $(1,11) \in R$, the symmetric property requires $(11,1) \in R$. Other omitted pairs are obtained by applying the transitive property. For example, $(7,11)$ and $(11,12)$ imply $(7,12)$. ■

Two elements a and b are equivalent if $(a, b) \in R$. An **equivalence class** is defined to be a maximal set of equivalent elements. *Maximal* means that no element outside the class is equivalent to an element in the class. Since it is not possible for an element to be in more than one equivalence class, an equivalence relation partitions the universe U into disjoint classes.

Example 6.4 Consider the equivalence relation of Example 6.3. Since elements 1 and 11, and 11 and 12 are equivalent, elements 1, 11, and 12 are equivalent. They are therefore in the same class. These three elements do not, however, form an equivalence class, as they are equivalent to other elements (e.g., 7). So $\{1, 11, 12\}$ is not a maximal set of equivalent elements. The set $\{1, 2, 7, 8, 11, 12\}$ is an equivalence class. The relation R defines two other equivalence classes: $\{3, 4, 5, 9, 13, 14\}$ and $\{6, 10\}$. Notice that the three equivalence classes are disjoint. ■

In the **offline equivalence class** problem, we are given n and R and we need to determine the equivalence classes. From the definition of an equivalence class, it follows that each element is in exactly one equivalence class. In the **online equivalence class** problem, we begin with n elements, each in a separate equivalence class. We are to process a sequence of the operations: (1) `combine(a,b)` ... combines the equivalence classes that contain elements a and b into a single class and (2) `find(theElement)` ... determines the class that currently contains element `theElement`. The purpose of the find operation is to determine whether two elements are in the same class. Hence the find operation is to be implemented to return the same answer for elements in the same class and different answers for elements in different classes.

We can write the combine operation in terms of two finds and a `unite` (or union) that actually takes two different classes and makes one. So `combine(a,b)` is equivalent to

```
classA = find(a);
classB = find(b);
if (classA != classB)
    unite(classA, classB);
```

Notice that with the find and union operations, we can add new relations to R . For instance, to add the relation (a, b) , we determine whether a and b are already in the same class. If they are, then the new relation is redundant. If they aren't, then we perform a `unite` on the two classes that contain a and b .

In this section we are concerned primarily with the online equivalence problem, which is more commonly known as the **union-find** problem. Although the solutions developed in this section are rather simple, they are not the most efficient. Faster solutions are developed in Section 11.9.2. A fast solution for the offline equivalence problem is developed in Section 8.5.5.

Applications

The following examples show how a machine-scheduling problem and a circuit-wiring problem may be modeled as online equivalence class problems. A version of the circuit wiring problem may be modeled as an offline equivalence class problem.

Example 6.5 A certain factory has a single machine that is to perform n tasks. Task i has an integer release time r_i and an integer deadline d_i . The completion of each task requires one unit of time on this machine. A **feasible schedule** is an assignment of tasks to time slots on the machine such that task i is assigned to a time slot between its release time and deadline and no slot has more than one task assigned to it.

Consider the following four tasks:

Task	A	B	C	D
Release time	0	0	1	2
Deadline	4	4	2	3

Tasks A and B are released at time 0, task C is released at time 1, and task D is released at time 2. The following task-to-slot assignment is a feasible schedule: do task A from 0 to 1; task C from 1 to 2; task D from 2 to 3; and task B from 3 to 4 (see Figure 6.14).

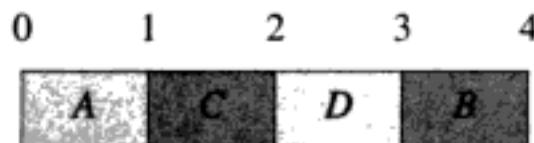


Figure 6.14 A schedule for four tasks

An intuitively appealing method to construct a schedule is

1. Sort the tasks into nonincreasing order of release time.
2. Consider the tasks in this nonincreasing order. For each task determine the free slot nearest to, but not after, its deadline. If this free slot is before the task's release time, fail. Otherwise, assign the task to this slot.

Exercise 74 asks you to prove that the strategy just described fails to find a feasible schedule only when such a schedule does not exist.

The online equivalence class problem can be used to implement step (2). For this step, let d denote the latest deadline of any task. The usable time slots are of the form "from $i - 1$ to i " where $1 \leq i \leq d$. We will refer to these usable slots as slots 1 through d . For any slot a , define $\text{near}(a)$ as the largest i such that $i \leq a$.

and slot i is free. If no such i exists, define $\text{near}(a) = \text{near}(0) = 0$. Two slots a and b are in the same equivalence class iff $\text{near}(a) = \text{near}(b)$.

Prior to the scheduling of any task, $\text{near}(a) = a$ for all slots, and each slot is in a separate equivalence class. When slot a is assigned a task in step (2), near changes for all slots b with $\text{near}(b) = a$. For these slots the new value of near is $\text{near}(a - 1)$. Hence when slot a is assigned a task, we need to perform a unite on the equivalence classes that currently contain slots a and $a - 1$. If with each equivalence class e we retain, in $\text{nearest}[e]$, the value of near of its members, then $\text{near}(a)$ is given by $\text{nearest}[\text{find}(a)]$. (Assume that the equivalence class name is taken to be whatever the `find` operation returns.) ■

Example 6.6 [From Wires to Nets] An electronic circuit consists of components, pins, and wires. Figure 6.15 shows a circuit with the three components A, B, and C. Each wire connects a pair of pins. Two pins a and b are **electrically equivalent** iff they are either connected by a wire or there is a sequence i_1, i_2, \dots, i_k of pins such that $a, i_1; i_1, i_2; i_2, i_3; \dots; i_{k-1}, i_k$; and i_k, b are all connected by wires. A **net** is a maximal set of electrically equivalent pins. *Maximal* means that no pin outside the net is electrically equivalent to a pin in the net.

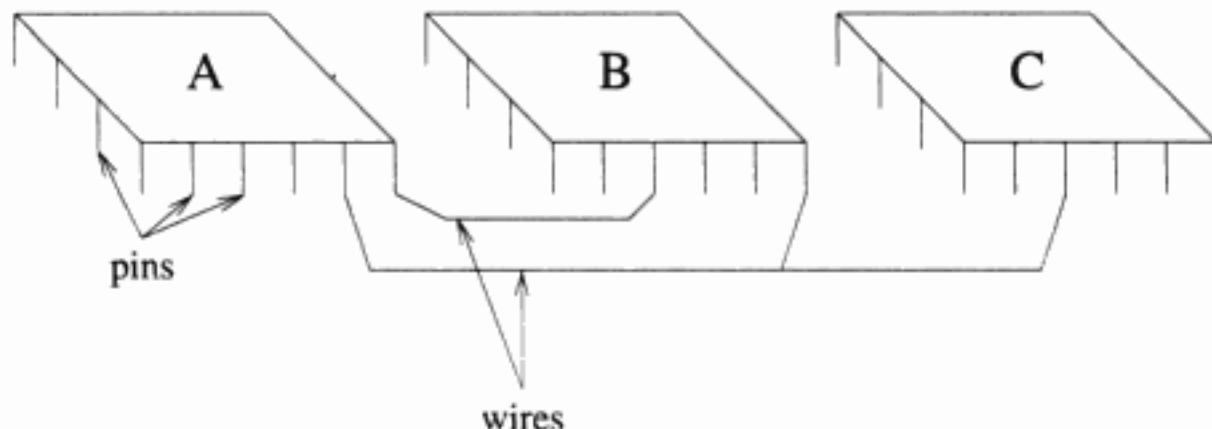


Figure 6.15 A three-chip circuit on a printed circuit board

Consider the circuit shown in Figure 6.16. In this figure only the pins and wires have been shown. The 14 pins are numbered 1 through 14. Each wire may be described by the two pins that it connects. For instance, the wire connecting pins 1 and 11 is described by the pair (1,11), which is equivalent to the pair (11,1). The set of wires is $\{(1,11), (7,11), (2,12), (12,8), (11,12), (3,13), (4,13), (13,14), (14,9), (5,14), (6,10)\}$. The nets are $\{1, 2, 7, 8, 11, 12\}$, $\{3, 4, 5, 9, 13, 14\}$ and $\{6, 10\}$.

In the **offline net finding problem**, we are given the pins and wires and are to determine the nets. This problem is modeled by the offline equivalence problem with each pin being a member of U and each wire a member of R .

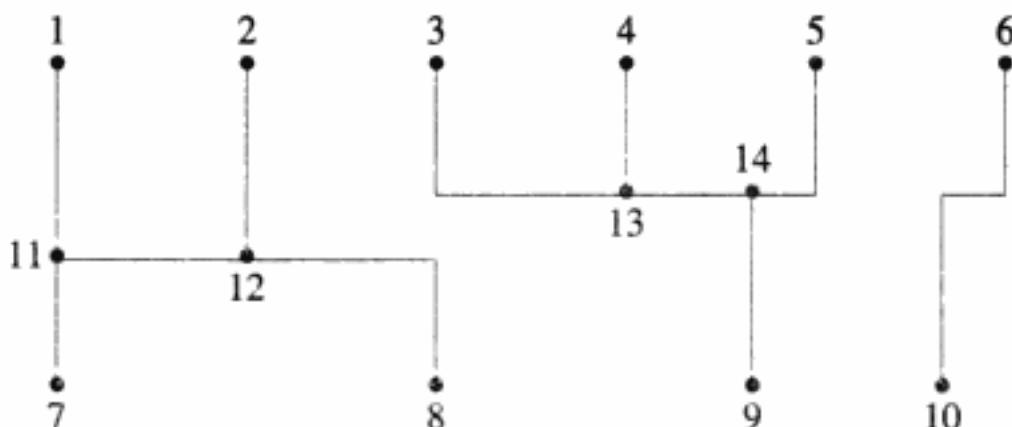


Figure 6.16 Circuit with pins and wires shown

In the **online** version we begin with a collection of pins and no wires and are to perform a sequence of operations of the form (1) add a wire to connect pins a and b and (2) find the net that contains pin a . The purpose of the find operation is to determine whether two pins are in the same net or in different nets. This version of the net problem may be modeled by the online equivalence class problem. Initially, there are no wires, and we have $R = \emptyset$. The net find operation corresponds to the equivalence class **find** operation and adding a new wire (a, b) corresponds to **combine** (a, b) , which is equivalent to **unite** $(\text{find}(a), \text{find}(b))$. ■

First Union-Find Solution

A simple solution to the online equivalence class problem is to use an array **equivClass** and let **equivClass**[i] be the class that currently contains element i . The methods to initialize, union, and find take the form given in Program 6.19. n is the number of elements. n and **equivClass** are global variables. To unite two different classes, we arbitrarily pick one of these classes and change the **equivClass** values of all elements in this class to correspond to the **equivClass** values of the elements of the other class. Note that the inputs to **unite** are **equivClass** values (i.e., the results of a **find** operation) and not element indexes. Even though **unite** works correctly when a redundant union (i.e., one in which **classA** = **classB**), we make the assumption that redundant unions are not performed. The **initialize** and **unite** methods have complexity $\Theta(n)$ (we assume that **new** does not throw an exception when invoked by **initialize**), and the complexity of **find** is $\Theta(1)$. From Examples 6.5 and 6.6, we see that in any application of these methods, we will perform one initialization, u unites, and f finds. The time needed for all of these operations is $\Theta(n+u*n+f) = \Theta(u*n+f)$.

```
int *equivClass, // equivalence class array
     n;           // number of elements

void initialize(int numberOfElements)
{// Initialize numberOfElements classes with one element each.
    n = numberOfElements;
    equivClass = new int [n + 1];
    for (int e = 1; e <= n; e++)
        equivClass[e] = e;
}

void unite(int classA, int classB)
{// Unite the classes classA and classB.
// Assume classA != classB
    for (int k = 1; k <= n; k++)
        if (equivClass[k] == classB)
            equivClass[k] = classA;
}

int find(int theElement)
{// Find the class that contains theElement.
    return equivClass[theElement];
}
```

Program 6.19 Union-find solution using arrays

Second Union-Find Solution

The time complexity of the union operation can be reduced by keeping a chain for each equivalence class because now we can find all elements in a given equivalence class by going down the chain for that class, rather than by examining all `equivClass` values. In fact, if each equivalence class knows its size, we can choose to change the `equivClass` values of the smaller equivalence class and perform the union operation even faster. By using integer pointers (also known as simulated pointers), we get quick access to the node that represents element `e`. We adopt the following conventions:

- `equivNode` is a struct with data members `equivClass`, `size`, and `next`. Program 6.20 gives the code for this struct.
- An array `node[1:n]` of type `equivNode` is used to represent the `n` elements together with the equivalence class chains.

```

struct equivNode
{
    int equivClass, // element class identifier
        size,      // number of elements in class
        next;       // pointer to next element in class
};

```

Program 6.20 The struct equivNode

- `node[e].equivClass` is both the value to be returned by `find(e)` and an integer pointer to the first node in the chain for the equivalence class `node[e].equivClass`.
- `node[e].size` is defined only if `e` is the first node on a chain. In this case `node[e].size` is the number of nodes on the chain that begins at `node[e]`.
- `node[e].next` gives the next node on the chain that contains node `e`. Since the nodes in use are numbered 1 through `n`, a NULL pointer can be simulated by the integer 0.

Program 6.21 gives the new code for `initialize`, `unite`, and `find`.

Since an equivalence class is of size $O(n)$, the complexity of the union operation is $O(n)$ when chains are used. The complexity of the initialization and find operations remain $O(n)$ and $\Theta(1)$, respectively. To determine the complexity of performing one initialization and a sequence of u unions and f finds, we will use the following lemma.

Lemma 6.1 *If we start with n classes, each have one element each and perform u nonredundant unions, then*

1. *No class has more than $u + 1$ elements.*
2. *At least $n - 2u$ singleton classes remain.*
3. $u < n$.

Proof See Exercise 72. ■

The complexity of the initialize and f finds is $O(n+f)$. For the u nonredundant unions, we note that the cost of each union is $\Theta(\text{size of smaller class})$. During the union elements are moved from the smaller class to the bigger one. The complexity of a single union is $O(\text{number of elements moved})$, and the complexity of all u unions is $O(\text{total number of element moves})$. Following a union operation, each element that is moved to a new class ends up in a class whose size is at least twice that of the class the element was in before the union operation (because elements move from an

```
equivNode *node; // array of nodes
int n;           // number of elements

void initialize(int numberOfElements)
{ // Initialize numberOfElements classes with one element each.
    n = numberOfElements;
    node = new equivNode [n + 1];

    for (int e = 1; e <= n; e++)
    {
        node[e].equivClass = e;
        node[e].next = 0; // no next node on chain
        node[e].size = 1;
    }
}

void unite(int classA, int classB)
{ // Unite the classes classA and classB.
    // Assume classA != classB
    // classA and classB are first elements in their chains

    // make classA smaller class
    if (node[classA].size > node[classB].size)
        swap(classA, classB);

    // change equivClass values of smaller class
    int k;
    for (k = classA; node[k].next != 0; k = node[k].next)
        node[k].equivClass = classB;
    node[k].equivClass = classB; // last node in chain

    // insert chain classA after first node in chain classB
    // and update new chain size
    node[classB].size += node[classA].size;
    node[k].next = node[classB].next;
    node[classB].next = classA;
}

int find(int theElement)
{ // Find the class that contains theElement.
    return node[theElement].equivClass;
}
```

Program 6.21 Union-find solution using chains and integer pointers

initially smaller class into an initially bigger class). Therefore, since at the end no class has more than $u + 1$ elements (Lemma 6.1(1)), no element can be moved more than $\log_2(u + 1)$ times during the u unions. Furthermore, from Lemma 6.1(2), at most $2u$ elements can move (because the elements left in singleton classes have never moved). So the total number of element moves cannot exceed $2u \log_2(u + 1)$. As a result, the time needed to perform the u unions is $O(u \log u)$. The complexity of the initialization and the sequence of u unions and f finds is therefore $O(n+u \log u+f)$.

EXERCISES

62. Is Program 6.17 a stable sort program?
63. Compare the run times of the bin sort methods given in Programs 6.17 and 6.18. Use $n = 10,000$; $50,000$; and $100,000$. What can you say about the overhead introduced by using the class `chain`?
64. In this exercise we shall develop a method to sort a chain using the radix sort technique.
 - (a) Write code for the method `chain<T>::radixSort(r, d)`, which sorts a chain into ascending order using the radix sort technique. The radix r and number of digits d in the radix r decomposition are inputs to your method. You may assume that a type conversion from the data type `T` to `int` is defined. The complexity of your method should be $O(d(r + n))$. Show that this is the case.
 - (b) Test the correctness of your method by compiling and executing it with your own test data.
 - (c) Compare the performance of your method with one that performs a linked insertion sort. Do so for $n = 100$; 1000 ; and $10,000$; $r = 10$; and $d = 3$.
65. (a) Write a method to sort n integers in the range 0 through $n^c - 1$ using the radix sort method and $r = n$. The complexity of your method should be $O(cn)$. Show that this is the case. Assume the integers are in a chain; the element type is `int`.
 - (b) Test the correctness of your method.
 - (c) Measure the run time of your method for $n = 10$; 100 ; 1000 ; and $10,000$ and $c = 2$. Present your results in tabular form and in graph form.
66. You are given a pile of n card decks. Each card has three fields: deck number, suit, and face value. Since each deck has at most 52 cards (some cards may be missing from a deck), the pile has at most $52n$ cards. You may assume there is at least one card from each deck. So the number of cards in the pile is at least n .

- (a) Explain how to sort this pile by deck number, within deck number by suit, and within suit by face value. You should make three bin sort passes over the pile to accomplish the sort.
- (b) Write a program to input n and a card pile and to output the sorted pile. You should represent the card pile as a chain. Each card has the fields: `deck`, `suit`, `face`, and `link`. The complexity of your program should be $O(n)$. Show that this is the case.
- (c) Test the correctness of your program.

67. [Convex Hull]

- (a) Let u , v , and w be three points in the plane. Assume that they are not collinear. Write a method to find a point inside the triangle formed by these three points.
 - (b) Let S be a set of points in the plane. Write a method to determine whether all the points are collinear. In case they are, your method should compute the endpoints of the shortest line that includes all the points. In case the points are not collinear, then you should find three noncollinear points from the given point set. You can use these three points together with your method for part (a) to determine a point inside the convex hull of S . The complexity of your method should be $O(n)$. Show that this is the case.
 - (c) Use the codes of (a) and (b) to refine Figure 6.13 into a Java program that inputs S and outputs the convex hull of S . During input the points may be collected into a doubly linked list that is later sorted by polar angle. For the sort step you may use one of the sort methods of Chapter 2, or if you have access to an $O(n \log n)$ sort, you may use it.
 - (d) Write additional convex hull programs that replace the use of a doubly linked list with (i) a chain and (ii) an array linear list.
 - (e) Test the correctness of your convex hull programs.
68. Do Exercise 67 using a singly linked list. Use the ideas of Exercise 24 to ensure that the `for` loop of step 3 of Figure 6.13 has complexity $O(n)$.
69. Develop a representation for integers that is suitable for performing arithmetic on arbitrarily large integers. The arithmetic is to be performed with no loss of accuracy. Write Java methods to input and output large integers and to perform the arithmetic operations add, subtract, multiply, and divide. The method for division will return two integers: the quotient and the remainder.
70. [Polynomials] A **univariate polynomial** of degree d has the form

$$c_d x^d + c_{d-1} x^{d-1} + c_{d-2} x^{d-2} + \cdots + c_0$$

where $c_d \neq 0$. The c_i s are the coefficients, and $d, d - 1, \dots$ are the exponents. By definition d is a nonnegative integer. For this exercise you may assume that the coefficients are also integers. Each $c_i x^i$ is a term of the polynomial. We wish to develop a Java class to support arithmetic involving polynomials. For this exercise we will represent each polynomial as a linear list $(c_0, c_1, c_2, \dots, c_d)$ of coefficients.

Develop a C++ class `polynomial` that should have an instance data member `degree`, which is the degree of the polynomial. It may have other instance data members also. Your polynomial class should support the following operations:

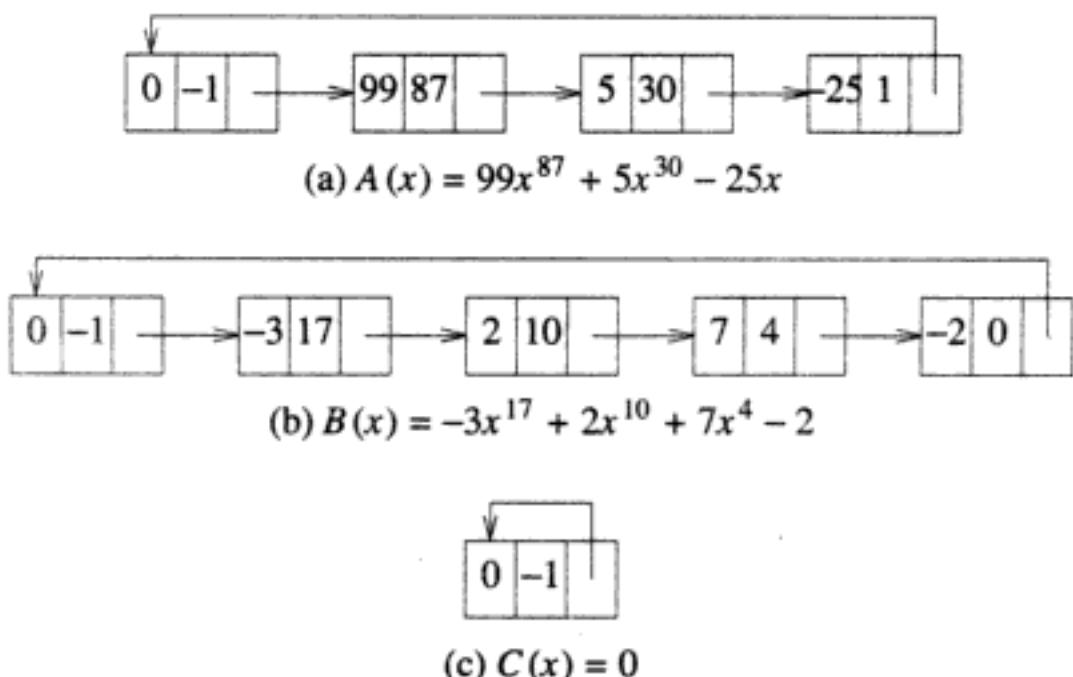
- (a) `polynomial()`—Create the zero polynomial. The degree of this polynomial is 0 and it has no terms. `polynomial()` is the class constructor.
- (b) `degree()`—Return the degree of the polynomial.
- (c) `input(inStream)`—Read in a polynomial from the input stream `inStream`. You may assume the input consists of the polynomial degree and a list of coefficients in ascending order of exponents.
- (d) `output(outStream)`—Output the polynomial to the output stream `outStream`. The output format should be the same as the input format.
- (e) `add(b)`—Add to polynomial `b` and return the result polynomial.
- (f) `subtract(b)`—Subtract the polynomial `b` and return the result.
- (g) `multiply(b)`—Multiply with polynomial `b` and return the result.
- (h) `divide(b)`—Divide by polynomial `b` and return the quotient.
- (i) `valueOf(x)`—Return the value of the polynomial at point `x`.

Test your code.

71. [Polynomials] Design and code a linked class to represent and manipulate univariate polynomials (see Exercise 70). Assume that the coefficients are integers. Use circular linked lists with header nodes. Each node should have the fields `exp` (exponent), `coeff` (coefficient), and `next` (pointer to next node). In addition to the header node, the circular list representation of a polynomial has one node for each term that has a nonzero coefficient. Terms whose coefficient is 0 are not represented. The terms are in decreasing order of exponent, and the header node has its exponent field set to -1 . Figure 6.17 gives some examples.

The external (i.e., for input or output) representation of a univariate polynomial will be assumed to be a sequence of numbers of the form $n, e_1, c_1, e_2, c_2, e_3, c_3, \dots, e_n, c_n$, where the e_i represent the exponents and the c_i the coefficients; n gives the number of terms in the polynomial. The exponents are in decreasing order; that is, $e_1 > e_2 > \dots > e_n$.

Your class should support all the methods of Exercise 70. Test the correctness of your code using suitable polynomials.

**Figure 6.17** Sample polynomials

72. Prove Lemma 6.1.
73. Write a C++ program for the online net finding problem of Example 6.6. Model the problem as the online equivalence class problem and use the chain method. Test the correctness of your program.
74. Prove that the strategy outlined in Example 6.5 fails to find a feasible schedule only when such a schedule does not exist.
75. Compare the run-time performance of Programs 6.19 and 6.21.
76. Develop a version of Program 6.21 in which the chains are replaced by array linear lists.
 - (a) Test your code.
 - (b) What is the time complexity of your new implementation?
 - (c) Compare the performance of Program 6.21 and your new implementation.
77. Develop a version of Program 6.21 in which the chains use C++ pointers rather than integer pointers. To access the node for element i in $O(1)$ time, keep an array $theNode$ such that $theNode[i]$ is a pointer to the node that represents element i .

- (a) Test your code.
 - (b) What is the time complexity of your new implementation?
 - (c) Compare the performance of Program 6.21 and your new implementation.
78. Write a C++ program for the scheduling problem of Example 6.5. Model the problem as the online equivalence class problem and use the chain method. Test the correctness of your program.

CHAPTER 7

ARRAYS AND MATRICES

BIRD'S-EYE VIEW

In practice, data are often available in tabular form. Although arrays are the most natural way to represent tabular data, we can often reduce both the space and time requirements of our programs by using a customized representation. This reduction is possible, for example, when a large portion of the table entries are 0.

This chapter begins by examining the row-major and column-major representations of a multidimensional array. These representations map a multidimensional array into a one-dimensional array.

The data object matrix is often represented as a two-dimensional array. However, matrices are normally indexed beginning at 1 rather than 0. Matrices also support operations such as add, multiply, and transpose, which are not supported by C++'s two-dimensional arrays. Therefore, we develop the class `matrix` that conforms more closely to the data object matrix.

We consider also the representation of matrices with special structures—diagonal, tridiagonal, triangular, and symmetric matrices. Using customized array representations, we can reduce the space requirements of these matrices considerably when compared to the space used by the natural two-dimensional array representation. The customized representations also result in reduced run times for most operations.

The final section of this chapter develops array and linked representations for sparse matrices (i.e., matrices with a large number of 0s) in which the positions of the 0s do not necessarily define a regular pattern.

7.1 ARRAYS

7.1.1 The Abstract Data Type

Each instance of an array is a set of pairs of the form (index, value). No two pairs in this set have the same index. The operations performed on the array follow.

- *Get an element*—Gets the value of the pair that has a given index.
- *Set an element*—Adds a pair of the form (index, value) to the set, and if a pair with the same index already exists, deletes the old pair.

These two operations define the abstract data type *array* (ADT 7.1).

```
AbstractDataType array
{
    instances
        set of (index, value) pairs, no two pairs have the same index
    operations
        get(index) : return the value of the pair with this index
        set(index, value) : add this pair to set of pairs, overwrite existing pair (if any) with
                            the same index
}
```

ADT 7.1 Abstract data type specification of an array

Example 7.1 The high temperature (in degrees Fahrenheit) for each day of last week may be represented by the following array:

```
high = {(Sunday, 82), (Monday, 79), (Tuesday, 85), (Wednesday, 92),
         (Thursday, 88), (Friday, 89), (Saturday, 91)}
```

Each pair of the array is composed of an index (day of week) and a value (the high temperature for that day). The name of the array is *high*. We can change the high temperature recorded for Monday to 83 by performing the following operation:

```
set(Monday, 83)
```

We can determine the high temperature for Friday by performing this operation:

get(Friday)

An alternative array to represent the daily high temperature is

$$\text{high} = \{(0, 82), (1, 79), (2, 85), (3, 92), (4, 88), (5, 89), (6, 91)\}$$

In this array the index is a number rather than the name of the day. The numbers $(0, 1, 2, \dots)$ replace the names of the days of the week (Sunday, Monday, Tuesday, \dots). ■

7.1.2 Indexing a C++ Array

An array is a standard data structure in C++. The index (also called **subscript**) of an array in C++ must be of the form

$$[i_1][i_2][i_3] \cdots [i_k]$$

where each i_j is a nonnegative integer. If k is one, the array is a one-dimensional array, and if k is two, it is a two-dimensional array. i_1 is the first coordinate of the index, i_2 the second, and i_k the k th. A 3-dimensional array **score**, whose values are of type integer, may be *created* in C++ using the statement

$$\text{int score}[u_1][u_2][u_3]$$

where the u_i s are positive constants or positive expressions derived from constants. With such a declaration, indexes with i_j in the range $0 \leq i_j < u_j$, $1 \leq j \leq 3$ are permitted. So the array can hold a maximum of $n = u_1 u_2 u_3$ values. Since each value in the array **score** is of type **int**, 4 bytes are needed for each. The memory, **sizeOf(score)**, needed for the entire array is therefore $4n$ bytes. The C++ compiler reserves this much memory for the array. This memory begins at byte *start* (say) and extends up to and including byte *start + sizeOf(score) - 1*.

7.1.3 Row- and Column-Major Mappings

Some applications of arrays require us to arrange the array elements into a serial or one-dimensional order. For example, the elements of an array can be output or input only one element at a time. Therefore, we must decide on the order in which the array elements are output or input. In Sections 7.3 and 7.4, we will see several types of two-dimensional tables (matrices) that we will map into a one-dimensional

array. To accomplish this mapping, we convert the two-dimensional arrangement of the table elements into a one-dimensional arrangement.

Let n be the number of elements in a k -dimensional array. The serialization of the array is done using a mapping function, which maps the array index $[i_1][i_2][i_3] \dots [i_k]$ into a number $map(i_1, i_2, \dots, i_k)$ in the range $[0, n - 1]$ such that array element with index $[i_1][i_2][i_3] \dots [i_k]$ is mapped to position $map(i_1, i_2, \dots, i_k)$ in the serial order.

When the number of dimensions is 1 (i.e., $k = 1$), the function

$$map(i_1) = i_1 \quad (7.1)$$

is used. When the number of dimensions is 2, the indexes may be arranged into a table with indexes that have the same first coordinate forming a row of the table and those with the same second coordinate forming a column (see Figure 7.1).

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]

Figure 7.1 Tabular arrangement of indexes for `int score[3][6]`

The mapping is obtained by numbering the indexes by row beginning with those in the first (i.e., top) row. Within each row, numbers are assigned from left to right. The result is shown in Figure 7.2(a). This way of mapping the positions in a two-dimensional array into a number in the range 0 through $n - 1$ is called **row major**. The numbers are assigned in row-major order. Figure 7.2(b) shows an alternative scheme, called **column major**. In column-major order the numbers are assigned by column beginning with the left column. Within a column the numbers are assigned from top to bottom.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

(a) Row-major mapping

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

(b) Column-major mapping

Figure 7.2 Mapping a two-dimensional array

When row-major order is used, the mapping function is

$$\text{map}(i_1, i_2) = i_1 u_2 + i_2 \quad (7.2)$$

where u_2 is the number of columns in the array. To verify the correctness of Equation 7.2, note that by the time the index $[i_1][i_2]$ is numbered in the row-major scheme, $i_1 u_2$ elements from the rows $0, \dots, i_1 - 1$ as well as i_2 elements from row i_1 have been numbered.

Let us try out the row-major mapping function on the sample 3×6 array of Figure 7.2(a). Since the number of columns, u_2 , is 6, the formula becomes

$$\text{map}(i_1, i_2) = 6i_1 + i_2$$

So $\text{map}(1, 3) = 6 + 3 = 9$, and $\text{map}(2, 5) = 6 * 2 + 5 = 17$. Both agree with the numbers given in Figure 7.2(a).

The row-major scheme may be extended to obtain mapping functions for arrays with more than two dimensions. Notice that in row-major order, we list first all indexes with the first coordinate equal to 0, then those with this coordinate equal to 1, and so on. Indexes with the same first coordinate are listed in increasing order of the second coordinate; that is, the indexes are listed in lexicographic order. For a three-dimensional array, we list first all indexes with the first coordinate equal to 0, then those with this coordinate equal to 1, and so on. Indexes with the same first coordinate are listed in order of the second coordinate, and indexes that agree on the first two coordinates are listed in order of the third. For example, the indexes of `score[3][2][4]` in row-major order are

<code>[0][0][0]</code>	<code>[0][0][1]</code>	<code>[0][0][2]</code>	<code>[0][0][3]</code>	<code>[0][1][0]</code>	<code>[0][1][1]</code>	<code>[0][1][2]</code>	<code>[0][1][3]</code>
<code>[1][0][0]</code>	<code>[1][0][1]</code>	<code>[1][0][2]</code>	<code>[1][0][3]</code>	<code>[1][1][0]</code>	<code>[1][1][1]</code>	<code>[1][1][2]</code>	<code>[1][1][3]</code>
<code>[2][0][0]</code>	<code>[2][0][1]</code>	<code>[2][0][2]</code>	<code>[2][0][3]</code>	<code>[2][1][0]</code>	<code>[2][1][1]</code>	<code>[2][1][2]</code>	<code>[2][1][3]</code>

The mapping function for a three-dimensional array is

$$\text{map}(i_1, i_2, i_3) = i_1 u_2 u_3 + i_2 u_3 + i_3$$

To see that this mapping function is correct, observe that the elements with the first coordinate i_1 are preceded by all elements whose first coordinate is less than i_1 . There are $u_2 u_3$ elements that have the same first coordinate. So there are $i_1 u_2 u_3$ elements with the first coordinate less than i_1 . The number of elements with the first coordinate equal to i_1 and the second coordinate less than i_2 is $i_2 u_3$, and the number with the first coordinate equal to i_1 , the second equal to i_2 , and the third less than i_3 is i_3 .

7.1.4 Array of Arrays Representation

C++ uses the so-called array-of-arrays representation to represent a multidimensional array. In this representation, a two-dimensional array is represented as a one-dimensional array in which each element is, itself, a one-dimensional array. To represent the two-dimensional array

```
int x[3][5];
```

we actually create a one-dimensional array **x** whose length is 3; each element of **x** is a one-dimensional array whose length is 5. Figure 7.3 shows the memory structure. Four separate memory blocks are used. One block (the lightly shaded block) is large enough for three pointers and each of the remaining blocks is large enough for 5 **ints**. At 4 bytes per pointer and **int**, a total of 72 bytes is used.

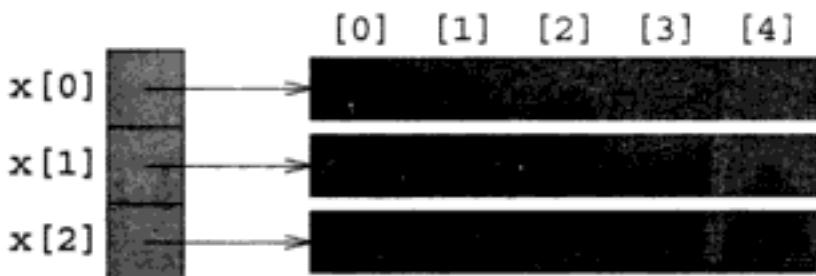


Figure 7.3 Memory structure for a two-dimensional array

C++ finds the element **x[i][j]** by using the mapping function for one-dimensional arrays (Equation 7.1) to get to the pointer in **x[i]**. This pointer gives us the address, in memory, of the zeroth element of row *i*. The mapping function for one-dimensional arrays is used once again to find the $[j]$ th element of row *i*.

A three-dimensional array is represented as a one-dimensional array, each of whose elements is a two-dimensional array. Each of these two-dimensional arrays is represented as shown in Figure 7.3.

7.1.5 Row-Major and Column-Major Representation

An alternative representation, not used by C++, is to actually create a one-dimensional array and then map our multidimensional array into this one-dimensional array using either a row- or column-major mapping. The two-dimensional array **x[3][5]** of **ints** that was considered above could be mapped into a 15-element array

```
int y[15];
```

using either a row-major or column-major mapping. In this case a single contiguous block of memory large enough to hold 15 **ints** is used. The total memory required drops from 72 bytes to 60 bytes.

To access $x[i][j]$, we must use the two-dimensional mapping function (Equation 7.2 in case a row-major mapping is used) to compute an index u and then access $y[u]$ using the one-dimensional mapping function. Depending on whether it takes more or less time to use the one-dimensional mapping function to fetch a pointer and then follow this pointer or to compute the two-dimensional mapping function, C++'s array representation scheme could be slower or faster than using a row- or column-major mapping.

7.1.6 Irregular Two-Dimensional Arrays

A two-dimensional array is regular in the sense that every row has the same number of elements. For example, every row of the 3×6 array `score` of Figure 7.1 has six elements. When two or more rows of an array have a different number of elements, we call the array **irregular**. Irregular arrays may be created and used as illustrated in Program 7.1. Notice that the only difference between regular and irregular arrays is that an irregular array may have rows whose length is different, whereas in a regular array all rows have the same length. The elements in regular and irregular arrays are accessed in the same way.

EXERCISES

1. (a) List the indexes of `score[2][3][2][2]` in row-major order.
(b) Develop the row-major mapping function for a four-dimensional array.
2. Develop the row-major mapping function for a five-dimensional array.
3. Develop the row-major mapping function for a k -dimensional array.
4. (a) List the indexes of `score[2][3][4]` in column-major order. Note that now all indexes with the third coordinate equal to 0 are listed first, then those with this coordinate equal to 1, and so on. Indexes with the same third coordinate are listed in order of the second, and those with the same last two coordinates in order of the first.
(b) Develop the column-major mapping function for a three-dimensional array.
5. (a) List the indexes of `score[2][3][2][2]` in column-major order.
(b) Develop the column-major mapping function for a four-dimensional array (see Exercise 4).
6. Develop the column-major mapping function for a k -dimensional array.
7. We wish to map the elements of a two-dimensional array beginning with the bottom row and within a row from left to right.
 - (a) List the indexes of `score[3][5]` in this order.

```
int main(void)
{
    int numberOfRows = 5;

    // define the length of each of the five rows
    int length[5] = {6, 3, 4, 2, 7};

    // declare a two-dimensional array variable
    // and allocate the desired number of rows
    int **irregularArray = new int* [numberOfRows];

    // now allocate space for the elements in each row
    for (int i = 0; i < numberOfRows; i++)
        irregularArray[i] = new int [length[i]];

    // use the array like any regular array
    irregularArray[2][3] = 5;
    irregularArray[4][6] = irregularArray[2][3] + 2;
    irregularArray[1][1] = 3;

    // output selected elements
    cout << irregularArray[2][3] << endl;
    cout << irregularArray[4][6] << endl;
    cout << irregularArray[1][1] << endl;

    return 0;
}
```

Program 7.1 Creating and using an irregular two-dimensional array

- (b) Develop the mapping function for $\text{score}[u_1][u_2]$.
8. We wish to map the elements of a two-dimensional array beginning with the right column and within a column from top to bottom.
- List the indexes of $\text{score}[3][5]$ in this order.
 - Develop the mapping function for $\text{score}[u_1][u_2]$.
9. A two-dimensional $m \times n$ array has mn elements.
- Determine the amount of memory used when these mn elements are stored using a two-dimensional C++ array and when they are stored in a one-dimensional array using row-major mapping. Assume that the

- elements are of type `int`. First do this exercise for the case $m = 10$ and $n = 2$ and then for general m and n .
- (b) How large can the ratio of the two memory requirements get?
10. A three-dimensional $m \times n \times p$ array has mnp elements.
- (a) Determine the amount of memory used when these mnp elements are stored using a three-dimensional C++ array and when they are stored in a one-dimensional array using row-major mapping. Assume that the elements are of type `int`. First do this exercise for the case $m = 10$, $n = 4$, and $p = 2$ and then for general m , n , and p .
 - (b) How large can the ratio of the two memory requirements get?
 - (c) When is one scheme expected to provide faster element access than the other?
11. A four-dimensional $m \times n \times p \times q$ array has $mnpq$ elements.
- (a) Determine the amount of memory used when these $mnpq$ elements are stored using a four-dimensional C++ array and when they are stored in a one-dimensional array using row-major mapping. Assume that the elements are of type `int`.
 - (b) How large can the ratio of the two memory requirements get?
12. A k -dimensional $u_1 \times u_2 \times \cdots \times u_k$ array has $u_1 u_2 \cdots u_k$ elements.
- (a) Determine the amount of memory used when these $u_1 u_2 \cdots u_k$ elements are stored using a k -dimensional C++ array and when they are stored in a one-dimensional array using row-major mapping. Assume that the elements are of type `int`.
 - (b) How large can the ratio of the two memory requirements get?
 - (c) When is one scheme expected to provide faster element access than the other?

7.2 MATRICES

7.2.1 Definitions and Operations

An $m \times n$ matrix is a table with m rows and n columns (Figure 7.4). m and n are the **dimensions** of the matrix.

Example 7.2 Matrices are often used to organize data. For instance, in an effort to document the assets of the world, we might first produce a list of asset types of interest. This list could include mineral deposits (silver, gold, etc.); animals (lions, elephants, etc.); people (physicians, engineers, etc.); and so on. We can determine

	col 1	col 2	col 3	col 4
row 1	7	2	0	9
row 2	0	1	0	5
row 3	6	4	2	0
row 4	8	2	7	3
row 5	1	4	9	6

Figure 7.4 A 5×4 matrix

the amount of each asset type present in the country. The data can be presented as a table with one column for each country and one row for each asset type. The result is an asset matrix with a number of columns n equal to the number of countries and a number of rows m equal to the number of asset types. We use the notation $M(i, j)$ to refer to the element in row i and column j of matrix M , $1 \leq i \leq m$, $1 \leq j \leq n$. If row i represents cats and column j represents the United States, then $asset(i, j)$ would be the number of cats in the United States.

Figure 7.5(a) shows an asset matrix for four countries; the assets listed in this matrix are platinum, gold, and silver. Country B has $asset(1, 2) = 5$ units of platinum, $asset(2, 2) = 2$ units of gold, and $asset(3, 2) = 10$ units of silver.

		country						scenario			
		A	B	C	D			asset	1	2	3
asset	platinum	2	5	1	0		platinum	20	15	50	
	gold	6	2	3	8		gold	15	12	40	
	silver	0	10	50	30		silver	1	1	2	

(a) *asset*(b) *value***Figure 7.5** Asset and value matrices

Figure 7.5(b) shows a matrix that gives the value of one unit of each asset type for three different economic scenarios. Under scenario 3 a unit of platinum is worth

$\text{value}(1, 3) = \$50$; a unit of gold is worth $\text{value}(2, 3) = \$40$; and a unit of silver is worth $\text{value}(3, 3) = \$2$. ■

The operations most commonly performed on matrices are transpose, addition or sum, and multiplication or product. The transpose of an $m \times n$ matrix M is an $n \times m$ matrix M^T with the property

$$M^T(i, j) = M(j, i), \quad 1 \leq i \leq n, \quad 1 \leq j \leq m$$

The sum of two matrices is defined only when the two matrices have the same dimensions (i.e., the same number of rows and the same number of columns). The sum of two $m \times n$ matrices A and B is a third $m \times n$ matrix C such that

$$C(i, j) = A(i, j) + B(i, j), \quad 1 \leq i \leq n, \quad 1 \leq j \leq m \quad (7.3)$$

The product $A * B$ of an $m \times n$ matrix A and a $q \times p$ matrix B is defined only when the number of columns in A equals the number of rows in B , that is, $n = q$. When $n = q$, the product is an $m \times p$ matrix C with the property

$$C(i, j) = \sum_{k=1}^n A(i, k) * B(k, j), \quad 1 \leq i \leq m, \quad 1 \leq j \leq p$$

Example 7.3 Consider the asset matrix described in Example 7.2. Suppose that the data are being accumulated by two agencies and neither duplicates the work of the other. The result is two $m \times n$ matrices: asset1 and asset2 . To get the desired asset matrix, we add the two matrices asset1 and asset2 .

Next suppose we have another matrix value (as in Figure 7.5(b)) that is an $m \times s$ matrix. $\text{value}(i, j)$ is the value of one unit of asset i under scenario j . Let $CV(i, j)$ be the value of the assets of country i under scenario j . Using the data of Figure 7.5, we see that the value of the assets held by country B under scenario 3 is

$$\begin{aligned} CV(2, 3) &= (\text{amount of platinum} * \text{value of platinum}) \\ &\quad + (\text{amount of gold} * \text{value of gold}) \\ &\quad + (\text{amount of silver} * \text{value of silver}) \\ &= \text{asset}(1, 2) * \text{value}(1, 3) + \text{asset}(2, 2) * \text{value}(2, 3) \\ &\quad + \text{asset}(3, 2) * \text{value}(3, 3) \\ &= 5 * 50 + 2 * 40 + 10 * 2 \\ &= 350 \end{aligned}$$

We see that CV is an $n \times s$ matrix and that

$$CV(i, j) = \sum_{k=1}^m asset(k, i) * value(k, j) = \sum_{k=1}^m asset^T(i, k) * value(k, j)$$

So CV satisfies the equation

$$CV = asset^T * value$$

Figure 7.6(a) gives the transpose of the asset matrix of Figure 7.5(a), and Figure 7.6(b) gives the CV matrix that corresponds to the asset and value matrices of Figure 7.5. ■

	P	G	S
A	2	6	0
B	5	2	10
C	1	3	50
D	0	8	30

(a) $asset^T$

	1	2	3
A	130	102	340
B	140	109	350
C	115	101	270
D	150	126	380

(b) $CV = asset^T * value$

Figure 7.6 Example for matrix transpose and product

C++ functions to compute the transpose of a matrix and to add and multiply two matrices represented as two-dimensional arrays were considered in Chapter 2 (Programs 2.21, 2.19, 2.22, and 2.23, respectively).

7.2.2 The Class matrix

A $\text{rows} \times \text{cols}$ matrix M , all of whose elements are integer, may be represented as a two-dimensional integer array

```
int x[rows][cols];
```

with $M(i, j)$ being stored as $x[i-1][j-1]$. This representation requires the user to write applications using array indexes that differ from matrix indexes by 1. Alternatively, we may define the array x as

```
int x[rows + 1][cols + 1];
```

and not use the array positions $[0][*]$ and $[*][0]$. In this section we develop a representation in which the elements of matrix M are mapped into a one-dimensional array in row-major order.

The class **matrix** uses a one-dimensional array **element** to store, in row-major order, the **rows * cols** elements of a **rows × cols** matrix. Program 7.2 gives the class header. Notice that we intend to overload the **()** operator so that matrices may be indexed in a program the same way they are indexed in mathematics. Additionally we intend to overload the arithmetic operators so that they work with objects of type **matrix**.

```
template<class T>
class matrix
{
    friend ostream& operator<<(ostream&, const matrix<T>&);

public:
    matrix(int theRows = 0, int theColumns = 0);
    matrix(const matrix<T>&);

    ~matrix() {delete [] element;}
    int rows() const {return theRows;}
    int columns() const {return theColumns;}
    T& operator()(int i, int j) const;
    matrix<T>& operator=(const matrix<T>&);

    matrix<T> operator+() const; // unary +
    matrix<T> operator+(const matrix<T>&) const;
    matrix<T> operator-() const; // unary minus
    matrix<T> operator-(const matrix<T>&) const;
    matrix<T> operator*(const matrix<T>&) const;
    matrix<T>& operator+=(const T&);

private:
    int theRows,      // number of rows in matrix
        theColumns; // number of columns in matrix
    T *element;      // element array
};
```

Program 7.2 Header for the class **matrix**

Program 7.3 gives the constructor and copy constructor for the class. Notice

that the constructor allows you to create a 0×0 matrix as well as matrices for which both `theRows > 0` and `theColumns > 0`.

```

template<class T>
matrix<T>::matrix(int theRows, int theColumns)
{// matrix constructor.
    // validate theRows and theColumns
    if (theRows < 0 || theColumns < 0)
        throw illegalParameterValue("Rows and columns must be >= 0");
    if ((theRows == 0 || theColumns == 0)
        && (theRows != 0 || theColumns != 0))
        throw illegalParameterValue
        ("Either both or neither rows and columns should be zero");

    // create the matrix
    this->theRows = theRows;
    this->theColumns = theColumns;
    element = new T [theRows * theColumns];
}

template<class T>
matrix<T>::matrix(const matrix<T>& m)
{// Copy constructor for matrices.
    // create matrix
    theRows = m.theRows;
    theColumns = m.theColumns;
    element = new T [theRows * theColumns];

    // copy each element of m
    copy(m.element,
        m.element + theRows * theColumns,
        element);
}

```

Program 7.3 Constructor and copy constructor for matrix

Program 7.4 gives the code to overload the assignment operator `=`.

To index a matrix using left and right parenthesis `()`, we overload the C++ function operator `()`, which can take any number of parameters. In our case, we use two parameters of type `int` with the overloaded `()`, because to index a matrix we need two integer parameters. Program 7.5 gives the code to overload `()`. This code returns a reference to the (i, j) th element of a matrix and this reference may

```
template<class T>
matrix<T>& matrix<T>::operator=(const matrix<T>& m)
{// Assignment. (*this) = m.
    if (this != &m)
        {// not copying to self
            delete [] element;
            theRows = m.theRows;
            theColumns = m.theColumns;
            element = new T [theRows * theColumns];
            // copy each element
            copy(m.element,
                  m.element + theRows * theColumns,
                  element);
        }
    return *this;
}
```

Program 7.4 Overloading the = operator for matrix

be used to either set or get the value of the (i, j) th element using statements such as $a(i, j) = 2$ and $x = a(i, j)$, where a is of type `matrix`.

```
template<class T>
T& matrix<T>::operator()(int i, int j) const
{// Return a reference to element (i,j).
    if (i < 1 || i > theRows
        || j < 1 || j > theColumns)
        throw matrixIndexOutOfBoundsException();
    return element[(i - 1) * theColumns + j - 1];
}
```

Program 7.5 Overloading the () operator for matrix

Program 7.6 gives the code for matrix addition. Since matrices have been mapped into one-dimensional arrays, we can add two matrices using a single `for` loop rather than two nested `for` loops as were used in Program 2.21. The codes for matrix operations such as increment (increase the value of each matrix entry by the same amount) and subtraction are similar to that for matrix addition.

The loop structure of the matrix multiplication code (Program 7.7) is similar to that of Program 2.23. There are three nested `for` loops. The innermost loop uses

```

template<class T>
matrix<T> matrix<T>::operator+(const matrix<T>& m) const
{// Return w = (*this) + m.
    if (theRows != m.theRows
        || theColumns != m.theColumns)
        throw matrixSizeMismatch();

    // create result matrix w
    matrix<T> w(theRows, theColumns);
    for (int i = 0; i < theRows * theColumns; i++)
        w.element[i] = element[i] + m.element[i];

    return w;
}

```

Program 7.6 Matrix addition

Equation 7.3 to compute the (i, j) th element of the product matrix. When we enter the innermost loop, `element[ct]` is the first element of row i and `m.element[cm]` is the first of column j . To go to the next element of row i , `ct` is to be incremented by 1 because in row-major order the elements of a row occupy consecutive positions. To go to the next element of column j , `cm` is to be incremented by `m.theColumns`, as consecutive elements of a column are `m.theColumns` positions apart in row-major order. When the innermost loop completes, `ct` is positioned at the end of row i and `cm` is at the end of column j . For the next iteration of the `for j` loop, `ct` needs to be at the start of row i and `cm` at the start of the next column of `m`. The resetting that occurs after the innermost loop completes positions `ct`. When the `for j` loop completes, `ct` should be set to the position of the first element of the next row and `cm` to that of the first element of the first column.

The matrix multiplication code can be made more efficient by reducing the number of cache misses as in the $\bar{i}kj$ order version of Program 4.4. The code for the remaining methods of `matrix` may be found at the Web site for this book.

Complexity

The complexity of the matrix constructor and destructor is $O(1)$ when T is a primitive data type of C++ (e.g., `int`, `double`). When T is a user-defined data type, the complexity of the constructor (destructor) is $O(\text{theRows} * \text{theColumns})$ because the constructor (destructor) for the data type T is invoked for every position in the array `element` when this array is created (deleted).

The asymptotic complexity of the copy constructor and the add method is $O(\text{theRows} * \text{theColumns})$ if we assume that the times to copy a matrix term

```
template<class T>
matrix<T> matrix<T>::operator*(const matrix<T>& m) const
{// matrix multiply.  Return w = (*this) * m.
    if (theColumns != m.theRows)
        throw matrixSizeMismatch();

    matrix<T> w(theRows, m.theColumns); // result matrix

    // define cursors for *this, m, and w
    // and initialize to location of (1,1) element
    int ct = 0, cm = 0, cw = 0;

    // compute w(i,j) for all i and j
    for (int i = 1; i <= theRows; i++)
    {// compute row i of result
        for (int j = 1; j <= m.theColumns; j++)
        { // compute first term of w(i,j)
            T sum = element[ct] * m.element[cm];

            // add in remaining terms
            for (int k = 2; k <= theColumns; k++)
            {
                ct++; // next term in row i of *this
                cm += m.theColumns; // next in column j of m
                sum += element[ct] * m.element[cm];
            }
            w.element[cw++] = sum; // save w(i,j)

            // reset to start of row and next column
            ct -= theColumns - 1;
            cm = j;
        }

        // reset to start of next row and first column
        ct += theColumns;
        cm = 0;
    }
}

return w;
}
```

Program 7.7 Matrix multiplication

and add two matrix terms are both $\Theta(1)$. The matrix multiplication code has the complexity $O(\text{theRows} * \text{theColumns} * \text{m.theColumns})$.

EXERCISES

13. (a) What is the transpose of the matrix of Figure 7.4?
 (b) What is the product of the matrix of Figure 7.4 and the transpose?
14. Do Exercise 13 using the matrix of Figure 7.2(b).
15. Add code for the methods `-=` (decrease each matrix entry by a specified amount), `<<` (input a matrix), `*=` (multiply each matrix entry by a specified value), and `/=` to the class `matrix`. Test your methods.
16. To the class `matrix` add the method `transpose()`, which returns the transpose of `*this`. Test your code.
17. (a) Develop the class `matrixAs2DArray` in which a matrix is represented as a two-dimensional array. Your class should include all methods of `matrix` as well as a method to transpose a matrix.
 (b) Test your methods.
 (c) Compare the performance of the matrix addition and multiplication methods of the classes `matrix` and `matrixAs2DArray`. Do this comparison by making actual run-time measurements. What can you say about the merits of using the row-major mapping instead of a two-dimensional array?

7.3 SPECIAL MATRICES

7.3.1 Definitions and Applications

A **square** matrix has the same number of rows and columns. Some special forms of square matrices that arise frequently are

- **Diagonal.** M is diagonal iff $M(i, j) = 0$ for $i \neq j$; see Figures 7.7(a) and 7.8(a).
- **Tridiagonal.** M is tridiagonal iff $M(i, j) = 0$ for $|i - j| > 1$; see Figures 7.7(b) and 7.8(b).
- **Lower triangular.** M is lower triangular iff $M(i, j) = 0$ for $i < j$; see Figures 7.7(c) and 7.8(c).
- **Upper triangular.** M is upper triangular iff $M(i, j) = 0$ for $i > j$; see Figures 7.7(d) and 7.8(d).

$$\begin{bmatrix} X & & & \\ & X & & \\ & & X & \\ & & & X \\ & & & & X \\ & & & & & X \\ & & & & & & X \end{bmatrix}$$

(a) Diagonal

$$\begin{bmatrix} X & X & & & & & & \\ X & X & X & & & & & \\ & X & X & X & & & & \\ & & X & X & X & & & \\ & & & X & X & X & & \\ & & & & X & X & X & \\ & & & & & X & X & X \\ & & & & & & X & X \end{bmatrix}$$

(b) Tridiagonal

$$\begin{bmatrix} X & & & & & & & \\ X & X & & & & & & \\ X & X & X & & & & & \\ X & X & X & X & & & & \\ X & X & X & X & X & & & \\ X & X & X & X & X & X & & \\ X & X & X & X & X & X & X & \\ X & X & X & X & X & X & X & X \end{bmatrix}$$

(c) Lower triangular

$$\begin{bmatrix} X & X & X & X & X & X & X & X \\ & X & X & X & X & X & X & X \\ & & X & X & X & X & X & X \\ & & & X & X & X & X & X \\ & & & & X & X & X & X \\ & & & & & X & X & X \\ & & & & & & X & X \\ & & & & & & & X \end{bmatrix}$$

(d) Upper triangular

X denotes an element that may be nonzero
Elements not shown are zero

Figure 7.7 Location of nonzero elements in special matrices

- **Symmetric.** Matrix M is symmetric iff $M(i, j) = M(j, i)$ for all i and j ; see Figure 7.8(e).

Example 7.4 Consider the six cities Gainesville, Jacksonville, Miami, Orlando, Tallahassee, and Tampa, which are all in Florida. We may number these cities 1 through 6 in the listed order. The distance between pairs of these cities may be represented using a 6×6 matrix $distance$. The i th row and column of this matrix represent the i th city, and $distance(i, j)$ is the distance between city i and city j . Figure 7.9 shows the distance matrix. Since $distance(i, j) = distance(j, i)$ for all i and j , the distance matrix is symmetric. ■

Example 7.5 Suppose we have a stack of n cartons with carton 1 at the bottom and carton n at the top. Each carton has width w and depth d . The height of the i th carton is h_i . The volume occupied by the stack is $w * d * \sum_{i=1}^n h_i$. In the **stack folding** problem, we are permitted to create substacks of cartons by selecting a fold point i and creating two adjacent stacks. One has cartons 1 through i and the other

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 6 \end{matrix}$$

(a) Diagonal

$$\begin{matrix} 2 & 1 & 0 & 0 \\ 3 & 1 & 3 & 0 \\ 0 & 5 & 2 & 7 \\ 0 & 0 & 9 & 0 \end{matrix}$$

(b) Tridiagonal

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 4 & 2 & 7 & 0 \end{matrix}$$

(c) Lower triangular

$$\begin{matrix} 2 & 1 & 3 & 0 \\ 0 & 1 & 3 & 8 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 \end{matrix}$$

(d) Upper triangular

$$\begin{matrix} 2 & 4 & 6 & 0 \\ 4 & 1 & 9 & 5 \\ 6 & 9 & 4 & 7 \\ 0 & 5 & 7 & 0 \end{matrix}$$

(e) Symmetric

Figure 7.8 4×4 special matrices

	GN	JX	MI	OD	TL	TM
GN	0	73	333	114	148	129
JX	73	0	348	140	163	194
MI	333	348	0	229	468	250
OD	114	140	229	0	251	84
TL	148	163	468	251	0	273
TM	129	194	250	84	273	0

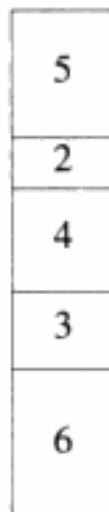
GN = Gainesville JX = Jacksonville MI = Miami	OD = Orlando TL = Tallahassee TM = Tampa
---	--

Distance in miles

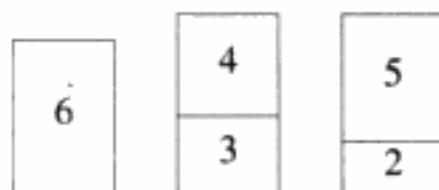
Figure 7.9 A distance matrix (source: Rand McNally Road Atlas)

cartons $i + 1$ through n . By repeating this folding process, we may obtain several stacks of cartons. If we create s stacks, the width of the arrangement is $s * w$, its depth is d , and the height h is the height of the tallest stack. The volume of the

space needed for the stacks is $s * w * d * h$. Since h is the height of a stack of boxes i through j for some i and j , $i \leq j$, the possible values for h are given by the $n \times n$ matrix H where $H(i, j)$ is 0 for $i > j$ and is $\sum_{k=i}^j h_k$ for $i \leq j$. Since the height of each carton is > 0 , an $H(i, j)$ value of 0 indicates an infeasible stack height. Figure 7.10(a) shows a five-carton stack. The numbers inside each rectangle give the carton height. Figure 7.10(b) shows a folding of the five-carton stack into three stacks. The height of the largest stack is 7. The matrix H is an upper-triangular matrix, as shown in Figure 7.10(c). One application of the stack folding problem is to the folding of a stack of electronic components so as to minimize the area occupied by the folded stack (see Web site). ■



(a) Stack



(b) Three-stack folding

	1	2	3	4	5
1	6	9	13	15	20
2	0	3	7	9	14
3	0	0	4	6	11
4	0	0	0	2	7
5	0	0	0	0	5

(c) H matrix

Figure 7.10 Stack folding

7.3.2 Diagonal Matrices

One way to represent a `rows` \times `rows` diagonal matrix D is to use a two-dimensional array `element[rows][rows]` and use `element[i-1][j-1]` to represent $D(i, j)$. This representation requires space for rows^2 objects of type `T`. However, since a diagonal matrix contains at most `rows` nonzero entries, we may use a one-dimensional array `element[rows]` and use `element[i-1]` to represent $D(i, i)$. The elements of the matrix D that are not represented in the array are all known to be zero. This representation, which requires space for only `rows` objects of type `T`, leads to the C++ class `diagonalMatrix` (Programs 7.8 through 7.10).

The complexity of the constructor is $O(1)$ when `T` is a primitive data type and $O(\text{rows})$ when `T` is a user-defined data type. The complexity of the methods `get` and `set` is $\Theta(1)$.

```

template<class T>
class diagonalMatrix
{
public:
    diagonalMatrix(int theN = 10);
    ~diagonalMatrix() {delete [] element;}
    T get(int, int) const;
    void set(int, int, const T&);
private:
    int n;          // matrix dimension
    T *element;   // 1D array for diagonal elements
};

template<class T>
diagonalMatrix<T>::diagonalMatrix(int theN)
{// Constructor.
// validate theN
if (theN < 1)
    throw illegalParameterValue("Matrix size must be > 0");

n = theN;
element = new T [n];
}

```

Program 7.8 Header and constructor for diagonalMatrix

```

template <class T>
T diagonalMatrix<T>::get(int i, int j) const
{// Return (i,j)th element of matrix.
// validate i and j
if (i < 1 || j < 1 || i > n || j > n)
    throw matrixIndexOutOfBoundsException();

if (i == j)
    return element[i-1]; // diagonal element
else
    return 0;           // nondiagonal element
}

```

Program 7.9 Get method for diagonalMatrix

```
template<class T>
void diagonalMatrix<T>::set(int i, int j, const T& newValue)
{// Store newValue as (i,j)th element.
    // validate i and j
    if (i < 1 || j < 1 || i > n || j > n)
        throw matrixIndexOutOfBoundsException();
    if (i == j)
        // save the diagonal value
        element[i-1] = newValue;
    else
        // nondiagonal value, newValue must be zero
        if (newValue != 0)
            throw illegalParameterValue
                ("nondiagonal elements must be zero");
}
```

Program 7.10 Set method for `diagonalMatrix`

7.3.3 Tridiagonal Matrix

In a `rows` × `rows` tridiagonal matrix, the nonzero elements lie on one of the three diagonals:

1. Main diagonal—for this, $i = j$.
2. Diagonal below main diagonal—for this, $i = j + 1$.
3. Diagonal above main diagonal—for this, $i = j - 1$.

The number of elements on these three diagonals is $3 * \text{rows} - 2$. We can use a one-dimensional array `element` with $3 * \text{rows} - 2$ positions to represent the tridiagonal matrix. Only the elements on the three diagonals are explicitly stored. Consider the 4×4 tridiagonal matrix of Figure 7.8(b). There are 10 elements on the main diagonal and the diagonals just above and below the main diagonal. If these elements are mapped into `element` by rows, then `element[0:9] = [2, 1, 3, 1, 3, 5, 2, 7, 9, 0]`; if the mapping is by columns, `element = [2, 3, 1, 1, 5, 3, 2, 9, 7, 0]`; and if the mapping is by diagonals beginning with the lowest, then `element = [3, 5, 9, 2, 1, 2, 0, 1, 3, 7]`. As we can see, there are several reasonable choices for the mapping of T into `element`. Each requires a different code for the `get` and `set` methods. Suppose that the class `tridiagonalMatrix` maps by diagonals. The data members and constructor are quite similar to those of the class `diagonal`. Program 7.11 gives the code for `get`; the code for `set` is similar and is on the Web site.

```

template <class T>
T tridiagonalMatrix<T>::get(int i, int j) const
{// Return (i,j)th element of matrix.

    // validate i and j
    if ( i < 1 || j < 1 || i > n || j > n)
        throw matrixIndexOutOfBoundsException();

    // determine element to return
    switch (i - j)
    {
        case 1: // lower diagonal
            return element[i - 2];
        case 0: // main diagonal
            return element[n + i - 2];
        case -1: // upper diagonal
            return element[2 * n + i - 2];
        default: return 0;
    }
}

```

Program 7.11 The method `get` for a tridiagonal matrix

An alternative space-efficient representation of a tridiagonal array is considered in Exercise 25. This alternative representation uses an irregular array (see Section 7.1.6).

7.3.4 Triangular Matrices

In an n -row lower-triangular matrix (Figure 7.7(c)), the nonzero region has one element in row 1, two in row 2, \dots , and n in row n ; and in an n -row upper-triangular matrix, the nonzero region has n elements in row 1, $n - 1$ in row 2, \dots , and one in row n . In both cases the total number of elements in the nonzero region is

$$\sum_{i=1}^n i = n(n+1)/2$$

Both kinds of triangular matrices may be represented by using a one-dimensional array of size $n(n + 1)/2$. Consider a lower-triangular matrix L mapped into a one-dimensional array `element`. Two possible ways to do the mapping are by rows and

by columns. If the mapping is done by rows, then the 4×4 lower-triangular matrix of Figure 7.8(c) has the mapping `element[0:9] = [2, 5, 1, 0, 3, 1, 4, 2, 7, 0]`. The column mapping results in `element = [2, 5, 0, 4, 1, 3, 2, 1, 7, 0]`.

Consider element $L(i, j)$ of a lower-triangular matrix. If $i < j$, the element is in the zero region. If $i \geq j$, the element is in the nonzero region. In a row mapping, the element $L(i, j)$, $i \geq j$, is preceded by $\sum_{k=1}^{i-1} k$ nonzero region elements that are in rows 1 through $i - 1$ and $j - 1$ such elements from row i . The total number of nonzero region elements that precede $L(i, j)$ in a row mapping is $i(i - 1)/2 + j - 1$. This expression also gives the position of $L(i, j)$ in `element`. Using this expression, we arrive at the `set` method given in Program 7.12; the method to get a value is similar. Both methods have time complexity $\Theta(1)$.

```
template<class T>
void lowerTriangularMatrix<T>::set(int i, int j, const T& newValue)
{// Store newValue as (i,j)th element.
    // validate i and j
    if ( i < 1 || j < 1 || i > n || j > n)
        throw matrixIndexOutOfBoundsException();

    // (i,j) in lower triangle iff i >= j
    if (i >= j)
        element[i * (i - 1) / 2 + j - 1] = newValue;
    else
        if (newValue != 0)
            throw illegalParameterValue
                ("elements not in lower triangle must be zero");
}
```

Program 7.12 The method `lowerTriangularMatrix<T>::set`

An alternative space-efficient representation of a triangular array is considered in Exercise 26. This alternative representation uses an irregular array (see Section 7.1.6).

7.3.5 Symmetric Matrices

An $n \times n$ symmetric matrix can be represented using a one-dimensional array of size $n(n + 1)/2$ by storing either the lower or upper triangle of the matrix using one of the schemes for a triangular matrix. The elements that are not explicitly stored may be computed from those that are explicitly stored.

EXERCISES

18. Tubing down Sleepy River is a pleasant activity that thousands of folks participate in during the summer. Sleepy River has seven places where you can get into or out of the river. These places are numbered 1 through 7; 2 is downstream from 1, 3 is downstream from 2, and so on. A different tube rental vendor does business at each location. The vendor that rents tubes at 1 will retrieve you and your tube at places 1, 3, 6, and 7; the vendor that rents tubes at 2 has a pickup service at 3, 5, and 6. The pickup services for the vendors that rent tubes at 3, 4, 5, 6, and 7 are, respectively, at 3, 5, 7; 5, 6, 7; 7; 6, 7; and 7.
- Write a 7×7 matrix in which the (i, j) entry is 1 if it is possible to rent a tube at i and be picked up at j and is 0 otherwise. Is your matrix symmetric, upper triangular, or lower triangular?
 - Write a 7×7 matrix in which the (i, j) entry is 1 if it is possible to be picked up at i when you rent your tube at j and is 0 otherwise. Is your matrix symmetric, upper triangular, or lower triangular?
 - Renumber the seven tube rental places in the order downstream to upstream. For this numbering scheme write a 7×7 matrix in which the (i, j) entry is 1 if it is possible to rent a tube at i and be picked up at j and is 0 otherwise. Is your matrix symmetric, upper triangular, or lower triangular?
19. There are five equally spaced kennels in a row. Each kennel has a dog that is chained to the kennel post, and the length of each chain equals the distance between two adjacent kennels. Assume that dog i is chained to kennel i .
- Which kennels can dog 3 visit?
 - Write a 5×5 matrix in which the (i, j) entry is 1 if dog i can visit kennel j and is 0 otherwise.
 - Is your matrix symmetric, upper triangular, lower triangular, tridiagonal, or diagonal?
20. (a) To the class `diagonalMatrix` (Program 7.8) add methods to input, output, add, subtract, multiply, and transpose diagonal matrices represented as one-dimensional arrays. Note that in each case the result is a diagonal matrix represented as a one-dimensional array.
- Test the correctness of your codes.
 - What is the time complexity of each of your methods?
21. (a) To the class `tridiagonalMatrix` (Program 7.11) add methods to input, output, add, subtract, and transpose tridiagonal matrices.
- Test the correctness of your codes.

- (c) What is the time complexity of each method?
22. (a) Develop a C++ class `tridiagonalByColumns` that maps a tridiagonal $n \times n$ matrix into a one-dimensional array of size $3n - 2$ by columns. Include methods for the input, output, get, set, add, subtract, and transpose operations.
- (b) Test the correctness of your codes.
- (c) What is the time complexity of each method?
23. Do Exercise 22 for the class `tridiagonalByRows` in which the $n \times n$ tridiagonal matrix is mapped into a one-dimensional array of size $3n - 2$ by rows.
24. Is the product of two tridiagonal matrices necessarily tridiagonal?
25. Develop the class `tridiagonalAsIrregularArray` in which a tridiagonal matrix is represented using a two-dimensional array `element`. When representing an $n \times n$ matrix, rows 0 and $n - 1$ of `element` have two positions each; the remaining rows have three positions each. See Section 7.1.6 to determine how to create such an array. Your class must include all methods included in the class `tridiagonalMatrix`.
- (a) Test your code.
 - (b) Comment on the relative merits of the one-dimensional array representation as used in the class `tridiagonalMatrix` and the irregular array representation as used in `tridiagonalAsIrregularArray`.
26. Develop the class `lowerTriangleAsIrregularArray` in which a lower-triangular matrix is represented using a two-dimensional array `element`. When representing an $n \times n$ matrix, row i of `element` has i positions. See Section 7.1.6 to determine how to create such an array. Your class must include all methods included in the class `lowerTriangularMatrix`.
- (a) Test your code.
 - (b) Comment on the relative merits of the one-dimensional array representation as used in the class `lowerTriangularMatrix` and the irregular array representation as used in `lowerTriangleAsIrregularArray`.
27. Develop the C++ class `upperTriangularMatrix` analogous to Program 7.12 for the case of an upper-triangular matrix. Include constructor, get, and set methods.
28. To the class `lowerTriangularMatrix` add methods to input, output, add, and subtract lower-triangular matrices. What is the time complexity of each method?

29. To the class `lowerTriangularMatrix` add the method `transpose`, which returns the transpose of the lower-triangular matrix `*this`. The transpose is an instance of the class `upperTriangularMatrix`. What is the time complexity of your code?
30. Let A and B be two $n \times n$ lower-triangular matrices. The total number of elements in the lower triangles of the two matrices is $n(n+1)$. Devise a scheme to represent both triangles in an array `element[n+1][n]`. [Hint: If you join the lower triangle of A and the upper triangle of B^T , you get an $(n+1) \times n$ matrix.] Write the get and set functions for both A and B . The complexity of each should be $\Theta(1)$.
31. Write a method to multiply two lower-triangular matrices that are members of the class `lowerTriangularMatrix` (Program 7.12). The result matrix is to be stored in a two-dimensional array. What is the time complexity of your method?
32. Write a method to multiply a lower-triangular and an upper-triangular matrix mapped into one-dimensional arrays by rows. The result matrix is in a two-dimensional array. What is the time complexity of your method?
33. Suppose that symmetric matrices are stored by mapping the lower-triangular region into one-dimensional arrays by rows. Develop a C++ class `lowerSymmetricMatrix` that includes methods for the get and set operations. The complexity of your methods should be $\Theta(1)$.
34. In an $n \times n$ **C-matrix**, all terms other than those in row 1, row n , and column 1 (see Figure 7.11)) are zero. A C-matrix has at most $3n - 2$ nonzero terms. A C-matrix may be compactly stored in a one-dimensional array by first storing row 1, then row n , and then the remaining column 1 elements.

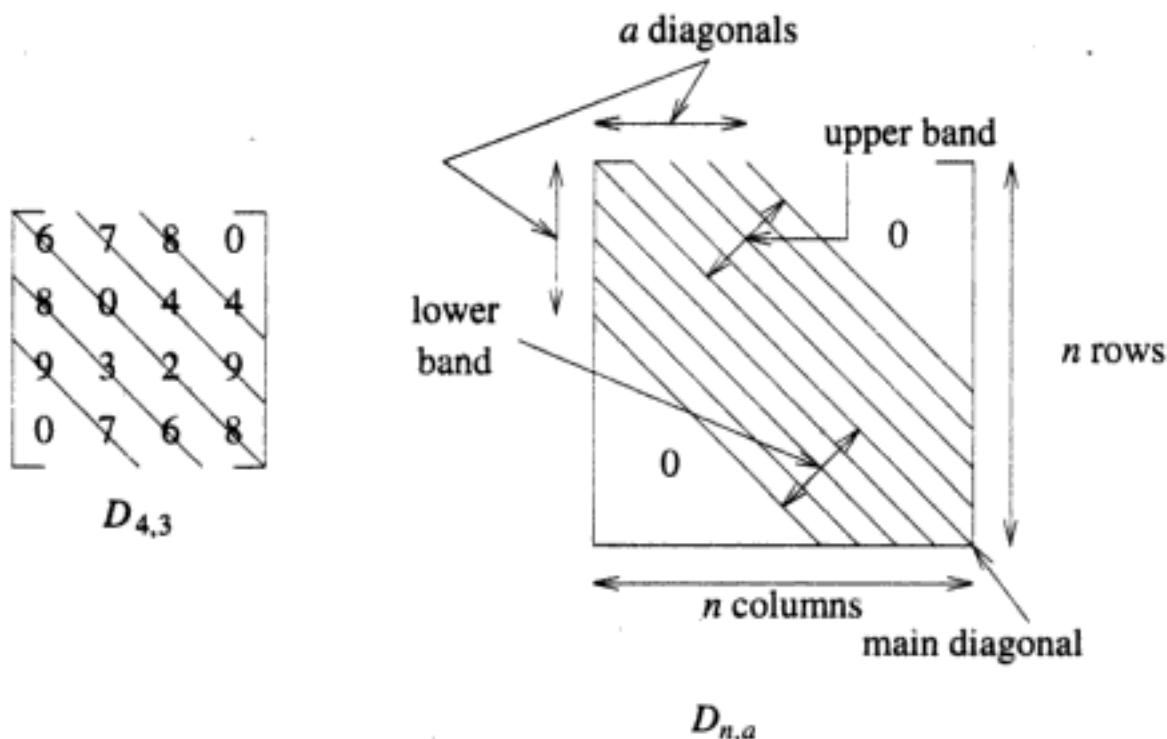
X	X	X	X	X	X	X
X						
X						
X						
X						
X	X	X	X	X	X	X

x denotes a possible nonzero
All other terms are zero

Figure 7.11 A C-matrix

- (a) Give a sample 4×4 C-matrix and its compact representation.

- (b) Show that an $n \times n$ C-matrix has at most $3n - 2$ nonzero terms.
- (c) Develop a class `cMatrix` that represents an $n \times n$ C-matrix in a one-dimensional array `element` as above. You should include the constructor and `get` and `set` methods.
35. An $n \times n$ square matrix M is an **antidiagonal** matrix iff all entries $M(i, j)$ with $i + j \neq n + 1$ equal zero.
- Give a sample of a 4×4 antidiagonal matrix.
 - Show that the antidiagonal matrix M has at most n nonzero entries.
 - Devise a way to represent an antidiagonal matrix in a one-dimensional array of size n .
 - Use the representation of (c) to arrive at the code for the C++ class `antidiagonalMatrix` that includes methods for the get and set operations.
 - What is the time complexity of your get and set codes?
 - Test your code.
36. An $n \times n$ matrix T is a **Toeplitz matrix** iff $T(i, j) = T(i - 1, j - 1)$ for all i and j , $i > 1$ and $j > 1$.
- Show that a Toeplitz matrix has at most $2n - 1$ distinct elements.
 - Develop a mapping of a Toeplitz matrix into a one-dimensional array of size $2n - 1$.
 - Use the mapping of (b) to obtain a C++ class `toeplitzMatrix` in which a Toeplitz matrix is mapped into a one-dimensional array of size $2n - 1$. Include methods for the get and store operations. The complexity of each should be $\Theta(1)$.
 - Write a method to multiply two Toeplitz matrices stored as in (b). The result is stored in a two-dimensional array. What is the time complexity of your code?
37. A **square band matrix** $D_{n,a}$ is an $n \times n$ matrix in which all the nonzero terms lie in a band centered around the main diagonal. The band includes the main diagonal and $a - 1$ diagonals below and above the main diagonal (Figure 7.12).
- How many elements are in the band matrix $D_{n,a}$?
 - What is the relationship between i and j for elements $d_{i,j}$ in the band of $D_{n,a}$?

**Figure 7.12** Square band matrix

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]
9	7	8	3	6	6	0	2	8	7	4	9	8	4

d ₂₀	d ₃₁	d ₁₀	d ₂₁	d ₃₂	d ₀₀	d ₁₁	d ₂₂	d ₃₃	d ₀₁	d ₁₂	d ₂₃	d ₀₂	d ₁₃
-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

Figure 7.13 Representation for matrix $D_{4,3}$ of Figure 7.12

- (c) Assume that the band of $D_{n,a}$ is mapped into a one-dimensional array \mathbf{b} by diagonals, starting with the lowest diagonal. Figure 7.13 shows the representation for band matrix $D_{4,3}$ of Figure 7.12.
Develop a formula for the location of an element $d_{i,j}$ in the lower band of $D_{n,a}$ (location(d_{10}) = 2 in the example above).
- (d) Develop the C++ class `squareBandMatrix` that uses the mapping of (c); include methods for the get and set operations. What is the time complexity of each method? Test your code.
- (e) Develop the class `squareBandAsIrregularArray` that uses a two-dimensional array `element` in which each row has as many positions as the width of the band at that row. For example, `element[0]` is a one-dimensional array with a positions; include methods for the get and set

operations. What is the time complexity of each method? Test your code.

- (f) What are the relative merits/demerits of the two representations used in (d) and (e)?

7.4 SPARSE MATRICES

7.4.1 Motivation

An $m \times n$ matrix is said to be **sparse** if “many” of its elements are zero. A matrix that is not sparse is **dense**. The boundary between a dense and a sparse matrix is not precisely defined. Diagonal and tridiagonal $n \times n$ matrices are sparse. Each has $O(n)$ nonzero terms and $O(n^2)$ zero terms. Is an $n \times n$ triangular matrix sparse? It has at least $n(n - 1)/2$ zero terms and at most $n(n + 1)/2$ nonzero terms. For the representation schemes in this section to be competitive over the standard two-dimensional array representation, the number of nonzero terms will need to be less than $n^2/3$ and in some cases less than $n^2/5$. In this context we will classify triangular matrices as dense.

Sparse matrices such as diagonal and tridiagonal matrices have sufficient structure in their nonzero regions that we can devise a simple representation scheme whose space requirements equal the size of the nonzero region. In this section we are concerned with sparse matrices with an irregular or unstructured nonzero region.

Example 7.6 A supermarket is conducting a study of the mix of items purchased by its customers. For this study, data are gathered for the purchases made by 1000 customers. These data are organized into a matrix, *purchases*, with *purchases*(*i, j*) being the quantity of item *i* purchased by customer *j*. Suppose that the supermarket has an inventory of 10,000 different items. The *purchases* matrix is therefore a $10,000 \times 1000$ matrix. If the average customer buys 20 different items, only about 20,000 of the 10,000,000 matrix entries are nonzero. However, the distribution of the nonzero entries does not fall into any well-defined structure.

The supermarket has a $10,000 \times 1$ matrix, *price*. *price*(*i*) is the selling price of one unit of item *i*. The matrix *spent* = *purchases*^T * *price* is a 1000×1 matrix that gives the amount spent by each customer. If a two-dimensional array is used to represent the matrix *purchases*, an unnecessarily large amount of memory is used and the time required to compute *spent* is also unnecessarily large. ■

7.4.2 Representation Using a Single Linear List

The nonzero entries of an irregular sparse matrix may be mapped into a linear list in row-major order. For example, the nonzero entries of the 4×8 matrix of Figure 7.14(a) in row-major order are 2, 1, 6, 7, 3, 9, 8, 4, 5.

To reconstruct the matrix structure, we need to record the originating row and column for each nonzero entry. So each element of the array into which the sparse

0	0	0	2	0	0	1	0
0	6	0	0	7	0	0	3
0	0	0	9	0	8	0	0
0	4	5	0	0	0	0	0

(a) A 4×8 matrix

terms	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7.	3	9	8	4	5

(b) Its linear list representation

Figure 7.14 A sparse matrix and its linear list representation

matrix is mapped needs to have three fields: `row` (the row of the matrix entry), `col` (the column of the matrix entry), and `value` (the value of the matrix entry). For this purpose we define the struct `matrixTerm` that has these three data members. The data type of `row` and `col` is `int` and that of `value` is `T`.

The nonzero entries of the matrix of Figure 7.14(a) may be stored in a linear list `terms` in row-major order as shown in Figure 7.14(b). The row labeled `terms` gives the list index of a matrix term. In addition to storing the nonzero entries of the matrix, we need to store the number of rows and columns in the matrix.

Suppose that our linear list `terms` is an instance of `arrayList`. If we assume that the nine nonzero elements of Figure 7.14(a) are stored as `ints`, the linear list representation requires 8 (for number of rows and columns) + $9 * 12$ (each nonzero element requires the storage of its row, column, and value; 4 bytes each) + 8 (for the size and capacity of the linear list `terms`) + 4 (for a reference to the array `terms.elements`) = 128 bytes. If we had represented our matrix using a 4×8 array `theArray`, the space used would have been $32 * 4$ (for the array entries) + $4 * 4$ (for the pointers in `thearray[]`) + 4 (for a reference to `theArray`) = 148 bytes. The space saving achieved by the linear list representation isn't much in this example. However, for the matrix `purchase` (see our supermarket example, Example 7.6), the array representation takes approximately $20,000 * 12 = 240,000$ bytes, whereas the two-dimensional array representation needs approximately $10,000,000 * 4 = 40,000,000$ bytes. The space saving is about 39,760,000 bytes! A corresponding amount of time is saved creating the linear list representation over initializing a two-dimensional array.

The linear list representation of a sparse matrix does not lead to efficient implementations of the get and set operations. The get operation takes $O(\log [\text{number of nonzero entries}])$ time when an array linear list and binary search are used. The set operation takes $O(\text{number of nonzero entries})$ time because we may need to move this many entries to make room for the new term. Both operations take $O(\text{number of nonzero entries})$ time when a linked linear list is used. Each of these operations takes $\Theta(1)$ time using the standard two-dimensional array representation. However, matrix operations such as transpose, add, and multiply can be performed efficiently

using the linear list representation.

The Class `sparseMatrix`

Based on our experiments of Section 6.1.6, we are motivated to use an array representation for `terms`. We use the class `arrayList` with the following methods added to it.

1. `reSet(newSize)` ... change the size of the list to `newSize` increasing the capacity of the array if necessary.
2. `set(theIndex, theElement)` ... make `theElement` the list element whose index is `theIndex`.
3. `clear()` ... make the list size zero

Program 7.13 gives the header for the class `sparseMatrix` which uses the row-major mapping of a sparse matrix into an `arrayList`. Notice that the only constructor this class has is the default constructor.

```
template<class T>
class sparseMatrix {
public:
    void transpose(sparseMatrix<T> &b);
    void add(sparseMatrix<T> &b, sparseMatrix<T> &c);
private:
    int rows,      // number of rows in matrix
        cols;      // number of columns in matrix
    arrayList<matrixTerm<T> > terms;
                    // list of nonzero terms
};
```

Program 7.13 Header for `sparseMatrix`

Program 7.14 gives the code to overload the output operator `<<`. Notice that this code employs an iterator to sequence through the elements in the `arrayList` in left-to-right order. This order gets the nonzero matrix elements in row-major order. If the output is printed or displayed on a screen we will see one matrix term per line.

Program 7.15 inputs the sparse matrix entries in row-major order and sets up the internal representation. Exercise 42 considers refinements of this code.

```

template <class T>
ostream& operator<<(ostream& out, sparseMatrix<T>& x)
{// Put x in output stream.

    // put matrix characteristics
    out << "rows = " << x.rows << " columns = "
        << x.cols << endl;
    out << "nonzero terms = " << x.terms.size() << endl;

    // put terms, one per line
    for (arrayList<matrixTerm<T>>::iterator i = x.terms.begin();
        i != x.terms.end(); i++)
        out << "a(" << (*i).row << ',', << (*i).col
            << ") = " << (*i).value << endl;

    return out;
}

```

Program 7.14 Overloading the output operator <<

Matrix Transpose .

Program 7.16 gives the code for the transpose method. We first set the number of rows and columns in the result matrix **b** and also make the size of the linear list **b.terms** equal to the number of terms in the transpose. Even though the list **b.terms** has none of the terms of the transpose yet, we set its size equal to the number of nonzero entries it will eventually have. This step is necessary so that we can use the method **arrayList<T>::set** to place entries into arbitrary positions in **b.terms**. If we do not change the size of **b.terms** in this manner, then we must grow the linear list one element at a time. As we will see, when we transpose a sparse matrix, the zeroth element of the matrix being transposed may be the sixth (say) element of the transpose. We cannot insert an element at position 6 of a linear list unless the list size is currently 6 or more. By beginning with a list whose size equals the final desired size (even though no element is defined or correct), we can essentially use the list as a one-dimensional array. The element in any position of the list can be assigned a new value using the method **set**.

Next we create two arrays **colSize** and **rowNext**. **colSize[i]** is the number of nonzero entries of the input matrix ***this** that are in column **i**, and **rowNext[i]** denotes the index in **b** for the next nonzero term that is in row **i** of the transpose. For the sparse matrix of Figure 7.14(a), **colSize[1:8] = [0, 2, 1, 2, 1, 1, 1, 1]**. Prior to the generation of any entries in the transpose matrix, **rowNext[1:8] = [0, 0, 2, 3, 5, 6, 7, 8]**.

```
template<class T>
istream& operator>>(istream& in, sparseMatrix<T>& x)
{// Input a sparse matrix.

    // input matrix characteristics
    int numberOfRows;
    cout << "Enter number of rows, columns, and #terms"
        << endl;
    in >> x.rows >> x.cols >> numberOfRows;

    // should validate input values here, left as an exercise

    // set size of x.terms and ensure enough capacity
    x.terms.reSet(numberOfTerms);

    // input terms
    matrixTerm<T> mTerm;
    for (int i = 0; i < numberOfRows; i++)
    {
        cout << "Enter row, column, and value of term "
            << (i + 1) << endl;
        in >> mTerm.row >> mTerm.col >> mTerm.value;
        // should validate input, left as an exercise

        x.terms.set(i, mTerm);
    }

    return in;
}
```

Program 7.15 Overloading the input operator >>

`colSize` is computed in the first two `for` loops by simply examining each term of the input matrix using an iterator. `rowNext` is computed in the next `for` loop. In this `for` loop, `rowNext[i]` is set to be the number of entries in rows 0 through `i-1` of the transpose matrix `b`, which is equal to the number of entries in columns 0 through `i-1` of the input matrix `*this`. Finally, in the last `for` loop, the nonzero entries are copied from the input matrix to their correct positions in `b`.

Although Program 7.16 is more complex than its counterpart for matrices stored as two-dimensional arrays (see Program 2.19), for matrices with many zero entries, Program 7.16 is faster. It is not too difficult to see that computing the transpose of

```

template<class T>
void sparseMatrix<T>::transpose(sparseMatrix<T> &b)
{// Return transpose of *this in b.

    // set transpose characteristics
    b.cols = rows;
    b.rows = cols;
    b.terms.reSet(terms.size());

    // initialize to compute transpose
    int* colSize = new int[cols + 1];
    int* rowNext = new int[cols + 1];

    // find number of entries in each column of *this
    for (int i = 1; i <= cols; i++) // initialize
        colSize[i] = 0;
    for (arrayList<matrixTerm<T> >::iterator i = terms.begin();
         i != terms.end(); i++)
        colSize[(*i).col]++;
}

// find the starting point of each row of b
rowNext[1] = 0;
for (int i = 2; i <= cols; i++)
    rowNext[i] = rowNext[i - 1] + colSize[i - 1];

// perform the transpose copying from *this to b
matrixTerm<T> mTerm;
for (arrayList<matrixTerm<T> >::iterator i = terms.begin();
     i != terms.end(); i++)
{
    int j = rowNext[(*i).col]++; // position in b
    mTerm.row = (*i).col;
    mTerm.col = (*i).row;
    mTerm.value = (*i).value;
    b.terms.set(j, mTerm);
}
}

```

Program 7.16 Transpose a sparse matrix

the *purchases* matrix of Example 7.6 using the linear list representation and method `transpose` is much faster than using a two-dimensional array representation and the transpose function of Program 2.19. The time complexity of `transpose` is $O(\text{cols} + \text{terms.size}())$.

Adding Two Matrices

The code of Program 7.17 computes `c = *this + b`. The result matrix `c` is produced by scanning the non-zero terms of `*this` and `b` from left to right. This scan is done using two iterators—`it` (for the matrix `*this`) and `ib` (for the matrix `b`). On each iteration of the `while` loop, we need to determine whether the position in `b` of the term `*it` is before, at the same place as, or after that of `*ib`. We can make this determination by comparing the row-major index of these two terms. However, it is actually simpler to compute and compare the row-major index plus the number of columns in the matrix, as we do (`tIndex` and `bIndex`).

The `while` loop of `add` is iterated at most `terms.size() + b.terms.size()` times, as on each iteration the iterator `it` for `*this` or the iterator `ib` for `b` or both advance by one. The first `for` loop is iterated at most `terms.size()` times, while the second is iterated $O(b.terms.size())$ times. Also, each iteration of each loop takes constant time. So the complexity of `add` is $O(\text{terms.size()} + \text{b.terms.size()})$. If the two matrices `*this` and `b` were represented as two-dimensional arrays, it would take $O(\text{rows} * \text{cols})$ time to add them. When `terms.size() + b.terms.size()` is much less than `rows * cols`, the sparse matrix representation results in a faster implementation.

7.4.3 Representation Using Many Linear Lists

An alternative sparse matrix representation results when we store the nonzero entries in each row in a separate linear list. In exploring this alternative, we use linked lists; array lists may be used instead (Exercise 52).

The Representation

We link together the nonzero entries in each row to form a chain (called a row chain) as shown by the unshaded nodes of Figure 7.15.

Each unshaded node represents a nonzero term of the sparse matrix. Each node in a row chain has the fields (data members) `element` and `next`. The `element` field of a node in a row chain has two subfields—`col` (the column number for the term) and `value` (the value of the term). Figure 7.16(a) shows the structure of a node on a row chain. Subfields of `element` are not shaded.

Row chains are created only for rows that have at least one nonzero term. The nodes on a row chain are linked in ascending order of their `col` value. The row chains (i.e., unshaded chains) are collected together by using another chain (called the header-node chain) as shown by the shaded nodes of Figure 7.15. Like a node

```

template<class T>
void sparseMatrix<T>::add(sparseMatrix<T> &b, sparseMatrix<T> &c)
{// Compute c = (*this) + b.

    // verify compatibility
    if (rows != b.rows || cols != b.cols)
        throw matrixSizeMismatch(); // incompatible matrices

    // set characteristics of result c
    c.rows = rows;
    c.cols = cols;
    c.terms.clear();
    int cSize = 0;

    // define iterators for *this and b
    arrayList<matrixTerm<T> >::iterator it = terms.begin();
    arrayList<matrixTerm<T> >::iterator ib = b.terms.begin();
    arrayList<matrixTerm<T> >::iterator itEnd = terms.end();
    arrayList<matrixTerm<T> >::iterator ibEnd = b.terms.end();

    // move through *this and b adding like terms
    while (it != itEnd && ib != ibEnd)
    {
        // row-major index plus cols of each term
        int tIndex = (*it).row * cols + (*it).col;
        int bIndex = (*ib).row * cols + (*ib).col;

        if (tIndex < bIndex)
            {// b term comes later
            c.terms.insert(cSize++, *it);
            it++;
        }
    }
}

```

Program 7.17 Adding two sparse matrices (continues)

on a row chain, a node on the header-node chain has two fields—**element** and **next**. The **element** field of a node on the header-node chain has two subfields—**row** (row number for corresponding row chain) and **rowChain** (the chain of unshaded nodes; **rowChain.firstNode** points to the first unshaded node). Figure 7.16(b) shows the structure of a node on the header-node chain.

The nodes on the header-node chain are linked together in ascending order of

```
else {if (tIndex == bIndex)
    {// both in same position

        // append to c only if sum not zero
        if ((*it).value + (*ib).value != 0)
        {
            matrixTerm<T> mTerm;
            mTerm.row = (*it).row;
            mTerm.col = (*it).col;
            mTerm.value = (*it).value + (*ib).value;
            c.terms.insert(cSize++, mTerm);
        }

        it++;
        ib++;
    }
    else
    {// a term comes later
        c.terms.insert(cSize++, *ib);
        ib++;
    }
}
}

// copy over remaining terms
for (; it != itEnd; it++)
    c.terms.insert(cSize++, *it);
for (; ib != ibEnd; ib++)
    c.terms.insert(cSize++, *ib);
}
```

Program 7.17 Adding two sparse matrices (concluded)

their `row` value. Each node on the header-node chain may be viewed as the header node of a row chain. An empty header-node chain represents a matrix with no nonzero terms.

Element Types

The struct `rowElement` defines a data type suitable for the elements of a row chain. Its data members are `col` (column index of the term) and `value` (value of the term). The struct `headerElement` defines a corresponding struct for elements in

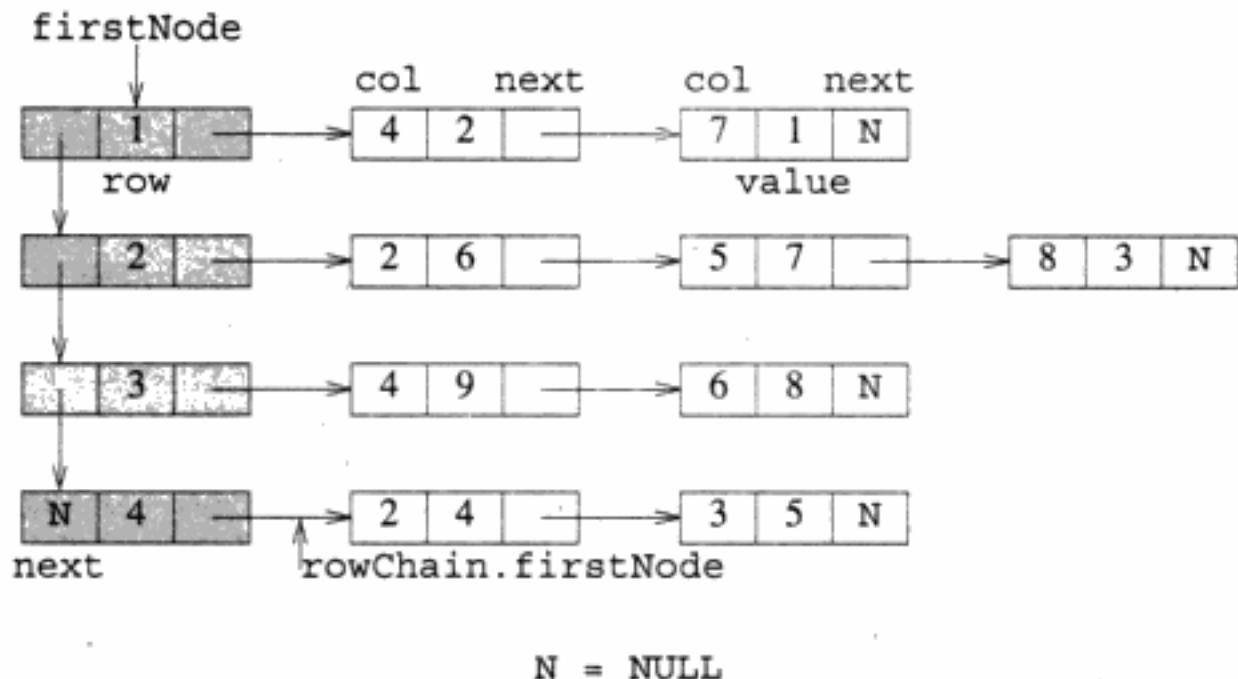
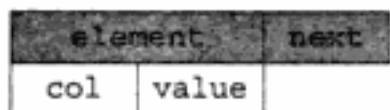
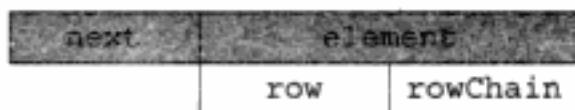


Figure 7.15 Linked representation of matrix of Figure 7.14(a)



(a) Node for nonzero term



(b) Node for header-node chain

Figure 7.16 Node structures in linked sparse matrix representation

the header-node chain. Its data members are `row` (index of row) and `rowChain` (the actual chain, data type is `extendedChain`).

The Class `linkedMatrix`

The class that uses the representation of Figure 7.15 is called `linkedMatrix`. The row chains and header-node chain of Figure 7.15 are actually represented as instances of `extendedChain` because we will need to append (i.e., add at the right end) elements to these chains. In an `extendedChain` (Program 6.12) an element can be appended in $\Theta(1)$ time, whereas it takes $\Theta(\text{size of chain})$ time to append to an instance of `chain` (Program 6.2). For our sparse matrix application we have added the method `zero()` to `extendedChain`. This method makes the chain size 0 but does not delete the chain nodes (unlike `clear` which makes the chain size 0

and also deletes all chain nodes).

The data members for `linkedMatrix` are almost the same as those for `sparseMatrix`; the exception is that the data member `terms` is replaced by `headerChain`, which is of type `extendedChain`. The codes to overload `<<` and `>>` are on the Web site.

The Method `linkedMatrix<T>::transpose`

For the transpose operation, we use bins to collect the terms of the input matrix `*this` that belong in the same row of the result. `bin[i]` is a chain for the terms of row `i` of the result matrix `b`. In the nested `while` loops of Program 7.18, we examine the terms of `*this` in row-major order by going down the header-node chain of the input matrix and making a left-to-right traversal of each row chain. We move along the header-node and row chains by using an iterator `ih` for the header-node chain and another iterator `ir` for the row chain. Each term encountered in this ordered traversal of the matrix `*this` is appended to the bin chain for its row in the result. The bin chains are collected together in the `for` loop to create the header-node chain of the result.

The time spent in the `while` loops is $O(\text{number of nonzero terms})$, and the time spent in the `for` loop is $O(\text{this->cols})$. Therefore, the overall time is $O(\text{number of nonzero terms} + \text{this->cols})$.

Exercise 51 asks you to implement the `add` method as well as other basic methods.

7.4.4 Performance Measurement

The space requirements of `sparseMatrix` and `linkedMatrix` are approximately the same. However, we can modify the former representation to use 33 percent less space (see Exercise 47); this modification does not reduce run-time efficiency. Although Exercise 53 considers an alternative linked representation for sparse matrices, this alternative representation takes 66 percent more space than does `linkedMatrix`.

Figures 7.17 and 7.18 give the measured run times for matrix addition and transpose using two-dimensional arrays as in Programs 2.21 and 2.19 (`2DArray`, `2DA`), `sparseMatrix` (SM) and `linkedMatrix` (LM). The `add` used two 500×500 sparse matrices; one had 1994 nonzero terms, and the other had 999 nonzero terms. For `transpose` a 500×500 matrix with 1994 nonzero terms was used.

The linked sparse matrix implementation, while slower than the array implementation of a sparse matrix, is faster than `2DArray`. The reduction in time obtained by the array sparse matrix implementation `sparseMatrix` (relative to the nonsparse implementation `2DArray`) is quite striking—matrix addition and transpose times are reduced by a factor of about 20.

```
template<class T>
void linkedMatrix<T>::transpose(linkedMatrix<T> &b)
{// Return transpose of *this as matrix b.
    b.headerChain.clear(); // delete all nodes from b

    // create bins to collect rows of b
    extendedChain<rowElement<T> > *bin;
    bin = new extendedChain<rowElement<T> > [cols + 1];

    // head node iterator
    extendedChain<headerElement<T> >::iterator
        ih = headerChain.begin(),
        ihEnd = headerChain.end();

    // copy terms of *this into bins
    while (ih != ihEnd)
    {// examine all rows
        int r = ih->row; // row number for row chain

        // row chain iterator
        extendedChain<rowElement<T> >::iterator
            ir = ih->rowChain.begin(),
            irEnd = ih->rowChain.end();

        rowElement<T> x;
        // terms from row r of *this go to column r of b
        x.col = r;

        while (ir != irEnd)
        {// copy a term from the row chain into a bin
            x.value = ir->value;
            // x will eventually be in row ir->col of transpose
            bin[ir->col].push_back(x);
            ir++; // next term in row
        }

        ih++; // go to next row
    }
}
```

Program 7.18 Transpose a sparse matrix (continues)

```
// set dimensions of transpose
b.rows = cols;
b.cols = rows;

// assemble header chain of transpose
headerElement<T> h;
// scan bins
for (int i = 1; i <= cols; i++)
    if (!bin[i].empty())
        {// row i of transpose
            h.row = i;
            h.rowChain = bin[i];
            b.headerChain.push_back(h);
            bin[i].zero(); // save from destructor
        }

h.rowChain.zero(); // save from destructor

delete [] bin;
}
```

Program 7.15 Transpose a sparse matrix (concluded)

Class	add	transpose
2DArray	2.69	1.97
sparseMatrix	0.13	0.09
linkedMatrix	***	1.57

*** Time not measured
Times are in milliseconds

Figure 7.17 Time taken by different matrix implementations

EXERCISES

38. (a) Draw figures similar to Figure 7.14(b) for the matrices of Figure 7.8.
- (b) Manually work out a sparse matrix transpose as implemented by `sparseMatrix<T>::transpose` using the matrix of Figure 7.8(b).
- (c) Manually work out a sparse matrix add as implemented by `sparseMatrix<T>::add` using the matrices of Figures 7.8(b) and (c).

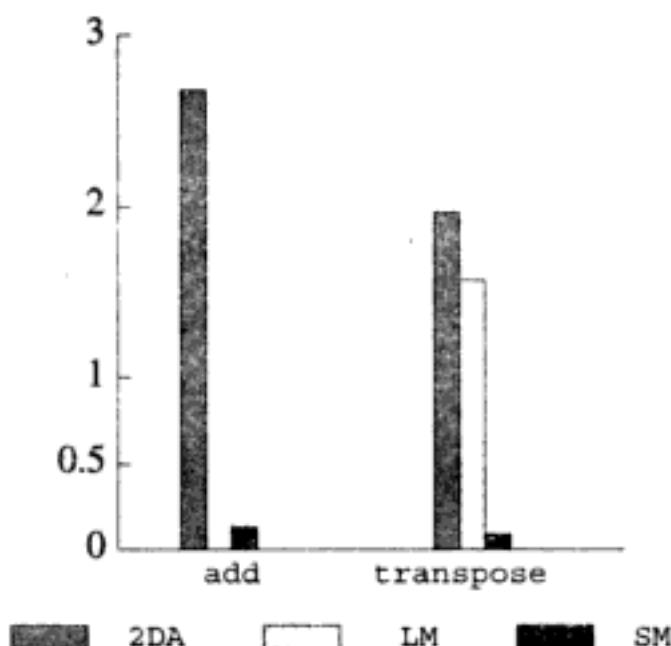


Figure 7.18 Sparse matrix run times in milliseconds

39. (a) Suppose that a 500×500 matrix that has 2000 nonzero terms is to be represented. How much space is needed when a 500×500 two-dimensional array of type `int` is used? How much space is needed when `sparseMatrix` is used?
 (b) How many nonzero elements must an $m \times n$ matrix have before the space required by `sparseMatrix` exceeds that required by an $m \times n$ two-dimensional array? You may assume that `T` is `int`.
40. Develop the formula for the row-major index of element (i, j) of a `rows` \times `cols` matrix. Why is it easier to compute the value `index = row-major index + cols` than it is to compute the row-major index? Show that if `index1` and `index2` are the indexes of two matrix elements, then `index1 < index2` iff the first element precedes the second element in row-major order.
41. Write the methods `get(theRow, theColumn)` and `set(theRow, theColumn, theValue)` for the class `sparseMatrix`. What is the time complexity of your methods?
42. Refine the input code of Program 7.15 so that it verifies that the terms are, in fact, input in row-major order; that the row and column indexes of each term are valid; and that each term is nonzero.
43. Write a copy constructor for the class `sparseMatrix`.

44. Suppose that a sparse matrix is mapped into an `arrayList` (with the additional methods indicated in this chapter) in column-major order of the nonzero terms.
- (a) Develop the representation of the sparse matrix of Figure 7.14(a).
 - (b) Write the get and set methods for sparse matrices stored in this way.
 - (c) What is the time complexity of your methods?
45. Write a method to multiply two sparse matrices represented using an array linear list. Assume that both matrices are mapped in row-major order. The result matrix is similarly represented.
46. Do Exercise 45 using a column-major mapping.
47. We can reduce the space required by the linear list representation of a sparse matrix by eliminating the data member `row` from `matrixTerm` and using an array `rowStart` such that `rowStart[i]` is the index of the first term in row `i`. The terms of row `i` have the indexes `rowStart[i] ... rowStart[i+1]`.
- (a) Draw a figure similar to Figure 7.14(b) for the sparse matrix of Figure 7.14(a). Use the representation of this exercise and show the terms in row-major order. Also give the values of `rowStart[1:5]`.
 - (b) Write the struct `newMatrixTerm` to represent a nonzero term. This struct differs from the struct `matrixTerm` only in that `newMatrixTerm` does not have the data member `row`.
 - (c) Write the class `newSparseMatrix` to implement a sparse matrix using the representation of this exercise. Your class must include all methods implemented for `sparseMatrix` as well as the methods `get` and `set` (see Exercise 41).
 - (d) Test your code.
 - (e) Make a qualitative comparison of the classes `newSparseMatrix` and `sparseMatrix`.
 - (f) Compare the run times of the `add` and `transpose` methods of the two classes using 500×500 sparse matrices with approximately 6000 nonzero terms.
48. Write a matrix multiply method for the representation of Exercise 47. Test your code.
49. (a) Draw figures similar to Figure 7.15 for the matrices of Figure 7.8.
(b) Manually work out a sparse matrix transpose as implemented by `linkedMatrix<T>::transpose` using the matrix of Figure 7.8(b).

50. (a) Suppose that a 500×500 matrix that has 2000 nonzero terms is to be represented. How much space is needed when a 500×500 two-dimensional array of type `int` is used? How much space is needed when `linkedMatrix` is used?
- (b) How many nonzero elements must an $m \times n$ matrix have before the space required by `linkedMatrix` exceeds that required by an $m \times n$ two-dimensional array?
51. Extend the class `linkedMatrix` by adding methods for the following operations:
- Set a term given the row index, column index, and value of the term.
 - Get a term given its row and column indexes.
 - Add two sparse matrices.
 - Subtract two sparse matrices.
 - Multiply two sparse matrices.

Also, refine the code for `>>` as described in Exercise 42. Test your code.

52. Develop the class `arrayMatrix` in which each row of nonzero terms is represented as a separate array linear list. The representation differs from that of Section 7.4.3 in that the linked lists of the figure are replaced by array lists. Implement methods to input, output, add, transpose, and multiply. Test your code.
53. [Orthogonal Linked List Representation] An alternative linked representation for sparse matrices uses nodes that have the fields `down`, `right`, `row`, `col`, and `value`. Each nonzero entry of the sparse matrix is represented by a node. The zero terms are not explicitly stored. The nodes are linked together to form two circular lists. The first list, the row list, is made up by linking nodes by rows and within rows by columns using the `right` field. The second list, the column list, is made up by linking nodes via the `down` field. In this list, nodes are linked by columns and within columns by rows. These two lists share a common header node. In addition, a node is added to contain the dimensions of the matrix.
- Write down any 5×8 matrix that has exactly nine nonzero terms such that at least one nonzero term appears in each row and each column. For this sparse matrix draw the linked representation.
 - Suppose that an $m \times n$ matrix with t nonzero terms is represented as above. How small must t be so that the above linked scheme uses less space than an $m \times n$ array uses?

- (c) Design a suitable external (i.e., one that can be used for input and output) representation for a sparse matrix. Your representation should not require explicit input of the zero terms.
- (d) Develop a class that uses the representation of this exercise. Include all the methods of `sparseMatrix`.
- (e) For each public method of the class, obtain its asymptotic time complexity. How do these complexities compare with the corresponding complexities for the methods of `sparseMatrix`?

CHAPTER 8

STACKS

BIRD'S-EYE VIEW

Stacks and queues are, perhaps, the most frequently used data structures. Both are restricted versions of the linear or ordered list data structure studied extensively in Chapters 5 and 6. In fact, stacks and queues are so widely used that the C++ STL provides the classes `stack` and `queue`, which are array implementations of these data structures. We will study stacks in this chapter and queues in the next. Even though C++ provides an implementation of a stack and a queue, we obtain our own stack and queue implementations just to learn how to implement these data structures.

The stack data structure is obtained from a linear list by restricting the insertions and removals to take place from the same end. As a result, a stack is a last-in-first-out (LIFO) structure. Since a stack is a special kind of linear list, it is natural to derive stack classes from corresponding linear list classes. Therefore, we may derive an array-stack class from any of the array linear list representations of Chapter 5; a linked-stack class may be derived from the class `chain` (Program 6.2). Although these derivations simplify the programming task, they result in code that incurs a significant run-time penalty. Since a stack is a very basic data structure that many applications employ, we also develop array- and linked-stack classes from scratch (i.e., not derived from any other class). These latter classes provide improved run-time performance over their derived counterparts.

Six application codes that make use of stacks are also developed. The first is a simple program to match left and right parentheses in an expression. The second is the classical Towers of Hanoi problem in which we move disks one at a time from a source tower to a destination tower using one intermediary tower; each tower operates as a stack. The third application uses stacks to represent shunting tracks in a railroad yard. The objective is to rearrange the cars in a train into the desired order. The fourth application is from the computer-aided design of circuits field. In this application we use a stack to determine whether a switch box can be feasibly routed. The fifth application revisits the offline equivalence class problem introduced in Section 6.5.4. A stack enables us to determine the equivalence classes in linear time. The final application considered in this chapter is the classical rat-in-a-maze problem in which we are to find a path from the entrance of a maze to its exit. You are urged to go through this application very carefully, as its treatment in this chapter illustrates many software-engineering principles. Additional stack applications appear in later chapters.

8.1 DEFINITION AND APPLICATIONS

Definition 8.1 A stack is a linear list in which insertions (also called additions and pushes) and removals (also called deletions and pops) take place at the same end. This end is called the top; the other end of the list is called the bottom. ■

Figure 8.1(a) shows a stack with four elements. Suppose we wish to add element E to the stack of Figure 8.1(a). This element will have to be placed on top of element D, giving us the configuration of Figure 8.1(b). If we are to delete an element from the stack of Figure 8.1(b), it will be element E. Following the deletion, the configuration of Figure 8.1(a) results. If we perform three successive deletions on the stack of Figure 8.1(b), the stack of Figure 8.1(c) results.

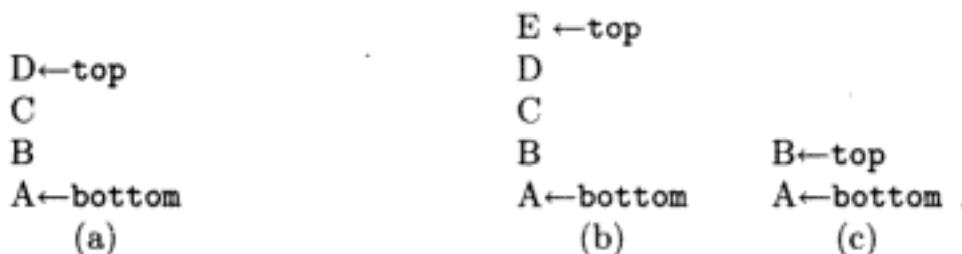


Figure 8.1 Stack configurations

From the preceding discussion, we see that a stack is a LIFO list. Lists of this type appear frequently in computing.

Example 8.1 [Stacks in the Real World]

- If you examine the paper tray of your printer (or copy machine), you will see that the next sheet that gets used is the one at the top; when you add a sheet to the paper tray, you add it to the top. So the paper tray maintains a stack of paper; the paper tray works in a LIFO manner. This LIFO behavior of the paper tray is quite convenient when you want to do that occasional letter on a preprinted letterhead sheet or you want to print the next page on a preprinted form—you simply put the letterhead sheet or form at the top of the paper tray and smile when the printer prints on the desired sheet.
- Walk into a cafeteria, and you'll see a stack of trays. When you get into the food line, you pick up a tray from the top of this stack (unless, of course, you spot a new tray not too far from the top); when the tray stack is replenished, trays are added at the top of the stack. So barring anomalous behavior (like picking up a new tray that is not at the stack top), the tray stack in a cafeteria operates just like the stack data structure we have defined—the tray stack works in a LIFO manner.

- The next time you are in a college bookstore at the start of a term, observe any pile of heavy text books. Each student who needs the book removes and purchases the book at the top of the pile. When the pile gets sufficiently low, a bookstore employee mysteriously appears and adds books to the top of the pile. The book pile works in a LIFO manner—the pile is a stack. ■

Example 8.2 [Recursion] How does your computer run a recursive method? Recursive methods are correctly executed using a **recursion stack**. When a method is invoked, a return address (i.e., the location of the program instruction to execute once the invoked method completes) and the values of all local variables and formal parameters of the invoked method are stored on the recursion stack. When a **return** is executed, the values of local variables and formal parameters are restored to their values prior to the method invocation (these prior values are at the top of the recursion stack) and program execution resumes from the return address, which is also at the top of the stack.

Suppose the recursive sum function (Program 1.31) is invoked from the function `outerFunction` using the statement

```
y = rSum(x, 2);
```

This statement is compiled into code to invoke `rSum` and is followed by code to store the value returned by `rSum` into `y`. Let l_1 be the address of the first instruction in the code to store the returned value into `y`. The `return rSum` statement of Program 1.31 is compiled into code to invoke `rSum`, followed by code to add the returned value to `a[n - 1]`, followed by code to return the result of this addition. Let l_2 be the address of the first instruction in the code to add the returned value to `a[n - 1]`.

When the invocation `rSum(x, 2)` is made from `outerFunction`, the return address (l_1) and the the values of the formal parameters and local variables of `rSum` are saved on the recursion stack as a tuple of the form

$$(\text{return address}, \text{values of formal parameters}, \text{values of local variables})$$

Since `a` and `n` have unspecified values at this time, the tuple $(l_1, *, *)$ (`*` denotes an unspecified value) is added to the recursion stack (note that `rSum` has no local variables) and the formal parameters of `rSum` are assigned their new values. The parameter `a` is assigned the value of `x`, which is a reference to element 0 of the array `x[]`, and `n` is assigned the value 2. Execution continues with the first instruction of `rSum`.

When `rSum` is invoked from within `rSum`, the tuple $(l_2, x, 2)$ is added to the stack; the formal parameters of `rSum` are assigned their new values (`x` and 1); and we continue with the first instruction of `rSum`. The function `rSum` is invoked again from within `rSum`, and $(l_2, x, 1)$ is added to the stack; the formal parameters are assigned their new values (`x` and 0); and we proceed to the first instruction of `rSum`. Now since `n` equals 0, the value 0 is to be returned by `rSum`. How do we know

whether we should return to l_1 or to l_2 ? This determination is made by removing the top tuple from the stack. The stack contents (from bottom to top) are

$$[(l_1, *, *), (l_2, x, 2), (l_2, x, 1)]$$

The top tuple $(l_2, x, 1)$ is removed from the stack, the values of the formal parameters and local variables of the function we are exiting (i.e., `rSum`) are reset (`a` is reset to x and `n` is reset to 1), and we continue with the instruction at l_2 . The sum $0 + x[0]$ is computed, and another return executed. At this time the recursion stack looks like this:

$$[(l_1, *, *), (l_2, x, 2)]$$

The top tuple $(l_2, x, 2)$ is removed from the stack, `a` is reset to x , `n` is reset to 2, the computed sum $0 + a[0]$ is to be returned, and we continue with the instruction at l_2 . This time `a[1]` is added to $0 + a[0]$, the top tuple $(l_1, *, *)$ is removed from the stack, `a` is set to $*$, `n` is set to $*$, the value $0 + a[0] + a[1]$ is to be returned, and we continue at l_1 . ■

EXERCISES

1. The following sequence of operations is done on an initially empty stack: push A , push B , pop, push T , push T , push U , pop, pop, push A , push D . Draw figures similar to those of Figure 8.1 to show the stack configuration after each operation.
2. Do Exercise 1 for the operation sequence: push S , push S , push T , push U , pop, pop, push A , push L , push G , pop, push C , push A , push B , pop, pop.
3. Identify three additional real-world applications of a stack.
4. Show the contents of the recursion stack following each invocation of and each return from the method `rSum` (Program 1.31). The initial invocation is `rSum(x, 3)`.
5. Show the contents of the recursion stack following each invocation of and each return from the method `factorial` (Program 1.29). The initial invocation is `factorial(3)`.
6. Show the contents of the recursion stack following each invocation of and each return from the method `perm` (Program 1.32). The initial invocation is `perm(x, 0, 2)`.

8.2 THE ABSTRACT DATA TYPE

The ADT stack is specified in ADT 8.1. We have chosen the stack operation names to be the same as the method names used in the STL class `stack`.

```
AbstractDataType stack
{
    instances
        linear list of elements; one end is called the bottom; the other is the top;
    operations
        empty() : Return true if the stack is empty, return false otherwise;
        size() : Return the number of elements in the stack;
        top() : Return the top element of the stack;
        pop() : Remove the top element from the stack;
        push(x) : Add element x at the top of the stack;
}
```

ADT 8.1 The abstract data type stack

Program 8.1 gives the C++ abstract class that corresponds to the `stack` abstract data type.

8.3 ARRAY REPRESENTATION

Since a stack is a linear list with the restriction that additions and deletions take place at the same end, we may use any of the linear list representations of Section 5.3.3. When we identify the stack top with the right end of the array linear list, the `push` and `pop` operations correspond to the best case for linear list inserts and erases. Consequently, both operations take $O(1)$ time.

8.3.1 Implementation as a Derived Class

Program 8.2 gives the class `derivedArrayList`, whose base classes are `arrayList` and `stack`.

The constructor for `derivedArrayList` simply invokes the constructor for the base `arrayList`, which allocates a one-dimensional array whose capacity (`length`) is `initialCapacity`. The default value for `initialCapacity` is 10. The codes for the remaining methods of `derivedArrayList` are also straightforward.

```
template<class T>
class stack
{
public:
    virtual ~stack() {}
    virtual bool empty() const = 0;
        // return true iff stack is empty
    virtual int size() const = 0;
        // return number of elements in stack
    virtual T& top() = 0;
        // return reference to the top element
    virtual void pop() = 0;
        // remove the top element
    virtual void push(const T& theElement) = 0;
        // insert theElement at the top of the stack
};
```

Program 8.1 The C++ abstract class stack

Complexity of derivedArrayStack Methods

The complexity of the constructor is $O(1)$ when T is a primitive data type and is $O(\text{initialCapacity})$ when T is a user-defined type. The complexity of `push` is $\Theta(1)$ except when we need to increase the capacity of the stack. In this latter case the complexity is $O(\text{stack size})$. The complexity of each of the remaining methods is $\Theta(1)$.

Comments on derivedArrayStack

The codes for `top` and `pop` check whether the stack is empty before they, respectively, invoke the base class methods `get` and `erase`. Since the invoked base-class methods will throw an exception when invoked with an empty stack, we can eliminate the stack empty check from `top` and `pop` without affecting the program outcome. However, since the `get` and `remove` methods of `arrayList` throw an exception of type `illegalIndex`, the user of our derived stack class will be bewildered upon having an exception of this type thrown when he/she invokes `top` and `pop`. An alternative is to replace the check for an empty stack by a `try-catch` construct in which the `catch` block catches the exception thrown by the base-class method and throws a new and meaningful exception in its place. Program 8.3 shows the code for the method `top` when we use this alternative. The corresponding class is called `derivedArrayStackWithCatch`.

The derivation from `arrayList` has the access modifier `private`. Consequently,

```
template<class T>
class derivedArrayStack : private arrayList<T>,
                           public stack<T>
{
    public:
        derivedArrayStack(int initialCapacity = 10)
            : arrayList<T> (initialCapacity) {}
        bool empty() const
            {return arrayList<T>::empty();}
        int size() const
            {return arrayList<T>::size();}
        T& top()
        {
            if (arrayList<T>::empty())
                throw stackEmpty();
            return get(arrayList<T>::size() - 1);
        }
        void pop()
        {
            if (arrayList<T>::empty())
                throw stackEmpty();
            erase(arrayList<T>::size() - 1);
        }
        void push(const T& theElement)
            {insert(arrayList<T>::size(), theElement);}
};
```

Program 8.2 An array stack class derived from `arrayList`

```
T& top()
{
    try {return get(arrayList<T>::size() - 1);}
    catch (illegalIndex)
        {throw stackEmpty();}
}
```

Program 8.3 Implementation of `top` using the `try-catch` construct

the public and protected methods and data members of `arrayList` are accessible only from within the code for `derivedArrayStack`. In particular, users of the de-

fined stack class are unable to access `arrayList` methods such as `get`, `insert`, and `erase`. Hence the LIFO stack discipline is enforced on instances of type `derivedArrayStack`.

An alternative, but very similar, implementation of an array stack would use a data member `stack` of type `arrayList` and define the stack methods in terms of operations on the linear list `stack`. The code would be quite similar to that of Program 8.2. Alternatively, the data member `stack` could be an array of type `T`, and the code for the `stack` methods would not employ any method of `linearList`. We explore this alternative in the next subsection.

8.3.2 The Class `arrayStack`

When we obtain a stack class by extending a linear list class as was done in Program 8.2, we pay a performance penalty. For example, whenever we add an element to a stack, the `push` method invokes `arrayList<T>::insert`, which does an index check, a possible array doubling, and a `copy_backward` prior to actually inserting the new element. The index check and `copy_backward` are unnecessary because when we add an element to a stack, the element is always added to the right end of the linear list.

One way to arrive at a faster implementation of an array stack is to develop a class that employs an array `stack` to hold the stack elements. Program 8.4 gives the class `arrayStack` that does just this. The bottom element of the stack is stored in `stack[0]`, and the top element in `stack[top]`. The codes for the methods of `arrayStack` may be arrived at from those of `arrayList` by eliminating the redundant code in `arrayList`. The asymptotic complexity of each method of `arrayStack` is the same as that of the corresponding method of `derivedArrayStack`.

8.3.3 Performance Measurement

Even though the array stack classes `arrayStack` (AS), `derivedArrayStack` (DAS), and the C++ STL container class `stack` (STL) implement all methods of the stack ADT so as to have the same asymptotic complexity, the observed performance of the methods is expected to be different for each class.

Define an n -sequence to be a sequence of n `push` operations followed by an alternating sequence of n `top` and n `pop` operations. Figure 8.2 gives the measured times to perform a 50,000,000-sequence, and Figure 8.3 shows these time as a bar chart. For all stack classes except the STL stack class, we obtained the run times for the cases: (1) start with a stack having the default capacity and (2) start with a stack whose initial capacity is 50,000,000. We did not try (2) for the STL stack class because this class does not have a constructor that allows us to specify an initial capacity.

The STL class `stack` took 2 times the time taken by `arrayStack` to perform a 50,000,000-sequence when the initial stack capacity is the default capacity. The performance ratio jumps to 3.7 when we start with an `arrayStack` whose capacity

```
template<class T>
class arrayStack : public stack<T>
{
public:
    arrayStack(int initialCapacity = 10);
    ~arrayStack() {delete [] stack;}
    bool empty() const {return stackTop == -1;}
    int size() const
        {return stackTop + 1;}
    T& top()
    {
        if (stackTop == -1)
            throw stackEmpty();
        return stack[stackTop];
    }
    void pop()
    {
        if (stackTop == -1)
            throw stackEmpty();
        stack[stackTop--].~T(); // destructor for T
    }
    void push(const T& theElement);
private:
    int stackTop;           // current top of stack
    int arrayLength;        // stack capacity
    T *stack;               // element array
};
```

Program 8.4 The class `arrayStack` (continues)

	initialCapacity	
Class	default	50,000,000
arrayStack	2.7	1.5
derivedArrayStack	7.5	6.3
stack	5.6	-

Times are in seconds

Figure 8.2 Time taken by different array stack implementations

```
template<class T>
arrayStack<T>::arrayStack(int initialCapacity)
{// Constructor.
    if (initialCapacity < 1)
        ostringstream s;
        s << "Initial capacity = " << initialCapacity << " Must be > 0";
        throw illegalParameterValue(s.str());
    }
    arrayLength = initialCapacity;
    stack = new T[arrayLength];
    stackTop = -1;
}

template<class T>
void arrayStack<T>::push(const T& theElement)
{// Add theElement to stack.
    if (stackTop == arrayLength - 1)
        {// no space, double capacity
            changeLength1D(stack, arrayLength, 2 * arrayLength);
            arrayLength *= 2;
        }
    // add at stack top
    stack[++stackTop] = theElement;
}
```

Program 8.4 The class `arrayStack` (concluded)

is 50,000,000 (this jump is due mainly to the fact that `stack` does not allow you to specify an initial capacity and so array resizing cannot be avoided). It is more appropriate to compare the performance of the STL class with that of `derivedArrayStack` because both of these classes derive from other concrete linear list classes. The STL class `stack` derives from the STL class `deque` (Exercise 9.9) while `derivedArrayStack` derives from `arrayList`. `stack` has a better performance than `derivedArrayStack` primarily because `deque` does no index checking while `arrayList` does index checking.

An interesting (and expected) observation is that the time spent resizing the array (this is just the difference in the time taken when the initial capacity is 10 and when it is 500,000) is approximately the same for both of stack implementations developed in this section (approximately 1.2 seconds). The time spent on array resizing is about 44 percent of the total time taken by `arrayStack` when we start

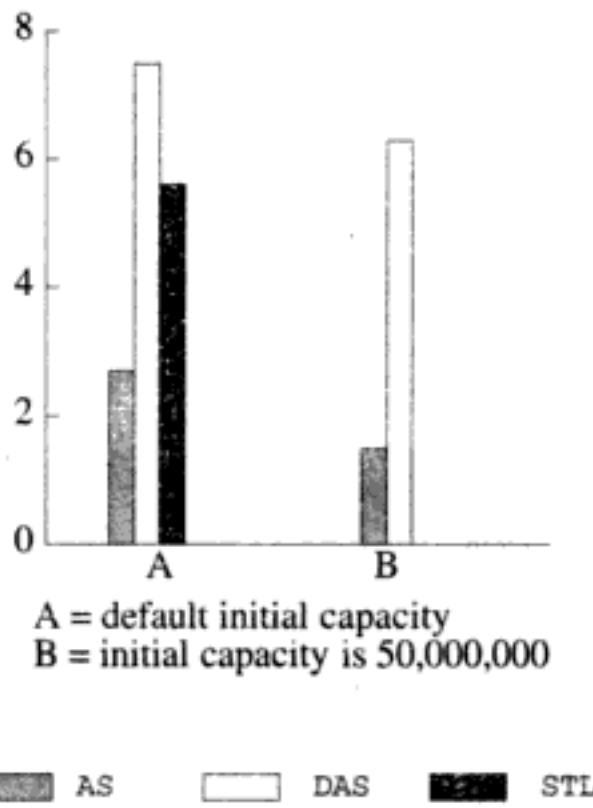


Figure 8.3 Stack run times in seconds

with an array having the default length 10.

EXERCISES

7. (a) Extend the stack ADT by adding functions to
 - i. Input a stack.
 - ii. Convert a stack into a string suitable for output.
 - iii. Split a stack in two. The first contains the bottom half elements, and the second the remaining elements.
 - iv. Combine two stacks by placing all elements of the second stack on top of those in the first. The relative order of elements from the second stack is unchanged.
- (b) Define the abstract class `extendedStack` that extends the abstract class `stack` (Program 8.1) and includes the methods that correspond to the functions of (a).
- (c) Develop the concrete classes `extendedDerivedArrayStack` and `extendedArrayStack` whose base class is `extendedStack` and which also are, respectively, derived from `derivedArrayStack` and `arrayStack`.

- (d) Test the correctness of your codes.
8. Consider the class `arrayStack`. Show that even though it is possible for an individual `push` operation to take $\Theta(\text{stack size})$ time, the time taken by any sequence of n stack operations is $O(n)$.
9. Consider the class `arrayStack`.
- As implemented in Program 8.4, the capacity of a stack (i.e., the length of the array `stack`) can increase but not decrease. To use space more efficiently, modify the implementation of `pop` so that you decrease the capacity to one-half of the current capacity whenever a `pop` reduces the stack occupancy below one-fourth of capacity.
 - Show that even though it is possible for an individual `push` and `pop` operation to take $\Theta(\text{capacity})$ time, the time taken by any sequence of n stack operations is $O(n)$.
10. Develop the concrete class `stackWithArrayList` that derives from the abstract class `stack`. This class has the single data member `list` whose datatype is `arrayList<T>`. Comment on the relative merits of the classes `derivedArrayStack` and `stackWithArrayList`.
11. Develop the C++ class `twoStacks` in which a single array is used to represent two stacks. Peg the bottom of one stack at position 0 and the bottom of the other at position `arrayLength-1`. The two stacks grow toward the middle of the array (see Figure 8.4). Your class must contain methods to perform all operations of the ADT `stack` on each of the two stacks. Further the complexity of each method should be $O(1)$ excluding the time for array resizing.

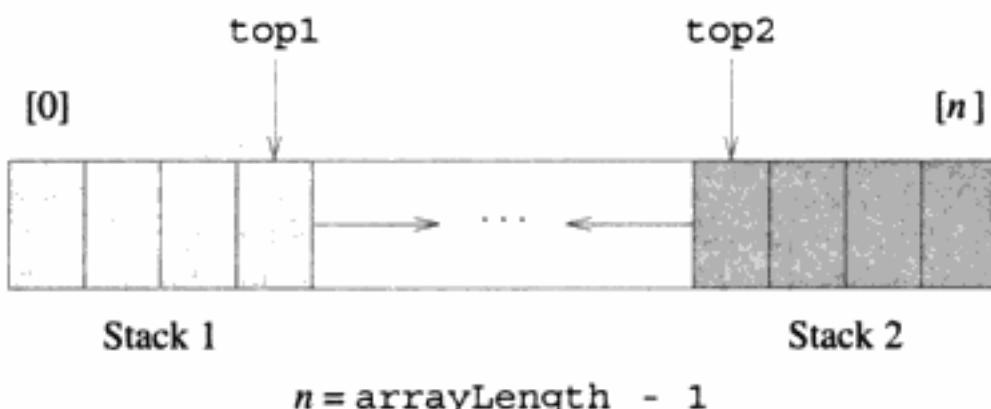


Figure 8.4 Two stacks in an array

8.4 LINKED REPRESENTATION

When using a chain to represent a stack, we must decide which end of the chain corresponds to the stack top. If we associate the right end of the chain with the stack top, then the stack operations `top`, `push`, and `pop` are implemented using invocations to the chain methods `get(size() - 1)`, `insert(size(), theElement)`, and `erase(size() - 1)`. Each of these chain operations takes $O(\text{size}())$ time. On the other hand, if we associate the left end of the chain with the stack top, then the chain operations to use are `get(0)`, `insert(0, theElement)`, and `erase(0)`. Each of these operations takes $\Theta(1)$ time. This analysis shows that we should use the left end of the chain to represent the stack top.

8.4.1 The Class `derivedLinkedList`

The code for the class `derivedLinkedList`, which derives from `chain` (Program 6.2) and which implements the abstract class `stack`, may be obtained from the code for `derivedArrayStack` (Program 8.2) by replacing the clause `private ArrayList<T>` with the clause `private chain<T>`; replacing all occurrences of the name `derivedArrayStack` with the name `derivedLinkedList`; and changing the index actual parameter to all uses of the methods `get`, `insert`, and `erase` to 0 so that these operations take place at the left end of the chain. What could be easier? By using the object-oriented programming principles of information hiding and encapsulation, we have greatly simplified program development. The complexity of each method of `derivedLinkedList` (including the constructor and `push` methods) is $\Theta(1)$.

8.4.2 The Class `linkedStack`

As was the case with the class `arrayStack` (Program 8.4), we can improve the run-time performance by customizing the code and not deriving our linked-stack class from the class `chain`. Program 8.5 gives the customized code.

8.4.3 Performance Measurement

`derivedLinkedList` and `linkedStack` took 41 and 40.5 seconds, respectively, to perform a 50,000,000-sequence (these times were obtained by measuring the times for a 10,000,000-sequence and multiplying by 5). Comparing with the times reported in Figure 8.2, we see that `linkedStack` takes 15 times the time `arrayStack` takes when started with a capacity of 10 and 27 times the time taken when the array stack's initial capacity is 50,000,000.

EXERCISES

12. In some stack applications the elements to be put on a stack are already in nodes of type `chainNode`. For these applications it is desirable to have the

```

template<class T>
class linkedStack : public stack<T>
{
public:
    linkedStack(int initialCapacity = 10)
        {stackTop = NULL; stackSize = 0;}
    ~linkedStack();
    bool empty() const
        {return stackSize == 0;}
    int size() const
        {return stackSize;}
    T& top()
    {
        if (stackSize == 0)
            throw stackEmpty();
        return stackTop->element;
    }
    void pop();
    void push(const T& theElement)
    {
        stackTop = new chainNode<T>(theElement, stackTop);
        stackSize++;
    }
private:
    chainNode<T>* stackTop; // pointer to stack top
    int stackSize;          // number of elements in stack
};

```

Program 8.5 Customized linked stack (continues)

methods `pushNode(chainNode* theNode)`, which adds `theNode` to the top of the stack (notice that no call to `new` is made), and `popNode`, which removes and returns the top node of the stack.

- (a) Write code for these methods
 - (b) Test your code.
 - (c) Compare the performance of a 10,000,000-sequence that uses `pushNode` and `popNode` with a 10,000,000-sequence that uses `push` and `pop`.
13. Develop the concrete class `extendedLinkedStack` that derives from both `linkedStack` and the abstract class `extendedStack` (see Exercise 7).

```
template<class T>
linkedStack<T>::~linkedStack()
{// Destructor.
    while (stackTop != NULL)
        {// delete top node
            chainNode<T>* nextNode = stackTop->next;
            delete stackTop;
            stackTop = nextNode;
        }
}

template<class T>
void linkedStack<T>::pop()
{// Delete top element.
    if (stackSize == 0)
        throw stackEmpty();

    chainNode<T>* nextNode = stackTop->next;
    delete stackTop;
    stackTop = nextNode;
    stackSize--;
}
```

Program 8.5 Customized linked stack (concluded)

14. Compare the performance of the array stack classes used in Figure 8.2 and the linked classes `derivedLinkedStack` and `linkedStack`. Do this by performing an alternating sequence of 10,000,000 `push` and `pop` operations. For the array classes start with the default initial capacity. Does array doubling occur in your experiment? Why?

8.5 APPLICATIONS

8.5.1 Parenthesis Matching

Problem Description

In this problem we are to match the left and right parentheses in a character string. For example, the string `(a*(b+c)+d)` has left parentheses at positions 0 and 3 and right parentheses at positions 7 and 10. The left parenthesis at position 0 matches the right at position 10, while the left parenthesis at position 3 matches the right parenthesis at position 7. In the string `(a+b))()`, the right parenthesis at position

5 has no matching left parenthesis, and the left parenthesis at position 6 has no matching right parenthesis. Our objective is to write a C++ program that inputs a string and outputs the pairs of matched parentheses as well as those parentheses for which there is no match. Notice that the parenthesis matching problem is equivalent to the problem of matching braces ({ and }) in a C++ program.

Solution Strategy

We observe that if we scan the input expression from left to right, then each right parenthesis is matched to the most recently seen unmatched left parenthesis. This observation motivates us to save the position of left parentheses on a stack as they are encountered in a left-to-right scan. When a right parenthesis is encountered, it is matched to the left parenthesis (if any) at the top of the stack. The matched left parenthesis is deleted from the stack.

C++ Implementation

Program 8.6 gives the C++ code.

Complexity

The time complexity of Program 8.6 is $O(n)$ where n is the length of the input expression. To see this, note that the program performs $O(n)$ push and $O(n)$ pop operations. Even though the complexity of an individual push operation is $O(\text{capacity})$ (because of array doubling), the complexity of $O(n)$ push operations is $O(n)$. The complexity of each pop operation is $O(1)$. Therefore, the complexity of $O(n)$ pop operations is $O(n)$.

8.5.2 Towers of Hanoi

Problem Description

The **Towers of Hanoi** problem is fashioned after the ancient Tower of Brahma ritual. According to legend, when the world was created, there was a diamond tower (tower 1) with 64 golden disks (Figure 8.5). The disks were of decreasing size and were stacked on the tower in decreasing order of size from bottom to top. Next to this tower are two other diamond towers (towers 2 and 3). Since the time of creation, Brahman priests have been attempting to move the disks from tower 1 to tower 2, using tower 3 for intermediate storage. As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk. According to legend, the world will come to an end when the priests have completed their task.

In the Towers of Hanoi problem, we are given n disks and three towers. The disks are initially stacked on tower 1 in decreasing order of size from bottom to top. We are to move the disks to tower 2, one disk at a time, such that no disk is ever

```
void printMatchedPairs(string expr)
{// Parenthesis matching.
    arrayStack<int> s;
    int length = (int) expr.size();

    // scan expression expr for ( and )
    for (int i = 0; i < length; i++)
        if (expr.at(i) == '(')
            s.push(i);
        else
            if (expr.at(i) == ')')
                try
                    {// remove location of matching '(' from stack
                     cout << s.top() << ' ' << i << endl;
                     s.pop(); // unstack match
                }
                catch (stackEmpty)
                    {// stack was empty, no match exists
                     cout << "No match for right parenthesis"
                         << " at " << i << endl;
                }
            }

    // remaining ')' in stack are unmatched
    while (!s.empty())
    {
        cout << "No match for left parenthesis at "
            << s.top() << endl;
        s.pop();
    }
}
```

Program 8.6 Program to output matched parentheses

on top of a smaller one. You may wish to attempt a solution to this problem for $n = 2, 3$, and 4 before reading further.

Solution Strategy

A very elegant solution results from the use of recursion. To get the largest disk to the bottom of tower 2, we move the remaining $n - 1$ disks to tower 3 and then move the largest to tower 2. Now we are left with the task of moving the $n - 1$ disks from tower 3 to tower 2. To perform this task, we can use towers 1 and 2. We can

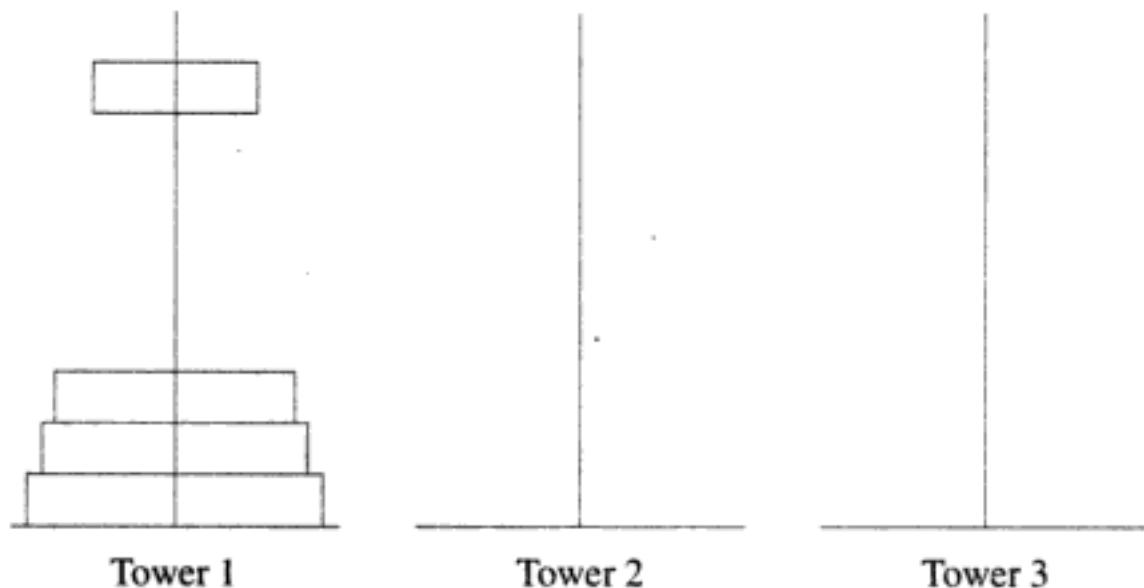


Figure 8.5 Towers of Hanoi

safely ignore the fact that tower 2 has a disk on it because this disk is larger than the disks being moved from tower 3. Therefore, we can place any disk on top of the disk on tower 2.

First Implementation

Program 8.7 gives recursive C++ code for this solution. The initial invocation is `towersOfHanoi(n, 1, 2, 3)`. The correctness of Program 8.7 is easily established.

```
void towersOfHanoi(int n, int x, int y, int z)
{// Move the top n disks from tower x to tower y.
// Use tower z for intermediate storage.
if (n > 0)
{
    towersOfHanoi(n-1, x, z, y);
    cout << "Move top disk from tower " << x
        << " to top of tower " << y << endl;
    towersOfHanoi(n-1, z, y, x);
}
}
```

Program 8.7 Recursive method for Towers of Hanoi

Complexity

The time taken by Program 8.7 is proportional to the number of lines of output generated, and the number of lines output is equal to the number of disk moves performed. Examining Program 8.7, we obtain the following recurrence for the number of moves, $\text{moves}(n)$:

$$\text{moves}(n) = \begin{cases} 0 & n = 0 \\ 2\text{moves}(n - 1) + 1 & n > 0 \end{cases}$$

This recurrence may be solved by using the substitution method of Chapter 2 (see Example 2.20). The result is $\text{moves}(n) = 2^n - 1$. We can show that $2^n - 1$ is, in fact, the least number of moves in which the disks can be moved. Since $n = 64$ in the Tower of Brahma, the Brahman priests will need quite a few years to finish their task. From the solution to the above recurrence, we conclude that the time complexity of `towersOfHanoi` is $\Theta(2^n)$ provided the method runs to completion.

Second Implementation

The output from Program 8.7 gives us the disk-move sequence needed to move the disks from tower 1 to tower 2. Suppose we wish to show the state (i.e., the disks together with their order bottom to top) of the three towers following each move. To show this state, we must store the state of the towers in memory and change the state of each as disks are moved. Following each move, we can output the tower states to an output device such as the computer screen, printer, or video recorder.

Since disks are removed from each tower in a LIFO manner, each tower may be represented as a stack. The three towers together contain exactly n disks at any time. Using linked stacks, we can get by with space for n elements. If array stacks are used, towers 1 and 2 must have a capacity of n disks each, while tower 3 must have a capacity of $n - 1$. Therefore, we need space for a total of $3n - 1$ disks. As our earlier analysis has shown, the time complexity of the Towers of Hanoi problem is exponential in n . So using a reasonable amount of computer time, the problem can be solved only for small values of n (say $n \leq 30$). For these small values of n , the difference in space required by the array and linked representations is sufficiently small that either may be used. Since the array implementations of a stack run faster than the linked implementations, we use an array implementation.

The code of Program 8.8 uses array stacks. `towersOfHanoi(n)` is just a preprocessor for the recursive method `moveAndShow`, which is modeled after the method of Program 8.7. The preprocessor creates the three stacks `tower[1:3]` that will store the states of the three towers. The disks are numbered 1 (smallest) through n (largest). Since the disks are modeled as integers, the data type for the stack elements is `int`. The initial configuration has all n disks in `tower[1]`; the remaining two towers have no disk. After constructing this initial configuration, the preprocessor invokes the method `moveAndShow`.

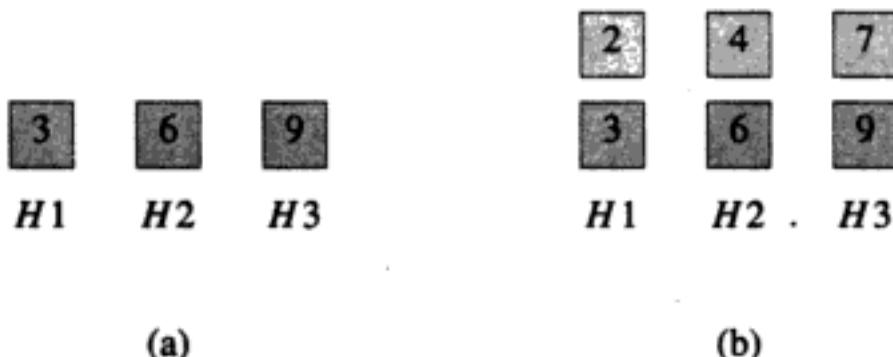


Figure 8.7 Track states

track are not in increasing order from top to bottom, the rearrangement cannot be completed. The current state of the holding tracks is shown in Figure 8.7(a).

Car 2 is considered next. It can be moved into any of the holding tracks while satisfying the requirement that car labels in any holding track be in increasing order, but moving it to $H1$ is preferred. If car 2 is moved to $H3$, then we have no place to move cars 7 and 8. If we move it to $H2$, then the next car, car 4, will have to be moved to $H3$ and we will have no place for cars 5, 7, and 8. *The least restrictions on future car placement arise when the new car u is moved to the holding track that has at its top a car with smallest label v such that $v > u$.* We will use this **assignment rule** to select the holding track.

When car 4 is considered, the cars at the top of the three holding tracks are 2, 6, and 9. Using our assignment rule, car 4 is moved to $H2$. Car 7 is then moved to $H3$. Figure 8.7(b) shows the current state of the holding tracks. The next car, car 1, is moved to the output track. It is now time to move car 2 from $H1$ to the output track. Next car 3 is moved from $H1$, and then car 4 is moved from $H2$. No other cars can be moved to the output at this time.

The next input car, car 8, is moved to $H1$. Then car 5 is moved from the input track to the output track. Following this move, car 6 is moved from $H2$. Then car 7 is moved from $H3$, car 8 from $H1$, and car 9 from $H3$.

While three holding tracks are sufficient to rearrange the cars from the initial ordering of Figure 8.6(a), other initial arrangements may need more tracks. For example, the initial arrangement $1, n, n - 1, \dots, 2$ requires $n - 1$ holding tracks.

C++ Implementation

We use k array stacks, $\text{track}[1:k]$, to represent the k holding tracks. Array stacks are used because they are faster than linked stacks. Program 8.9 gives the global variables we use.

The function `railroad` (Program 8.10) determines a sequence of moves that results in rearranging cars with initial ordering `inputOrder[1:theNumberOfCars]`

```
arrayStack<int> *track; // array of holding tracks
int numberOfCars;
int numberOfTracks;
int smallestCar; // smallest car in any holding track
int itsTrack; // holding track with car smallestCar
```

Program 8.9 Global variables for railroad car problem

using at most `theNumberOfTracks` holding tracks. If such a sequence does not exist, `railroad` returns `false`. Otherwise, it returns `true`.

Function `railroad` begins by creating an array `track` of stacks. `track[i]` represents holding track i , $1 \leq i \leq \text{numberOfTracks}$. The `for` loop maintains the invariant: *at the start of this loop, the car with label `nextCarToOutput` is not in a holding track.*

In iteration i of the `for` loop, car `inputOrder[i]` is moved from the input track. This car is to move to the output track only if `inputOrder[i]` equals `nextCarToOutput`. If car `inputOrder[i]` is moved to the output track, `nextCarToOutput` increases by one, and it may be possible to move one or more of the cars in the holding tracks. These cars are moved to the output by the `while` loop. If car `inputOrder[i]` cannot be moved to the output, then no car can be so moved. Consequently, car `inputOrder[i]` is added to a holding track using the stated track assignment rule.

Programs 8.11 and 8.12, respectively, give the functions `outputFromHoldingTrack` and `putInHoldingTrack` utilized by `railroad`. `outputFromHoldingTrack` outputs instructions to move a car from a holding track to the output track. It also updates `smallestCar` and `itsTrack`. The method `putInHoldingTrack` puts car c into a holding track using the stated track assignment rule. It also outputs instructions to move the car to the chosen holding track and updates `smallestCar` and `itsTrack` if necessary.

Complexity

For the time complexity of `railroad` (Program 8.10), we first observe that both `outputFromHoldingTrack` and `putInHoldingTrack` have complexity $O(\text{numberOfTracks})$. Since at most `numberOfCars-1` cars can be output from the `while` loop of `railroad` and at most `numberOfCars-1` put into holding tracks from the `else` clause, the total time spent in the functions `outputFromHoldingTrack` and `putInHoldingTrack` is $O(\text{numberOfTracks} * \text{numberOfCars})$. The remainder of the `for` loop of `railroad` takes $\Theta(\text{numberOfCars})$ time. So the overall complexity of Program 8.10 is $O(\text{numberOfTracks} * \text{numberOfCars})$. This complexity can be reduced to $O(\text{numberOfCars} * \log(\text{numberOfTracks}))$ by using a balanced binary search tree (such as an AVL tree) to store the labels of the cars at the top of

```
bool railroad(int inputOrder[],
              int theNumberOfCars, int theNumberOfTracks)
{// Rearrange railroad cars beginning with the initial order.
// Return true if successful, false if impossible.

    numberOfCars = theNumberOfCars;
    numberOfTracks = theNumberOfTracks;

    // create stacks for use as holding tracks
    track = new arrayStack<int> [numberOfTracks + 1];

    int nextCarToOutput = 1;
    smallestCar = numberOfCars + 1; // no car in holding tracks

    // rearrange cars
    for (int i = 1; i <= numberOfCars; i++)
        if (inputOrder[i] == nextCarToOutput)
            // send car inputOrder[i] straight out
            cout << "Move car " << inputOrder[i]
                << " from input track to output track" << endl;
            nextCarToOutput++;

            // output from holding tracks
            while (smallestCar == nextCarToOutput)
            {
                outputFromHoldingTrack();
                nextCarToOutput++;
            }
        }
    else
        // put car inputOrder[i] in a holding track
        if (!putInHoldingTrack(inputOrder[i]))
            return false;

    return true;
}
```

Program 8.10 The function railroad

the holding tracks (see Chapter 15). When a balanced binary search tree is used in this way, the functions `outputFromHoldingTrack` and `putInHoldingTrack` can

```
void outputFromHoldingTrack()
{// Output the smallest car from the holding tracks.

    // remove smallestCar from itsTrack
    track[itsTrack].pop();
    cout << "Move car " << smallestCar << " from holding "
        << "track " << itsTrack << " to output track" << endl;

    // find new smallestCar and itsTrack by checking top of all stacks
    smallestCar = numberofCars + 2;
    for (int i = 1; i <= numberofTracks; i++)
        if (!track[i].empty() && (track[i].top() < smallestCar))
    {
        smallestCar = track[i].top();
        itsTrack = i;
    }
}
```

Program 8.11 The function `outputFromHoldingTrack`

be rewritten to have complexity $O(\log(\text{numberofTracks}))$. The use of a balanced binary search tree for this application is recommended only when `numberofTracks` is large.

8.5.4 Switch Box Routing

Problem Description

In the switch box routing problem, we are given a rectangular routing region with pins at the periphery. Pairs of pins are to be connected together by laying a metal path between the two pins. This path is confined to the routing region and is called a wire. If two wires intersect, an electrical short occurs. So wire intersections are forbidden. Each pair of pins that is to be connected is called a **net**. We are to determine whether the given nets can be routed with no intersections. Figure 8.8(a) shows a sample switch box instance with eight pins and four nets. The nets are (1, 4), (2, 3), (5, 6), and (7, 8). The wire routing of Figure 8.8(b) has a pair of intersecting wires (those for nets (1, 4) and (2, 3)), whereas the routing of Figure 8.8(c) has no intersections. Since the four nets can be routed with no intersections, the given switch box is a **routable switch box**. (In practice, we also require a minimum separation between adjacent wires. We ignore this additional requirement here.) Our problem is to input a switch box routing instance and determine whether it is routable.

```
bool putInHoldingTrack(int c)
{// Put car c into a holding track. Return false iff there is
// no feasible holding track for this car.

    // find best holding track for car c
    // initialize
    int bestTrack = 0,                      // best track so far
        bestTop = numberOfWorks + 1; // top car in bestTrack

    // scan tracks
    for (int i = 1; i <= numberOfWorks; i++)
        if (!track[i].empty())
            {// track i not empty
                int topCar = track[i].top();
                if (c < topCar && topCar < bestTop)
                    {// track i has smaller car at top
                        bestTop = topCar;
                        bestTrack = i;
                    }
            }
        else // track i empty
            if (bestTrack == 0) bestTrack = i;

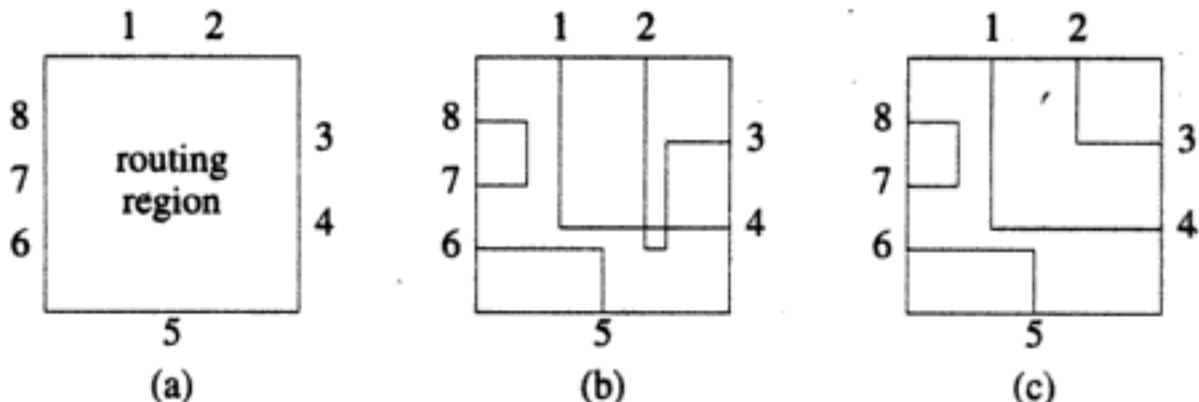
    if (bestTrack == 0) return false; // no feasible track

    // add c to bestTrack
    track[bestTrack].push(c);
    cout << "Move car " << c << " from input track "
        << "to holding track " << bestTrack << endl;

    // update smallestCar and itsTrack if needed
    if (c < smallestCar)
    {
        smallestCar = c;
        itsTrack = bestTrack;
    }

    return true;
}
```

Program 8.12 The function putInHoldingTrack

**Figure 8.8** Sample switch box

While the wires in both Figures 8.8(b) and (c) are composed of straight line segments parallel to the x - and y -axes, segments that are not parallel to these axes as well as segments that are not straight lines are permissible.

Solution Strategy

To solve the switch box routing problem, we note that when a net is connected, the wire partitions the routing region into two regions. The pins that fall on the boundary of a partition do not depend on the wire path, but only on the pins of the net that was routed. For instance, when net (1, 4) is routed, we get two regions. One contains the pins 2 and 3, and the other contains the pins 5 through 8. If there is now a net with one pin in one region and the other in a different region, this new net cannot be routed and the routing instance is unroutable. If there is no net with this property, then since the wires cannot cross between regions, we can attempt to determine whether each region is independently routable. To make this determination, we pick a net in one of the regions; this net partitions its region into two regions, and none of the remaining nets should have a pin in one partition and another in the other partition.

We can implement this strategy by moving around the periphery of the switch box in either clockwise or counterclockwise order, beginning at any pin. If we traverse the pins of Figure 8.8(a) in clockwise order, beginning at pin 1, the pins are examined in the order, 1, 2, ..., 8. The pins that lie between pin 1 and its net partner, pin 4, define one region of the first partition, and those that lie between pins 4 and 1 define the other. We will place pin 1 on a stack and continue processing pins until pin 4 is encountered. This procedure allows us to process one of the regions before going on to the other. The next pin, pin 2, and its net partner, pin 3, partition the current region into two regions. As before, pin 2 is placed on the stack, and we proceed to pin 3. Since pin 3's partner, pin 2, is at the top of the stack, we have completed a region and pin 2 is deleted from the stack. Next we

encounter pin 4 whose partner is now at the top of the stack. The processing of a region is now complete, and pin 1 is deleted from the stack. Proceeding in this way, we are able to complete the processing of all created regions, and the stack is empty after pin 8 is examined.

What happens on a nonroutable instance? Suppose the nets for Figure 8.8(a) are (1, 5), (2, 3), (4, 7), and (6, 8). Pins 1 and 2 are put on the stack initially. When pin 3 is examined, pin 2 is deleted from the stack. Next pin 4 is added to the stack, as pin 4 and the pin at the stack top do not define a region boundary. When pin 5 is examined, it is also added to the stack. Even though pins 1 and 5 have both been seen, we are unable to complete the processing of the first region defined by this net, as pin 4's routing has to cross the boundary. As a result, when we complete the examination of all pins, the stack will not be empty.

C++ Implementation and Complexity

Program 8.13 gives a C++ function that implements this strategy. This function assumes that the number of pins is even and that each pin has a net number. So for the example in Figure 8.8(c), the input array `net` is [1, 2, 2, 1, 3, 3, 4, 4]. The complexity of the program is $O(n)$ where n is the number of pins.

8.5.5 Offline Equivalence Class Problem

Problem Description

The offline equivalence problem was defined in Section 6.5.4. The inputs to this problem are the number of elements n , the number of relation pairs r , and the r relation pairs. We are to partition the n elements into equivalence classes.

Solution Strategy

The solution is in two phases. In the first phase we input the data and set up n lists to represent the relation pairs. For each relation pair (i, j) , i is put on $list[j]$ and j is put on $list[i]$.

Example 8.3 Suppose that $n = 9$, $r = 11$, and the 11 relation pairs are (1, 5), (1, 6), (3, 7), (4, 8), (5, 2), (6, 5), (4, 9), (9, 7), (7, 8), (3, 4), and (6, 2). The nine lists are

```

list[1] = [5, 6]
list[2] = [5, 6]
list[3] = [7, 4]
list[4] = [8, 9, 3]
list[5] = [1, 2, 6]

```

```
bool checkBox(int net[], int n)
{// Determine whether the switch box is routable.
// net[0..n-1] is array of pin to net assignments.
// n is number of pins.

arrayStack<int>* s = new arrayStack<int>(n);

// scan nets clockwise
for (int i = 0; i < n; i++)
    // process pin i
    if (!s->empty())
        // check with top net
        if (net[i] == net[s->top()])
            // net[i] is routable, delete from stack
            s->pop();
        else s->push(i);
    else s->push(i);

// any unrouted nets left?
if (s->empty())
    // no nets remain
    cout << "Switch box is routable" << endl;
    return true;
}

cout << "Switch box is not routable" << endl;
return false;
}
```

Program 8.13 Switch box routing

<i>list[6]</i>	=	[1, 2, 5]
<i>list[7]</i>	=	[3, 9, 8]
<i>list[8]</i>	=	[4, 7]
<i>list[9]</i>	=	[4, 7]

Element order within a list is not important.



In the second phase, the equivalence classes are identified by first locating an element that has not been output as part of an equivalence class. This element becomes the seed for the next equivalence class. The seed is output as the first

member of the next equivalence class. From the seed we identify all other members of the class as follows. The seed is put onto a list, *unprocessedList*, of elements that are in the same equivalence class as the seed and whose lists have yet to be processed. We remove an element i from *unprocessedList* and process *list*[i]. All elements on *list*[i] are in the same equivalence class as the seed; elements on *list*[i] that haven't already been identified as class members are output and added to *unprocessedList*. This process of removing an element i from *unprocessedList* and then outputting and adding elements in *list*[i] that haven't already been output to *unprocessedList* continues until the *unprocessedList* becomes empty. At this time we have completed a class, and we proceed to find a seed for the next class.

Example 8.4 Consider the data of Example 8.3. Let 1 be the first seed; 1 is output as part of a new class and is also added to *unprocessedList*. Next 1 is removed from *unprocessedList*, and *list*[1] is processed. The elements 5 and 6 that are in *list*[1] are output as part of the same class as element 1; 5 and 6 are also added to *unprocessedList*. Either 5 or 6 is removed from *unprocessedList*, and its list is processed. Suppose that 5 is removed. The elements 1, 2, and 6 that are in *list*[5] are examined. Since 1 and 6 have already been output, we ignore them. Element 2 is output and added to *unprocessedList*. When the remaining elements (6 and 2) that are in *unprocessedList* are removed and processed, no additional element is output or added to *unprocessedList*; this list becomes empty, and we have identified an equivalence class.

To find another equivalence class, we search for a seed—an element not yet output. Element 3 has not been output and is used as the seed for the next class. Elements 3, 4, 7, 8, and 9 are output as part of this next class. Since no seeds remain, we have found all the classes. ■

C++ Implementation

To proceed with an implementation, we must select a representation for *list* and *unprocessedList*. The operations performed on *list* are to insert and examine all elements. Since it doesn't matter where elements are inserted in *list*, any linear list or stack representation may be used. We select a specific linear list or stack representation based on which is expected to provide the best space and time performance.

The total number of elements in all n of the lists *list*[1 : n] is $2r$. Therefore, as far as space requirements go, all our array linear list and stack classes require space for between $2r$ and $4r$ elements (because of array doubling, the allocated array length may be up to two times the number of elements). Our linked classes require space for $2r$ elements and $2r$ pointers. Our run-time performance studies of linear list and stack implementations (see Sections 5.6, 6.1.6, 8.3.3, and 8.4.3) show that the linked implementations of these data structures are slower than their array counterparts. So we eliminate the linked representations from further consideration as far the offline equivalence class problem is concerned.

By inserting new elements at the right end of a list, our application will exhibit the best-case time performance for `arrayList` (Section 5.3). However, if we use an `arrayStack`, we will do slightly better. For performance reasons `unprocessedList` is also implemented as an `arrayStack`.

Program 8.14 gives a two-part program for the offline equivalence problem. In the first part, n , r , and the r pairs are input, and the stack (list) for each of the n elements constructed. The stack `list[i]` for element i contains all elements j such that (i, j) or (j, i) is an input relation pair. Our code can be made more robust by verifying that every pair (a, b) that is input has both a and b in the range $[1, n]$. Exercise 31 asks you to modify the code so as to validate the input relations.

The second part of the program outputs the equivalence classes. For the second part we maintain an array `out` such that `out[i] = true` iff element i has been output as a member of some equivalence class. A stack `unprocessedList` assists in locating all elements of an equivalence class. This stack holds all elements that have been output as part of the current class and that may lead to additional elements of the class. To find the seed for the next equivalence class, we scan the array `out` for an element not yet output. If there is no such element, then there is no next class. If such an element is found, it begins the next class.

Complexity

For the complexity analysis, we assume that no exceptions are thrown during execution (in particular, the input pairs (a, b) are valid). Part 1 of the program (input and initialize the array `list[]` of relation pairs) takes $\Theta(n+r)$ time. For part 2, we note that since each of the n elements is output exactly once, each is added to `unprocessedList` once and deleted from `unprocessedList` once. So the total time spent pushing and popping elements from `unprocessedList` is $\Theta(n)$. Finally, when an element j is removed from `unprocessedList`, all elements on `list[j]` are examined by popping them off of `list[j]`. Each element in each `list[j]` is popped exactly once. So the time required to pop and examine all elements on all lists `list[1:n]` is $\Theta(r)$ (note that the total number of elements on all lists `list[1:n]` is $2 * r$ following the input phase). Allowing for the possibility that an exception may occur, we conclude that the overall complexity of Program 8.14 is $O(n+r)$. The complexity when no exception occurs is $\Theta(n+r)$.

Since every program for the offline equivalence class problem must examine each relation and element at least once, it is not possible to solve the offline equivalence problem in less than $O(n+r)$ time.

8.5.6 Rat in a Maze

Problem Description

A **maze** (Figure 8.9) is a rectangular area with an entrance and an exit. The interior of the maze contains walls or obstacles that one cannot walk through. In

```
int main()
{
    int n,    // number of elements
        r;    // number of relations

    cout << "Enter number of elements" << endl;
    cin >> n;
    if (n < 2)
    {
        cout << "Too few elements" << endl;
        return 1;    // terminate with error
    }

    cout << "Enter number of relations" << endl;
    cin >> r;
    if (r < 1)
    {
        cout << "Too few relations" << endl;
        return 1;    // terminate with error
    }

    // create an array of empty stacks, stack[0] not used
    arrayStack<int>* list = new arrayStack<int> [n+1];

    // input the r relations and put on lists.
    int a, b;    // (a, b) is a relation
    for (int i = 1; i <= r; i++)
    {
        cout << "Enter next relation/pair" << endl;
        cin >> a >> b;
        list[a].push(b);
        list[b].push(a);
    }
}
```

Program 8.14 Offline equivalence class program (continues)

our mazes these obstacles are placed along rows and columns that are parallel to the rectangular boundary of the maze. The entrance is at the upper-left corner, and the exit is at the lower-right corner.

Suppose that the maze is to be modeled as an $n \times m$ matrix with position (1,1) of the matrix representing the entrance and position (n, m) representing the

```
// initialize to output equivalence classes
arrayStack<int> unprocessedList;
bool* out = new bool[n + 1];
for (int i = 1; i <= n; i++)
    out[i] = false;

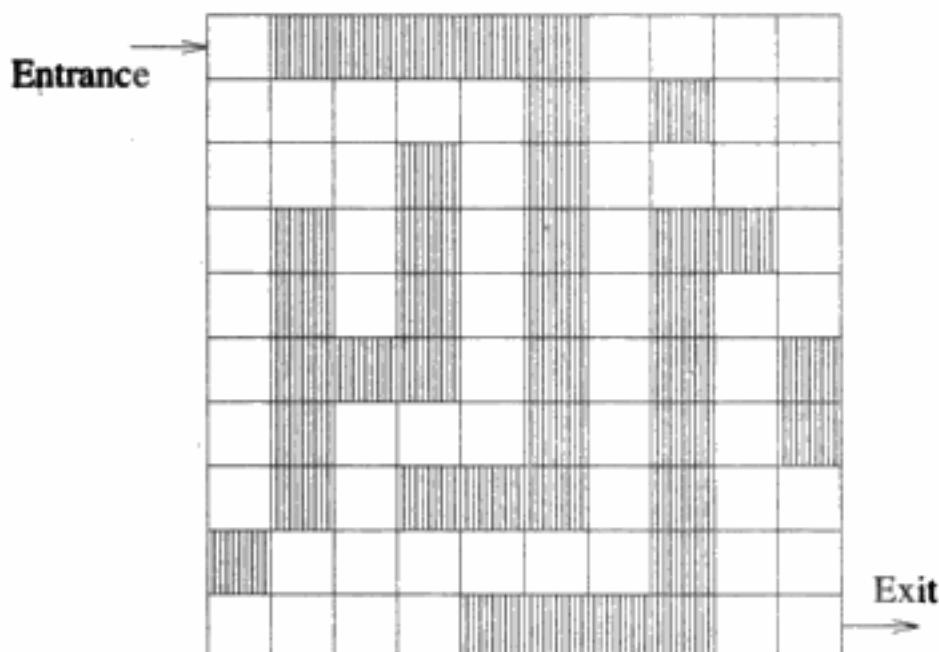
// output equivalence classes
for (int i = 1; i <= n; i++)
    if (!out[i])
        {// start of a new class
            cout << "Next class is: " << i << " ";
            out[i] = true;
            unprocessedList.push(i);
            // get rest of class from unprocessedList
            while (!unprocessedList.empty())
            {
                int j = unprocessedList.top();
                unprocessedList.pop();

                // elements on list[j] are in the same class
                while (!list[j].empty())
                {
                    int q = list[j].top();
                    list[j].pop();
                    if (!out[q]) // q not yet output
                    {
                        cout << q << " ";
                        out[q] = true;
                        unprocessedList.push(q);
                    }
                }
            }
            cout << endl;
        }

cout << "End of list of equivalence classes" << endl;

return 0;
}
```

Program 8.14 Offline equivalence class program (concluded)

**Figure 8.9** A maze

0	1	1	1	1	1	0	0	0	0
0	0	0	0	0	1	0	1	0	0
0	0	0	1	0	1	0	0	0	0
0	1	0	1	0	1	0	1	1	0
0	1	0	1	0	1	0	1	0	0
0	1	1	1	0	1	0	1	0	1
0	1	0	0	0	1	0	1	0	1
0	1	0	1	1	1	0	1	0	0
1	0	0	0	0	0	0	1	0	0
0	0	0	0	1	1	1	1	0	0

Figure 8.10 Matrix representation of maze of Figure 8.9

exit. n and m are, respectively, the number of rows and columns in the maze. Each maze position is described by its row and column intersection. The matrix has a 1 in position (i, j) iff there is an obstacle at the corresponding maze position. Otherwise, there is a 0 at this matrix position. Figure 8.10 shows the matrix representation of the maze of Figure 8.9. The **rat-in-a-maze** problem is to find a path from the entrance to the exit of a maze. A **path** is a sequence of positions, none of which is blocked, and such that each (other than the first) is the north, south, east, or west neighbor of the preceding position (Figure 8.11).

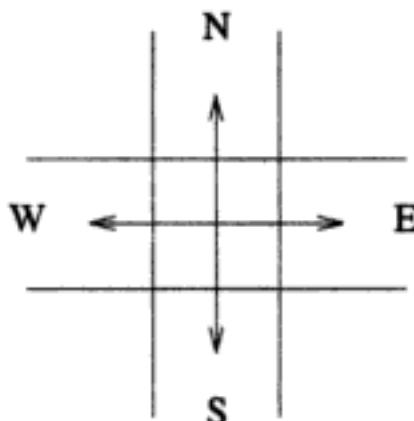


Figure 8.11 The four options for a move from any position in the maze

You are to write a program to solve the rat-in-a-maze problem. You may assume that the mazes for which your program is to work are square (i.e., $m = n$) and are sufficiently small so that the entire maze can be represented in the memory of the target computer. Your program will be a stand-alone product that will be used directly by persons wishing to find a path in a maze of their choice.

Design

We will use the top-down modular methodology to design the program. It is not too difficult to see the three aspects to the problem: input the maze, find a path, and output the path. We will use one program module for each task. A fourth module that displays a welcome message and identifies the program and its author is also desirable. While this module is not directly related to the problem at hand, the use of such a module enhances the user-friendliness of the program.

The module that finds the path does not interact directly with the user and will therefore contain no help facility and will not be menu driven. The remaining three modules interact with the user, and we need to expend some effort designing their user interface. The user interface should make the user want to use your program rather than competing programs.

Let us begin with the welcome module. We wish to display a message such as

**Welcome To
RAT IN A MAZE
©Joe Bloe, 2000**

While displaying this message might seem like a trivial task, we can use various design elements to obtain a pleasing effect. For example, the message can be multi-colored to take advantage of the user's color display. The three lines of the welcome display need to be positioned on the screen, and we can change the character size

from one line to the next (or even from character to character). The welcome message can be introduced on the display with a reasonable time lapse between the introduction of one character and the next. Alternatively, the time lapse can be very small. In addition, we might consider the use of sound effects. We also need to determine the duration for which the message is to be displayed. It should be displayed long enough so that the user can read it, but not long enough to leave the user yawning. As you can see, the design of the welcome message (and the whole user interface in general) requires strong artistic skills.

For the input module we must decide whether we want the input as a matrix of 0s and 1s or whether we will display a maze of the desired size and then ask the user to click a mouse at the squares that contain an obstacle. We must also decide on the colors to use, whether we will have audio during input, and so on.

The input module can also verify that the entrance and exit of the maze are not blocked. If they are, then no path exists. In all likelihood the user made an error in input. The following discussion assumes that the input module performs this verification and that the entrance and exit are not blocked.

Once again, we see that what initially appeared to be a simple task (read in a matrix) is actually quite complex if we want to do it in a user-friendly way.

The output module design involves essentially the same considerations as the design of the input module.

Program Plan

The design phase has already pointed out the need for four program modules. We also need a root (or main) module that invokes these four modules in the following sequence: welcome module, input module, find path module, and output module.

Our program will have the modular structure of Figure 8.12. Each program module can be coded independently. The root module will be coded as the method `main`; the welcome, input, find path, and output path modules will each be a single private method.

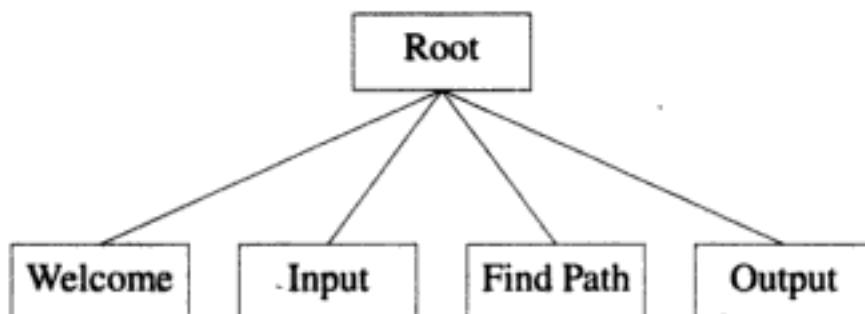


Figure 8.12 Modular structure of rat-in-a-maze program

At this point we see that our program is going to have the form given in Figure 8.13.

```
// function welcome comes here
// function inputMaze comes here
// function findPath comes here
// function outputPath comes here

void main()
{
    welcome();
    inputMaze();
    if (findPath())
        outputPath();
    else
        cout << "No path" << endl;
}
```

Figure 8.13 Form of rat-in-a-maze program

Program Development

Substantial data structure and algorithm issues arise in the development of the path-finding module only. Consequently, we will develop just this module here. Exercise 33 asks you to develop the remaining modules. Without thinking too much about the coding of the path-finding module, we can arrive at the C++ pseudocode given in Figure 8.14. This code is readily seen to be correct. Unfortunately, we cannot present it to a computer in this form, and we need to refine the pseudocode into pure C++ code.

```
bool findPath()
{
    Search the maze for a path to the exit;
    if (a path is found) return true;
    else return false;
}
```

Figure 8.14 First version of findPath

Before attempting a refinement of Figure 8.14 that will get us closer to C++

code, let us figure out how we are to search the maze for a path. We begin with the entrance as our present position. If the present position is the exit, then we have found a path and we are done. If we are not at the exit, then we block the present position (i.e., place an obstacle there) so as to prevent the search from returning here. Next we see whether there is an adjacent maze position that is not blocked. If so, we move to this new adjacent position and attempt to find a path from there to the exit. If we are unsuccessful, we attempt to move to some other unblocked adjacent maze position and try to find a path from there. To facilitate this move, we save the current position on a stack before advancing to a new adjacent position. If all adjacent unblocked positions have been tried and no path is found, there is no path from entrance to exit in the maze.

Let us use the above strategy on the maze of Figure 8.9. We begin with the position (1,1) on the stack and move to its only unblocked neighbor (2,1). The position (1,1) is blocked to prevent the search path from moving through this position later. From (2,1) we can move to (3,1) or (2,2). Suppose we decide to move to (3,1). Prior to the move, we block (2,1) and add it to the stack. From (3,1) we may move to either (4,1) or (3,2). If we move to (4,1), (4,1) gets blocked and added to the stack. From (4,1) we move to (5,1), (6,1), (7,1), and (8,1). The path cannot be extended from (8,1). Our stack now contains the path from (1,1) to (8,1). To try another path, we back up to (7,1) by deleting this position from the stack. As there are no unblocked positions adjacent to (7,1), we back up to (6,1) by deleting this position from the stack. In this way we back up to position (3,1) from which we are again able to move forward (i.e., move to (3,2)). Notice that the stack always contains the path from the entrance to the current position. If we reach the exit, the entrance-to-exit path will be on the stack.

To refine Figure 8.14, we need representations for the maze, which is a matrix of zeros and ones, each maze position, and the stack. Let us consider the maze first. The maze is naturally represented as a two-dimensional array `maze` of type `int`. (Since each array position can take on only one of the values 0 and 1, we could use the data type `bool` and represent the value 1 by `true` and the value 0 by `false`. This approach would reduce the space required for the array `maze`.) Position (i, j) of the maze matrix corresponds to position $[i][j]$ of the array `maze`.

From interior (i.e., nonboundary) positions of the maze, four moves are possible: right, down, left, and up. From positions on the boundary of the maze, either two or three moves are possible. To avoid having to handle positions on the boundaries of the maze differently from interior positions, we will surround the maze with a wall of obstacles. For an $m \times m$ maze, this wall will occupy rows 0 and $m + 1$ and columns 0 and $m + 1$ of the array `maze` (see Figure 8.15).

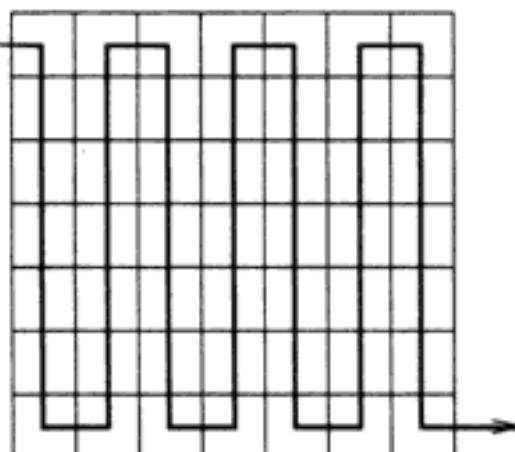
All positions in the maze are now within the boundary of the surrounding wall, and we can move to four possible positions from each position (some of these four positions may have obstacles). By surrounding the maze with our own boundary, we have eliminated the need for our program to handle boundary conditions, which significantly simplifies the code. This simplification is achieved at the cost of a

1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	0	0	0	0	0	1	
1	0	0	0	0	0	1	0	1	0	0	0	1	
1	0	0	0	1	0	1	0	0	0	0	0	1	
1	0	1	0	1	0	1	0	1	1	0	1		
1	0	1	0	1	0	1	0	1	0	0	1		
1	0	1	1	1	0	1	0	1	0	1	1		
1	0	1	0	0	0	1	0	1	0	1	1		
1	0	1	0	1	1	1	0	1	0	0	1		
1	1	0	0	0	0	0	0	1	0	0	1		
1	0	0	0	0	1	1	1	1	0	0	1		
1	1	1	1	1	1	1	1	1	1	1	1	1	

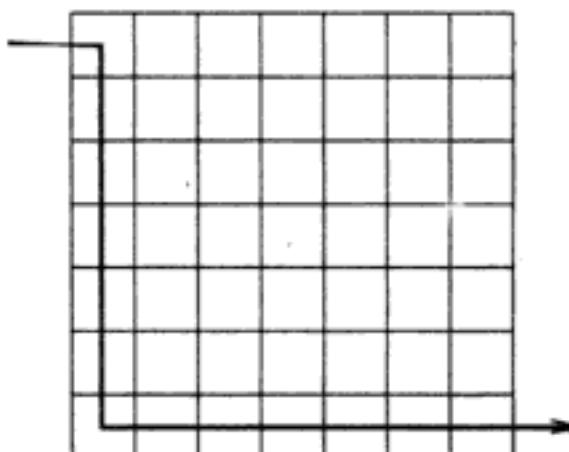
Figure 8.15 Maze of Figure 8.9 with wall of 1s around it

slightly increased space requirement for the array `maze`.

Each maze position is described by its row and column index, which are, respectively, called the row and column coordinates of the position. We may define a class `position` with data members `row` and `col` and use objects of type `position` to keep track of maze positions. The stack, `path`, that maintains the path from the entrance to the current position may be represented as an array stack. An $m \times m$ -maze with no blockages can have paths with as many as m^2 positions (see Figure 8.16(a)).



(a) A long path



(b) A short path

Figure 8.16 Paths in a maze with no blockages

Since no path repeats a position and the maze has only m^2 positions, no path can have more than m^2 positions. Further, as the last position on a path is not stored on the stack, at most $m^2 - 1$ positions can get stacked. Notice that a maze with no blockages always has a path with at most $2m$ positions between any two points (see, for example, Figure 8.16(b)). However, we have no assurance at this time that our path finder will find the shortest path.

We can now refine Figure 8.14 and obtain Figure 8.17, which is closer to being a C++ program.

Now we need to tackle the problem of determining a neighbor of position `here` that can be moved to. The task of trying out alternative moves is simplified if we select from the options available at any position in some systematic way. For example, we may first attempt to move right, then down, then left, and finally up. Once an option has been selected, we need to know the coordinates of the position to move to. These coordinates are easily computed by maintaining a table of offsets as in Figure 8.18. The moves right, down, left, and up have, respectively, been numbered 0, 1, 2, and 3. In the table of Figure 8.18, `offset[i].row` and `offset[i].col`, respectively, give the amounts to be added to the `row` and `col` coordinates of the present position to move to the adjacent position in direction `i`. For example, if we are at position (3, 4), then the position on the right has row coordinate `3+offset[0].row = 3` and column coordinate `4+offset[0].col = 5`.

To avoid moving to positions that we have been through before, we place an obstacle (i.e., set `maze[i][j] = 1`) at each position `maze[i][j]` that we move to.

Incorporating these refinements into the code of Figure 8.17 results in the C++ code of Program 8.15. In the code of Program 8.15, the variable `size` contains the number m of rows and columns in the maze.

The method `findPath` begins by creating an empty stack `path`. It then initializes the array of offsets and builds a wall of obstacles around the maze. In the `while` loop we attempt to advance the path forward from the current position `here` by trying the move options in the following order: right, down, left, and up. If we are able to move forward, the present location is stored on the stack `path` and a forward move is made. If a forward move isn't possible, we try to back up to a previous position. If there is no position to back up to (i.e., the stack is empty), there is no path to the exit. Otherwise, we can back up. Once we back up to the top position on the stack (`next`), we need to move forward by trying the next move option. This option can be computed from the positions `next` and `here`. Notice that `here` is a neighbor of `next`. In fact, at some previous time in the program, we moved from `next` to `here`, and this move was the last move made from `next`. The next move option to try is correctly computed by the following code:

```
if (next.row == here.row)
    option = 2 + next.col - here.col;
else option = 3 + next.row - here.row;
```

```
bool findPath()
{// Find a path from (1,1) to the exit (size, size).
// Return true if successful, false if impossible.

    path = new arrayStack<position>;

    // initialize offsets
    position offset[4];
    offset[0].row = 0; offset[0].col = 1;    // right
    offset[1].row = 1; offset[1].col = 0;    // down
    offset[2].row = 0; offset[2].col = -1;   // left
    offset[3].row = -1; offset[3].col = 0;   // up

    // initialize wall of obstacles around maze
    for (int i = 0; i <= size + 1; i++)
    {
        maze[0][i] = maze[size + 1][i] = 1; // bottom and top
        maze[i][0] = maze[i][size + 1] = 1; // left and right
    }

    position here;
    here.row = 1;
    here.col = 1;
    maze[1][1] = 1; // prevent return to entrance
    int option = 0; // next move
    int lastOption = 3;

    // search for a path
    while (here.row != size || here.col != size)
    {// not exit
        // find a neighbor to move to
        int r, c;
        while (option <= lastOption)
        {
            r = here.row + offset[option].row;
            c = here.col + offset[option].col;
            if (maze[r][c] == 0) break;
            option++; // next option
        }
    }
}
```

Program 8.15 Code to find a path in a maze (continues)

```
bool findPath()
{// Find a path from (1,1) to the exit (m,m).
 Initialize wall of obstacles around maze;

// initialize variable to keep track of
// our current position in the maze
here.row = 1;
here.col = 1;

maze[1][1] = 1; // prevent return to entrance

// search for a path to the exit
while (not at exit) do
{
    find a neighbor to move to;
    if (there is such a neighbor)
    {
        add position here to path stack;
        // move to and block neighbor
        here = neighbor;
        maze[here.row][here.col] = 1;
    }
    else
    {
        // cannot move forward, backup
        if (path empty) return false;
        back up to position here, which is at top of path stack;
    }
}
return true;
}
```

Figure 8.17 Refined version of Figure 8.14

For the time complexity analysis, we see that in the worst case we may move to each unblocked position of the input maze. Each such position may get added to the stack at most three times. (Each time we move forward from a position, it is added to the stack; at most three forward moves are possible from any position.) Hence each position may be removed from the stack at most three times. Further, at each position $\Theta(1)$ time is spent examining its neighbors. So the time complexity

Move	Direction	offset[move].row	offset[move].col
0	right	0	1
1	down	1	0
2	left	0	-1
3	up	-1	0

Figure 8.18 Table of offsets

```

// was a neighbor found?
if (option <= lastOption)
{ // move to maze[r][c]
    path->push(here);
    here.row = r;
    here.col = c;
    maze[r][c] = 1; // set to 1 to prevent revisit
    option = 0;
}
else
{ // no neighbor to move to, back up
    if (path->empty())
        return false; // no place to back up to
    position next = path->top();
    path->pop();
    if (next.row == here.row)
        option = 2 + next.col - here.col;
    else option = 3 + next.row - here.row;
    here = next;
}
}

return true; // at exit
}

```

Program 8.15 Code to find a path in a maze (concluded)

is $O(\text{unblocked})$ where *unblocked* is the number of unblocked positions in the input maze. This complexity is $O(\text{size}^2) = O(m^2)$.

When you get to Section 16.8.4 you'll see that the strategy used by `findPath` is really a depth-first search, which is just a special case of a more general strategy called backtracking. So `findPath` is really an application of depth-first search,

backtracking, and stacks.

EXERCISES

15. Write a program to determine whether or not a character string has an unmatched parenthesis. You should not use a stack. Test your code. What is the time complexity of your program?
16. Write a version of Program 8.6 that looks for matched pairs of parentheses and matched pairs of brackets ([and]). In the string $(a+[b*(c-d)+f])$, the matched pairs are (0,14), (3,13), and (6,10); and in the string $(a+[b*(c-d]+f))$, there is a nesting problem because the left parenthesis at 6 should be matched by a right parenthesis before a left bracket is encountered. Test your code.
17. Do Exercise 16 for the case when you have parentheses, brackets, and braces ({ and }).
18. Manually determine the sequence of disk moves for the four-disk Tower of Hanoi problem.
19. Establish the correctness of Program 8.7 by induction on the number of disks.
20. Assume that the Towers of Hanoi disks are labeled 1 through n with the smallest disk being disk 1. Modify Program 8.7 so that it also outputs the label of the disk that is being moved. This modification requires a simple change to the output statement. Do not make any other changes.
21. Write code for the `showState` method of Program 8.8 assuming that the output device is a computer screen. If necessary, add methods to `arrayStack` so that you can access the disks on a tower in a convenient manner. You will need to introduce a time delay so that the display does not change too rapidly. Show each disk in a different color.
22. The Towers of HaHa problem is like the Towers of Hanoi problem. However, the disks are numbered 1 through n ; odd-numbered disks are red, and even-numbered ones are yellow. The disks are initially on tower 1 in the order 1 through n from top to bottom. The disks are to be moved to tower 2, and at no time should a disk sit on top of a disk that has the same color. The initial and final disk order are the same.
 - (a) Write a program to move the disks from tower 1 to tower 2 using tower 3 for intermediate storage.
 - (b) How many disk moves does your program make?
23. Investigate the Towers of Hanoi problem under the assumption that you have $k > 1$ intermediate towers. The availability of more towers reduces the number

of moves needed. For example, when the number of intermediate towers is $n - 1$, a total of $2n - 1$ disk moves suffices. A good place to start is the case when two intermediate towers are available.

24. (a) You have a railroad shunting yard with three shunting tracks that operate as stacks. The initial ordering of cars is 3, 1, 6, 7, 2, 8, 5, 4. Draw figures similar to Figures 8.6 and 8.7 to show the configuration of the shunting tracks, the input track, and the output track following each car move made by the solution of Section 8.5.3.
(b) Do part (a) for two shunting tracks.
25. In our solution to the railroad car rearrangement problem (Section 8.5.3), we use k array stacks to represent k holding tracks. How large can each stack get? What is the total stack space required?
26. (a) Does Program 8.10 succeed in rearranging the cars whenever it is possible to do this rearrangement using k tracks?
(b) The total number of car moves required is $n + (\text{number of cars moved to a holding track})$. Suppose that the initial car arrangement can be rearranged using k tracks and Program 8.10. Does Program 8.10 perform the rearrangement using the minimum number of moves? Prove your answer.
27. Develop a program for the railroad car rearrangement problem under the assumption that holding track i can hold at most s_i cars, $1 \leq i \leq k$.
28. Walk through Program 8.13 for the case when the nets are (1, 6), (2, 5), (3, 4), (7, 10), (8, 9), (12, 13), and (11, 14). Show the stack configuration after the examination of each pin.
29. In the switch box routing application, we noted that processing can stop when two pins of the same net get on to the stack. Write a new version of `checkbox` that does this. The time complexity of your new method should be $O(n)$ where n is the number of pins. You may assume that the net numbers are 1 through $n/2$. How large a stack do you need?
30. Do the following for the offline equivalence class problem:
 - (a) Give the lists $list[1 : n]$ for the case when $n = 9$, $r = 9$, and the input relation pairs are (1, 3), (4, 2), (3, 8), (6, 7), (5, 8), (6, 2), (1, 5), (4, 7), and (9, 7).
 - (b) Walk through the second phase of the solution strategy using the lists of part (a). Provide an explanation of your progress as is done in Example 8.4.

31. Program 8.14 does not validate the relations as they are input. Modify this program so that it makes sure that each `a` and `b` that is input is in the range `[1, n]` and throws an exception of type `myInputException` whenever this is not the case.
32. (a) Modify Program 8.14 so that `list[]` is an array of type `arrayList` rather than `arrayStack`. Use a linear list iterator to examine the elements on a linear list in phase 2 of the program.
(b) Experimentally compare the performance of Program 8.14 and your new code.
33. Complete the rat-in-a-maze code. Write a pleasing C++ program by doing the following:
 - (a) Write a `welcome` function that incorporates graphics and audio.
 - (b) Write a robust `inputMaze` function that validates the data that is input. Also provide user prompts for input.
 - (c) Write the `outputPath` function to output the path from the maze to the exit (not from the exit to the entrance).

Test your codes using sample mazes.

34. Modify the code for the rat-in-a-maze problem so that the code works for mazes in which you are allowed to move to the north, northeast, east, southeast, south, southwest, west, and northwest neighbors of a position. Test the correctness of the modified code using suitable mazes.
35. Develop a better bound than $m^2 - 1$ for the maximum size of the stack path.
36. The strategy used to find a path in a maze is really a recursive one. From the present position we find a neighbor to move to and then determine whether there is a path from this neighbor to the exit. If so, we are done. If not, we find another neighbor to move to. Use recursion to find a path in a maze. Test the correctness of your code using suitable mazes.
37. Study the rat-in-a-maze animation that is on the Web site for this book.
 - (a) Identify heuristics you could program into the rat-in-a-maze program to select the next move in a more intelligent fashion than is done in Program 8.15. For example, should you preferentially follow along a wall of blocked positions looking for a break in the wall?
 - (b) Modify Program 8.15 to incorporate your heuristics.
 - (c) Test the correctness of the new code.
 - (d) Compare the run-time performance of the new code and that of Program 8.15.

38. You are given an array, `data[]`, of integers. Your task is to compute another integer array `lastAsBig[]`. Informally, `lastAsBig[i]` gives you the nearest position to the left where the data value is at least as big. For example, when `data[] = [6, 2, 3, 1, 7, 5]`, `lastAsBig[] = [-1, 0, 0, 2, -1, 4]`. More formally, `lastAsBig[i]` is the largest integer j such that $j < i$ and $\text{data}[j] \geq \text{data}[i]$. In case no there is no such j , then `lastAsBig[i] = -1`.

One application of `lastAsBig` is in weather reporting. Let `data[i]` be the high temperature recorded in Gainesville in day i of the current year. If `lastAsBig[i]` is -1 , then we have not seen a temperature this high earlier in the year. When `lastAsBig[i] \neq -1`, `lastAsBig[i]` gives the last time this year that the temperature was this high; $i - \text{lastAsBig}[i]$ gives the number of days since the temperature was this high (this year).

- (a) Give two more applications for `lastAsBig`.
- (b) Write a method to compute `lastAsBig` that uses a stack. The time complexity of your method should be $O(\text{data.length})$.
- (c) Test your method.

8.6 REFERENCES AND SELECTED READINGS

The switch box routing algorithm is from Hsu and Pinter. It is described in the papers “General River Routing Algorithm” by C. Hsu, *ACM/IEEE Design Automation Conference*, pages 578–583, 1983 and “River-Routing: Methodology and Analysis” by R. Pinter, *Third Caltech Conference on VLSI*, March 1983.

CHAPTER 9

QUEUES

BIRD'S-EYE VIEW

A queue, like a stack, is a special kind of linear list. In a queue insertions and deletions take place from different ends of the linear list. Consequently, a queue is a first-in-first-out (FIFO) list. Another variety of queue—a priority queue—from which deletions are made in order of element priority is developed in Chapter 12. The C++ STL class `queue` is an array implementation of the queue data structure. This class derives from the STL class `deque`, which is an array implementation of the double-ended queue data structure (Exercise 9).

Although queue classes may be derived easily from any of the linear list classes developed in Chapters 5 and 6, we do not do so in this chapter. For run-time efficiency reasons, the array and linked classes for a queue are developed from scratch.

In the applications section we develop four sample codes that use a queue. The first is for the railroad-switching problem considered initially in Section 8.5.3. In this chapter the problem has been modified so that the shunting tracks at the railroad yard are FIFO rather than LIFO. The second application is the classical Lee's algorithm to find the shortest path for a wire that is to connect two given points. This application may also be viewed as a variant of the rat-in-a-maze problem of Section 8.5.6. In this variant we must find the shortest path between the maze entrance and exit. Notice that the code developed in Section 8.5.6 does not guarantee to find a shortest path. That code simply guarantees to find a path

(of unspecified length) whenever the maze has at least one entrance-to-exit path. The third application, from the computer-vision field, labels the pixels of a binary image so that two pixels have the same label iff they are part of the same image component. The final application is a machine shop simulation. The machine shop has several machines, each capable of performing a different task. Each job in the shop requires one or more tasks to be performed. We provide a program to simulate the flow of jobs through the machine shop. Our program determines the total time each job spends waiting to be processed as well as the total wait at each machine. We can use this information to improve the machine shop. Although the machine shop simulator developed in this chapter uses FIFO queues, real-world machine shops may require some or all of these FIFO queues be replaced by priority queues. Additional queue applications appear in later chapters.

9.1 DEFINITION AND APPLICATIONS

Definition 9.1 A queue is a linear list in which insertions (also called additions and puts) and deletions (also called removals) take place at different ends. The end at which new elements are added is called the back or rear, and that from which old elements are deleted is called the front. ■

A queue with three elements is shown in Figure 9.1(a). The first element we delete from the queue of Figure 9.1(a) is A. Following the deletion, the configuration of Figure 9.1(b) results. To add element D to the queue of Figure 9.1(b), we must place it after element C. The new configuration is shown in Figure 9.1(c).

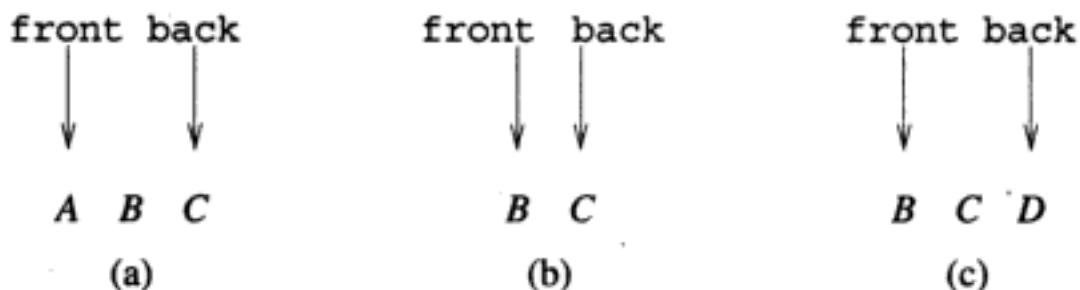


Figure 9.1 Sample queues

So a queue is a FIFO list, whereas a stack is a LIFO list.

Example 9.1 [Queues in the Real World]

- Although the stack of trays in a cafeteria works in a LIFO manner (see Example 8.1), the food line you stand in works in a FIFO manner. Customers exit at the checkout register in the order in which they entered the food line—the food line is a queue. Most other lines you find yourself in—the check-out line at a store, the line in front of a bank teller, the line of cars waiting at a car wash, the line at the postal center—also work in a FIFO manner.
- In a soda vending machine, you have a column of soda cans for each variety of soda dispensed. In each column cans are piled one on top of another. When you buy a can of soda, the can at the bottom of a column is given; when the stock is replenished, cans are added to the top of the column. The vending machine dispenses soda cans in a FIFO manner; the machine maintains a queue for each variety of soda dispensed.
- In a distributed system a single queue often feeds a bank of queues. A distributed system that has m servers has a bank of m server queues, one for each server. In addition, there is a queue called the broker or trader queue. Requests for service are first queued in the broker queue; a broker (or trader

or dispatcher) examines the service requests in the broker queue in a FIFO order and sends each request to the queue for the most appropriate server; servers handle the service requests in their queues in a FIFO order. Two specific examples follow.

1. In a distributed file system, computer files are replicated across file servers so as to provide a better level of service. All requests for a file are first queued in a broker queue; a broker dispatches each request to the least loaded server that has a copy of the requested file; the dispatched request waits its turn in the queue for the file server to which it is dispatched.
2. Voter polling stations provide a rather interesting application of queues. When you arrive at the polling station, you join a broker queue. When you get to the front of the broker queue, a polling volunteer directs you to a server queue based on the first letter of your surname. At the head of each server queue is a volunteer who checks your ID, gets your signature, and issues a ballot card. Once you have a card, you get into another queue and wait for a booth where you can punch holes in the ballot card to select your desired candidates. ■

EXERCISES

1. The following sequence of operations is done on an initially empty queue: add *X*, add *Y*, remove, add *D*, add *A*, remove, add *T*, add *A*. Draw figures similar to those of Figure 9.1 to show the queue configuration after each operation.
2. Identify three additional real-world applications of a queue. Do not include applications that involve people in a single line or vending machines. Distributed systems that involve people are okay.
3. Identify three real-world applications in which a stack is used sometimes and a queue is used at other times. For example, if you examine napkin dispensers, you will notice that some work as a stack and others work as a queue.
4. Which applications from Section 8.5 can use a queue instead of a stack without affecting the correctness of the program?

9.2 THE ABSTRACT DATA TYPE

The ADT queue is specified in ADT 9.1. The ADT function names are the same as those used in the C++ STL container class `queue`.

Program 9.1 gives the C++ abstract class that corresponds to ADT 9.1.

```

AbstractDataType queue
{
    instances
        ordered list of elements; one end is called the front; the other is the back;
    operations
        empty() : Return true if the queue is empty, return false otherwise;
        size() : Return the number of elements in the queue;
        front() : Return the front element of the queue;
        back() : Return the back element of the queue;
        pop() : Remove an element from the front of the queue;
        push(x) : Add element x at the back of the queue;
}

```

ADT 9.1 The abstract data type queue

```

template<class T>
class queue
{
    public:
        virtual ~queue() {}
        virtual bool empty() const = 0;
            // return true iff queue is empty
        virtual int size() const = 0;
            // return number of elements in queue
        virtual T& front() = 0;
            // return reference to the front element
        virtual T& back() = 0;
            // return reference to the back element
        virtual void pop() = 0;
            // remove the front element
        virtual void push(const T& theElement) = 0;
            // add theElement at the back of the queue
};

```

Program 9.1 The abstract class queue

9.3 ARRAY REPRESENTATION

9.3.1 The Representation

Suppose that the queue elements are mapped into an array queue using Equation 9.1.

$$\text{location}(i) = i \quad (9.1)$$

This equation worked well for the array representation of linear lists and stacks. Element i of the queue is stored in $\text{queue}[i]$, $i \geq 0$. Let `arrayLength` be the length or capacity of the array `queue` and let `queueFront` and `queueBack`, respectively, be the locations of the front and back elements of the queue. When Equation 9.1 is used, `queueFront` equals 0 and the queue size is `queueBack+1`. An empty queue has `queueBack = -1`. Using Equation 9.1, the queues of Figure 9.1 are represented as in Figure 9.2.

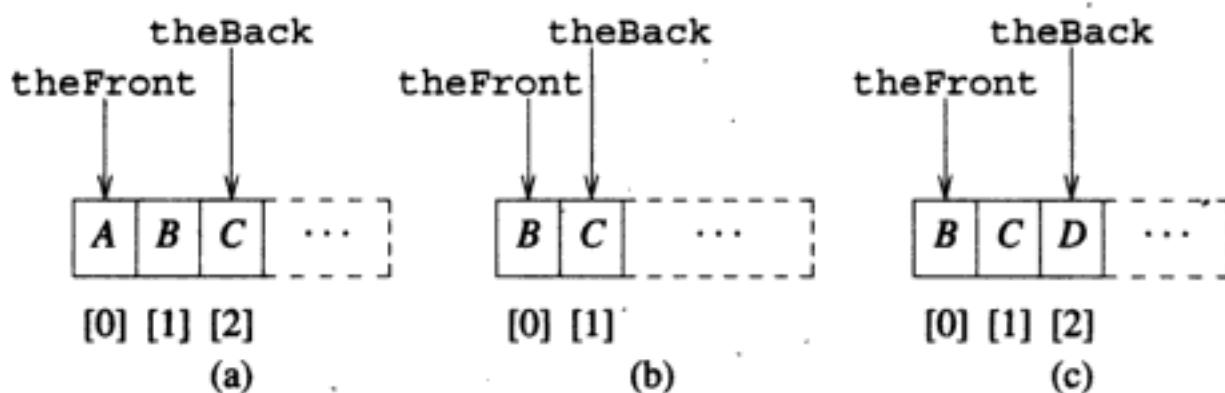


Figure 9.2 Queues of Figure 9.1 using Equation 9.1

To push an element into a queue, we need to increase `queueBack` by 1 and place the new element at `queue[queueBack]`, which means that a push operation requires $\Theta(1)$ time. To pop an element, we must slide the elements in positions 1 through `queueBack` one position down the array. Sliding the elements takes $\Theta(n)$ time where n is the number of elements in the queue following the pop.

We can pop an element in $\Theta(1)$ time if we use Equation 9.2 instead of Equation 9.1.

$$\text{location}(i) = \text{location}(\text{front element}) + i \quad (9.2)$$

Equation 9.2 does not require us to shift the queue one position left each time an element is popped from the queue. Instead, we simply increase $\text{location}(\text{front element})$ by 1. Figure 9.3 shows the representation of the queues of Figure 9.1 that results

when Equation 9.2 is used. Notice that `queueFront = location(front element)`, `queueBack = location(last element)`, and an empty queue has `queueBack < queueFront`.

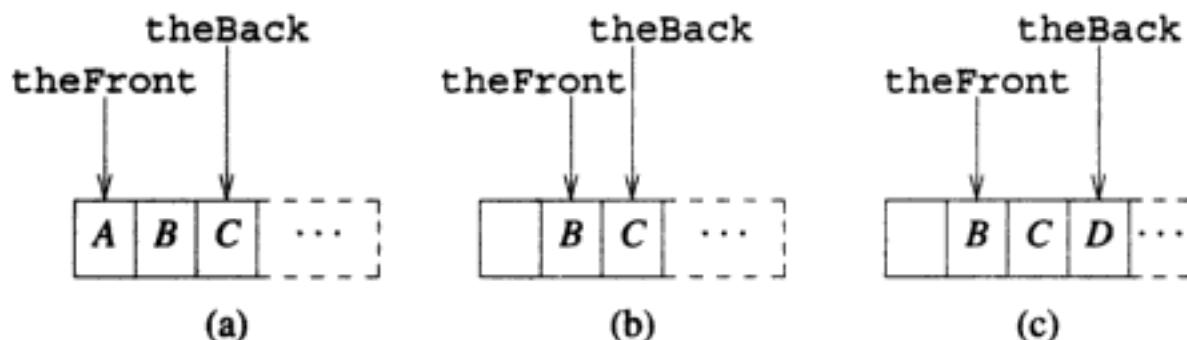


Figure 9.3 Queues of Figure 9.1 using Equation 9.2

As Figure 9.3(b) shows, each pop operation causes `queueFront` to move right by 1. Hence there will be times when `queueBack = arrayLength - 1` and `queueFront > 0`. At these times the number of elements in the queue is less than `arrayLength`, and there is space for additional elements at the left end of the array. To continue inserting elements into the queue, we can shift all elements to the left end of the queue (as in Figure 9.4) and create space at the right end. This shifting increases the worst-case time for a push operation from $\Theta(1)$, when Equation 9.1 is used, to $\Theta(\text{arrayLength})$. So the trade-off for improved efficiency of the pop operation is a loss of efficiency for the push operation.

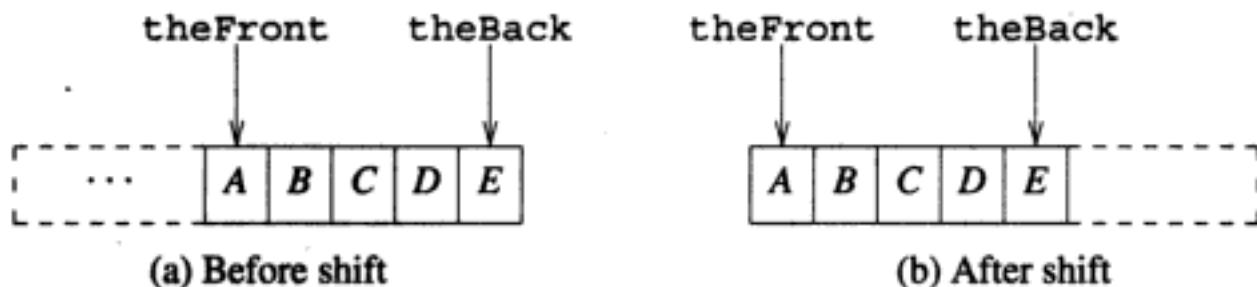
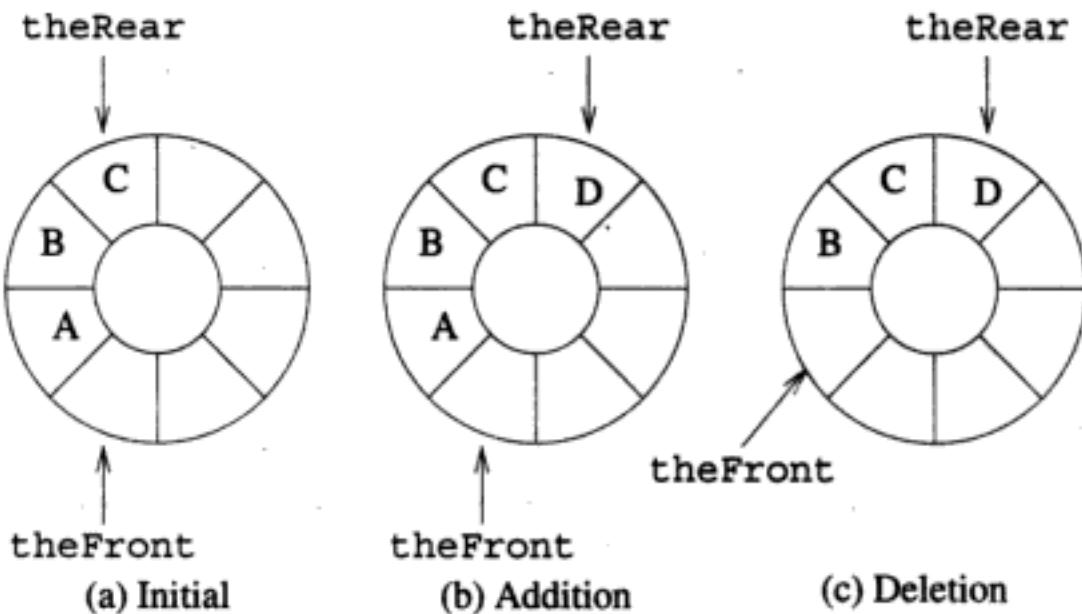


Figure 9.4 Shifting a queue

The worst-case push and pop times (assuming no array resizing is needed) become $\Theta(1)$ when we permit the queue to wrap around the end of the array. At this time it is convenient to think of the array positions as arranged in a circle (Figure 9.5) rather than in a straight line (Figure 9.4).

When the array is viewed as a circle, each array position has a next and a previous position. The position next to position `arrayLength - 1` is 0, and the

**Figure 9.5** Circular queues

position that precedes 0 is `arrayLength - 1`. When the back of the queue is at `arrayLength - 1`, the next element is put into position 0. The circular array representation of a queue uses the following mapping function:

$$\text{location}(i) = (\text{location(front element)} + i) \% \text{arrayLength} \quad (9.3)$$

In Figure 9.5 we have changed the convention for the variable `queueFront`. This variable now points one position counterclockwise from the location of the front element in the queue. The convention for `queueBack` is unchanged. This change simplifies the codes.

Pushing an element into the queue of Figure 9.5(a) results in the queue of Figure 9.5(b). Popping an element from the queue of Figure 9.5(b) results in the queue of Figure 9.5(c).

A queue is empty iff `queueFront = queueBack`. The initial condition `queueFront = queueBack = 0` defines an initially empty queue. If we push elements into the queue of Figure 9.5(b) until the number of elements in the array `queue` equals `arrayLength` (i.e., the queue becomes full), we obtain the configuration of Figure 9.6. This configuration has `queueFront = queueBack`, which is the same condition as when the queue is empty! Therefore, we cannot distinguish between an empty and a full queue. To avoid this difficulty, we will not permit a queue to get full. Before pushing an element into a queue, we verify whether this push will cause the queue to get full. If so, we double the length of the array `queue` and

then proceed with the push. Using this strategy, the array `queue` can have at most `arrayLength-1` elements in it.

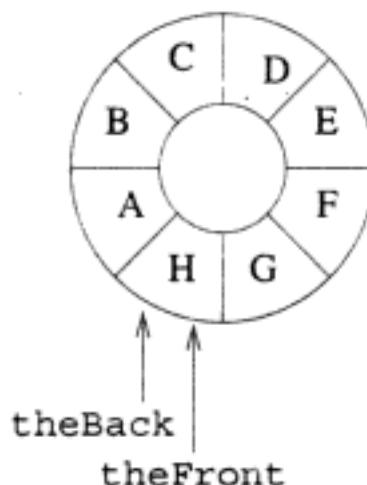


Figure 9.6 A circular queue with `arrayLength` elements

9.3.2 The Class `arrayQueue`

The class `arrayQueue` uses Equation 9.3 to map a queue into a one-dimensional array `queue`. The data members of `arrayQueue` are `queueFront`, `queueBack`, and `queue`; the codes for all methods other than `push` and `pop` are similar to those for the corresponding methods of `arrayStack`. These similar codes are available from the Web site. The method `push` (Program 9.2) uses customized code (Program 9.3) to double the array length.

To visualize array doubling when a circular queue is used, it is better to flatten out the array as in Figure 9.7(a). This figure shows a queue with seven elements in an array whose length is 8. Figure 9.7(b) shows the flattened view of the same queue. Figure 9.7(c) shows the array after array doubling by `changeLength1D` (Program 5.2).

To get a proper circular queue configuration, we must slide the elements in the right segment (i.e., elements *A* and *B*) to the right end of the array as in Figure 9.7(d). The array doubling copies `arrayLength` (this is the capacity of the array `queue` prior to array doubling) elements, and when the second segment is slid right, up to `arrayLength-2` additional elements are copied. The number of elements copied can be limited to `arrayLength-1` by customizing the array doubling code. Figure 9.7(e) shows an alternative configuration for the array after doubling. This configuration may be obtained as follows:

- Create a new array `newQueue` of twice the length.

```
template<class T>
void arrayQueue<T>::push(const T& theElement)
{// Add theElement to queue.

    // increase array length if necessary
    if ((queueBack + 1) % arrayLength == queueFront)
        {// double array length
            // code to double array size comes here
        }

    // put theElement at the queueBack of the queue
    queueBack = (queueBack + 1) % arrayLength;
    queue[queueBack] = theElement;
}
```

Program 9.2 Pushing an element into a queue

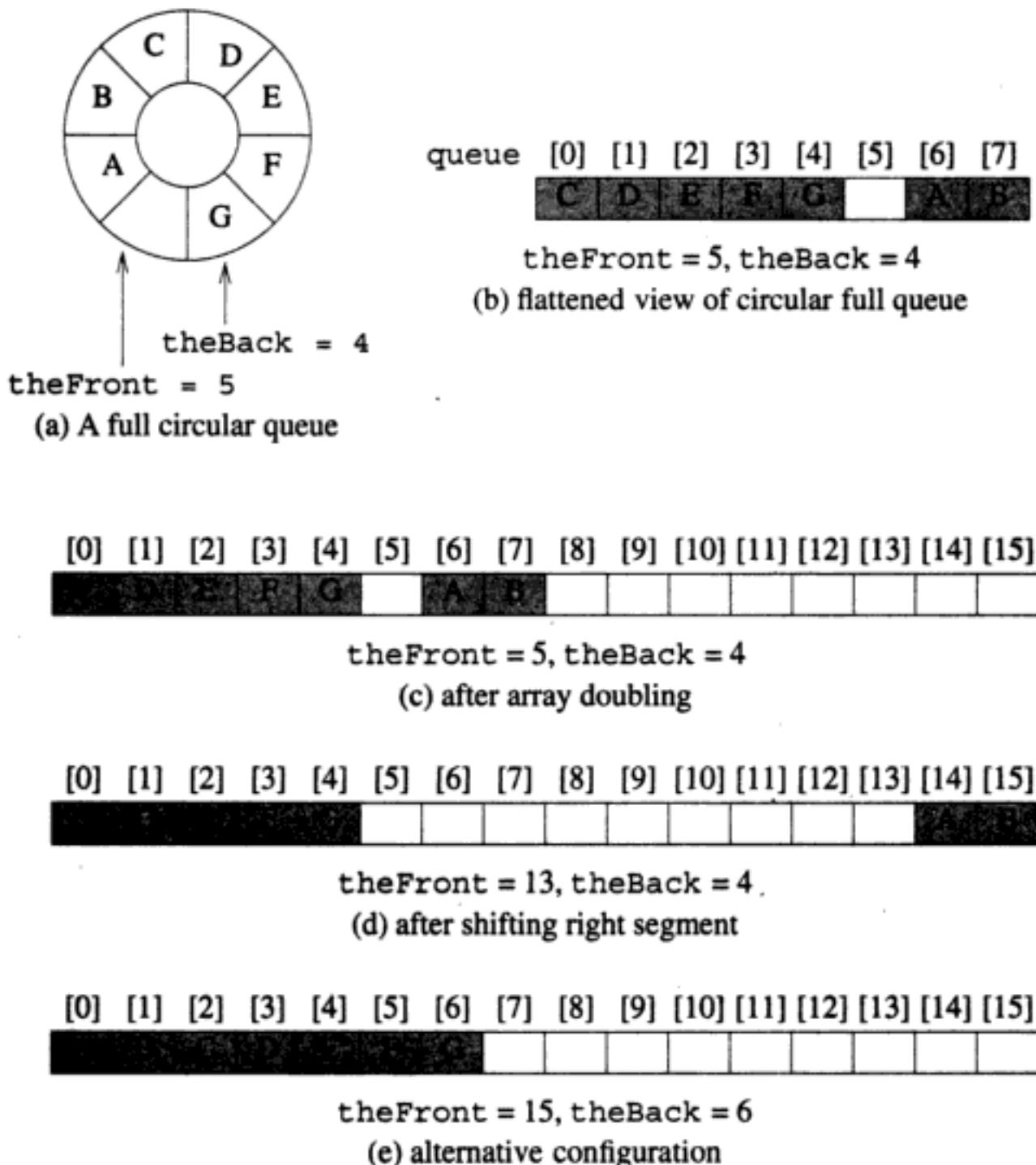
- Copy the second segment (i.e., the elements `queue[queueFront+1]` through `queue[arrayLength - 1]`) to positions in `newQueue` beginning at 0.
- Copy the first segment (i.e., the elements `queue[0]` through `queue[queueBack]`) to positions in `newQueue` beginning at `arrayLength-queueFront-1`.

The code of Program 9.3 obtains the configuration of Figure 9.7(e). The segment copying is done using the STL function `copy`. Program 9.4 gives the code for the `pop` method.

The complexity of the queue constructor is $O(1)$ when the `T` is a primitive datatype and is $O(\text{initialCapacity})$ when `T` is a user-defined type. The complexity of `empty`, `size`, `front`, `back`, and `pop` is $\Theta(1)$; and the complexity of `push` is $\Theta(1)$ when no array doubling is done and is $\Theta(\text{queue size})$ when array doubling is done. From the analysis used to establish Theorem 5.1, it follows that the complexity of m invocations of `push` is $O(m)$.

EXERCISES

5. (a) Extend the queue ADT by adding functions to
 - i. Input a queue.
 - ii. Output a queue.
 - iii. Split a queue into two queues. The first of the resulting queues contains the first, third, fifth, ... elements of the original queue; the second contains the remaining elements.

**Figure 9.7 Doubling array length**

- iv. Combine two queues by selecting elements alternately from the two queues beginning with queue 1. When a queue exhausts, append the remaining elements from the other queue to the combined queue. The relative order of elements from each queue is unchanged.

```
// allocate a new array
T* newQueue = new T[2 * arrayLength];

// copy elements into new array
int start = (theFront + 1) % arrayLength;
if (start < 2)
    // no wrap around
    copy(queue + start, queue + start + arrayLength - 1, newQueue);
else
{ // queue wraps around
    copy(queue + start, queue + arrayLength, newQueue);
    copy(queue, queue + theBack + 1, newQueue + arrayLength - start);
}

// switch to newQueue and set theFront and theBack
theFront = 2 * arrayLength - 1;
theBack = arrayLength - 2; // queue size arrayLength - 1
arrayLength *= 2;
queue = newQueue;
```

Program 9.3 Doubling the length of the array queue

```
void pop()
    { // remove queueFront element
        if (queueFront == queueBack)
            throw queueEmpty();
        queueFront = (queueFront + 1) % arrayLength;
        queue[queueFront].~T(); // destructor for T
    }
```

Program 9.4 Popping an element from a queue

- (b) Define the abstract class `extendedQueue` that derives from the abstract class `queue` and includes methods that correspond to the functions of (a).
 - (c) Develop code for the concrete class `extendedarrayQueue` that derives from the classes `arrayQueue` and `extendedQueue`.
 - (d) Test your code.
6. Develop the concrete class `slowArrayQueue` that derives from `queue` and use

the mapping of Equation 9.2 Test your code and compare its performance with that of `arrayQueue`.

7. Modify the representation used in the class `arrayQueue` so that a queue can hold as many elements as the length of the array `queue`. For this modification replace the variable `queueBack` with the variable `queueSize`, which equals the size of the queue. Use the convention that the front element is at `queue[queueFront]`. Note that the location of the back element may be computed from `queueSize` and `queueFront`. Test the correctness of your modified code.
8. Modify the representation used in the class `arrayQueue` so that a queue can hold as many elements as the length of the array `queue`. For this modification introduce another data member `lastOp` that keeps track of the last operation (from among the operations `push` and `pop`) performed on the queue. Notice that if the last operation performed was `push`, the queue cannot be empty. Also, if the last operation was `pop`, the queue cannot be full. So `lastOp` can be used to distinguish between an empty and full queue when `queueFront = queueBack`. Test the correctness of your modified code.
9. A `deque` (pronounced *deck*) is an ordered list to/from which we can make pushes and pops at/from either end. Therefore, we can call it a double-ended queue.
 - (a) Define the ADT `deque`. Include the operations `empty`, `size`, `front`, `back`, `push_front`, `push_back`, `pop_front`, and `pop_back`.
 - (b) Define an abstract C++ class `deque` that includes methods for each function of the ADT `deque`.
 - (c) Use Equation 9.3 to represent a deque. Develop a concrete C++ class `arrayDeque` that derives from `deque`. Note that the C++ STL has a concrete class `deque` that is an array implementation of the data structure `deque`.
 - (d) Test your code using suitable test data.
10. (a) Develop the concrete class `dequeStack` that derives from `stack` (Program 8.1) and `arrayDeque` (see Exercise 9).
 - (b) What is the time complexity of each method of `dequeStack`?
 - (c) Comment on the expected performance of the methods of `dequeStack` relative to their counterparts in `arrayStack`.
11. (a) Develop the concrete class `dequeQueue` that derives from `queue` (Program 9.1) and `arrayDeque` (see Exercise 9).
 - (b) What is the time complexity of each method of `dequeQueue`?
 - (c) Comment on the expected performance of the methods of `dequeQueue` relative to their counterparts in `arrayQueue`.

9.4 LINKED REPRESENTATION

A queue, like a stack, can be represented as a chain. We need two variables, `queueFront` and `queueBack`, to keep track of the two ends of a queue. There are two possibilities for binding these two variables to the two ends of a chain. The nodes can be linked from front to back (Figure 9.8(a)) or from back to front (Figure 9.8(b)). The relative difficulty of performing pushes and pops determines the direction of linkage. Figures 9.9 and 9.10, respectively, illustrate the mechanics of a push and a pop. We can see that both linkage directions are well suited for pushes, but the front-to-back linkage is more efficient for pops. Hence we will link the nodes in a queue from front to back.

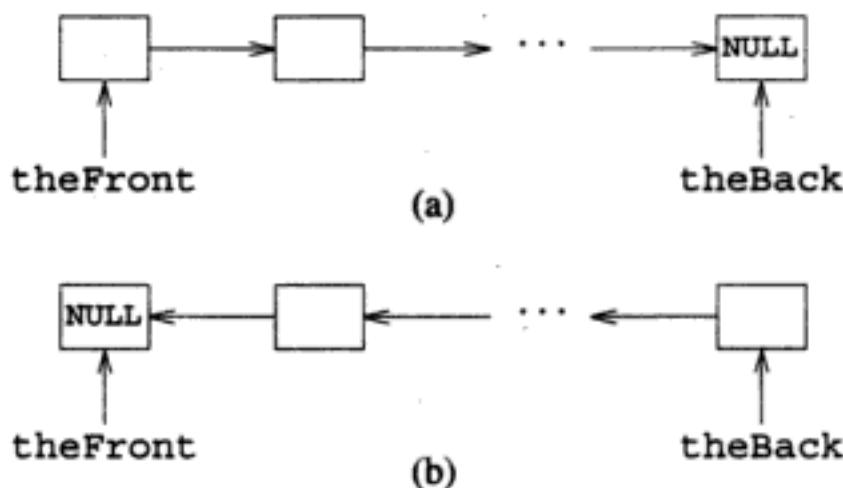


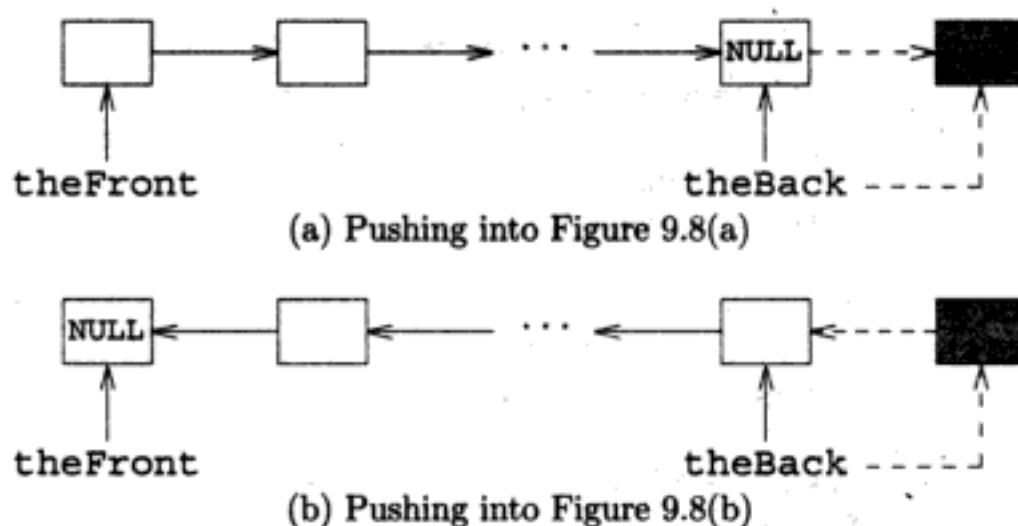
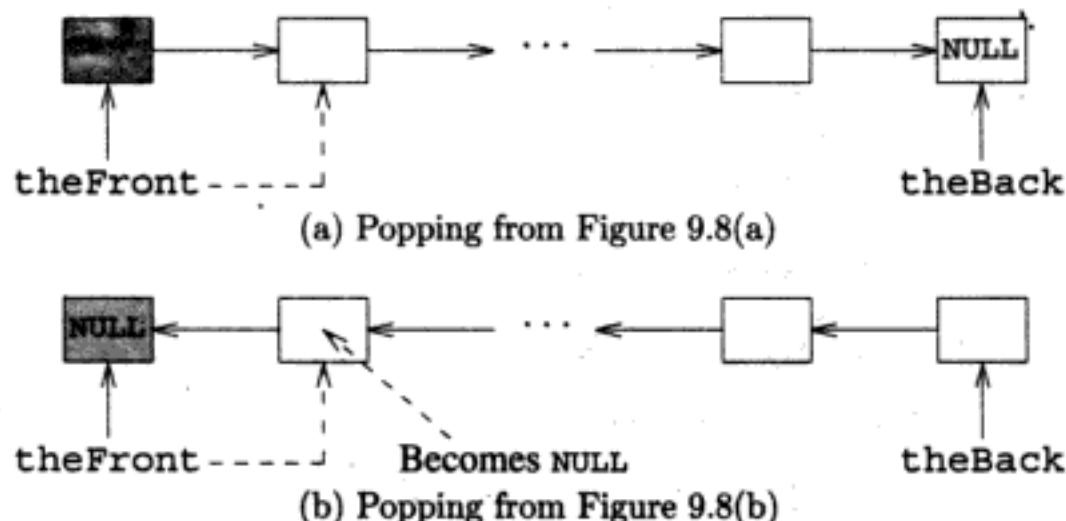
Figure 9.8 Linked queues

We can use the initial values `queueFront = queueBack = NULL` and the boundary value `queueFront = NULL` iff the queue is empty. The class `linkedQueue` may be defined as a derived class of `extendedChain` (Program 6.12). Exercise 12 considers this way of developing `linkedQueue`. In this section we develop the class `linkedQueue` from scratch.

Program 9.5 gives the push and pop methods of `linkedQueue`. You should run through these codes by hand using an empty queue, a queue with one element, and a queue with many elements as examples. The complexity of each of the linked queue methods is $\Theta(1)$.

EXERCISES

12. Develop the class `linkedQueueFromExtendedChain`, which implements a linked queue by deriving from `extendedChain` (Section 6.1.5) and `queue`.
13. Do Exercise 5 using a linked queue.

**Figure 9.9** Pushing an element into a linked queue**Figure 9.10** Popping an element from a linked queue

14. Compare the performance of `arrayQueue` and `linkedQueue` by performing a sequence of 1,000,000 push operations followed by 1,000,000 pop operations.
15. In some queue applications the elements to be put on a queue are already in nodes of type `chainNode`. For these applications it is desirable to have the methods `pushNode(chainNode theNode)`, which adds `theNode` at the back of the queue (notice that no call to `new` is made), and `popNode`, which removes and returns the front node of the queue.
 - (a) Write code for these methods.

```
template<class T>
void linkedQueue<T>::push(const T& theElement)
{// Add theElement to back of queue.

    // create node for new element
    chainNode<T>* newNode = new chainNode<T>(theElement, NULL);

    // add new node to back of queue
    if (queueSize == 0)
        queueFront = newNode;           // queue empty
    else
        queueBack->next = newNode;   // queue not empty
    queueBack = newNode;

    queueSize++;
}

template<class T>
void linkedQueue<T>::pop()
{// Delete front element.
    if (queueFront == NULL)
        throw queueEmpty();

    chainNode<T>* nextNode = queueFront->next;
    delete queueFront;
    queueFront = nextNode;
    queueSize--;
}
```

Program 9.5 The push and pop methods of `linkedQueue`

- (b) Test your code.
 - (c) Compare the time required by a sequence of 1,000,000 push operations followed by 1,000,000 pop operations with that required by 1,000,000 `pushNodes` followed by 1,000,000 `popNodes`.
16. See Exercise 9 for the definition of a deque.
- (a) Develop a concrete C++ class `doublyLinkedDeque` that derives from the abstract class `deque` of Exercise 9 and uses a doubly linked list. Your class should not derive from any other class.
 - (b) What is the complexity of each method of your class?

- (c) Test your code using suitable test data.
17. Do Exercise 16 using a singly linked list (i.e., a chain) rather than a doubly linked list. Name your class `linkedDeque`.
18. Do Exercise 16 using a singly linked circular list rather than a doubly linked list. Name your class `circularDeque`.

9.5 APPLICATIONS

9.5.1 Railroad Car Rearrangement

Problem Description and Solution Strategy

We will reconsider the railroad car rearrangement problem of Section 8.5.3. This time the holding tracks lie between the input and output track as in Figure 9.11. These tracks operate in a FIFO manner and so may be regarded as queues. As in the case of Section 8.5.3, moving a car from a holding track to the input track or from the output track to a holding track is forbidden. All car motion is in the direction indicated by the arrowheads of Figure 9.11.

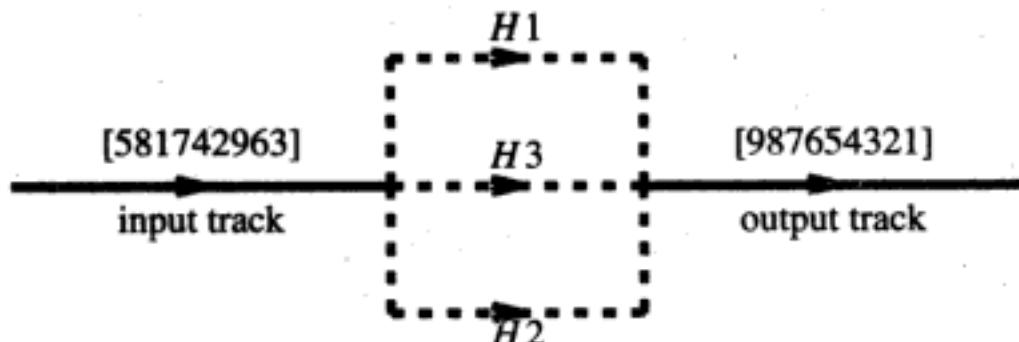


Figure 9.11 A three-track example

We reserve track H_k for moving cars directly from the input track to the output track. So only $k - 1$ tracks are available to hold cars that are not ready to be output.

Consider rearranging nine cars that have the initial ordering 5, 8, 1, 7, 4, 2, 9, 6, 3. Assume that $k = 3$ (Figure 9.11). Car 3 cannot be moved directly to the output track, as cars 1 and 2 must come before it. So car 3 is moved to H_1 . Car 6 can be placed behind car 3 in H_1 , as car 6 is to be output after car 3. Car 9 can now be placed after car 6 in H_1 . Car 2 cannot be placed after car 9, as car 2 is to be output before car 9. So it is placed at the front of H_2 . Car 4 can now be placed after car 2 in H_2 , and car 7 can be placed after it. Car 1 can be moved to the output using H_3 . Next car 2 is moved from H_2 to the output. Then car 3 is moved from H_1 to the output, and car 4 is moved from H_2 to the output. Car 5 is to be output next.

It is still in the input track. So car 8 is moved from the input track to H_2 . Then car 5 is moved to the output track. Now cars 6, 7, 8, and 9 are moved from their holding tracks to the output track.

When a car is to be moved to a holding track, we can use the following track selection method. *Move car c to a holding track that contains only cars with a smaller label; if several such tracks exist, select one with the largest label at its left end; otherwise, select an empty track (if one remains).*

First Implementation

We can implement the car rearrangement algorithm by using queues for the $k - 1$ holding tracks that can hold cars. We can model the code after Programs 8.9 through 8.12. Program 9.6 gives the new code for `outputFromHoldingTrack`, and Program 9.7 gives the new code for `putInHoldingTrack`. The method `railroad` of Program 8.10 needs to be modified. The changes are (1) decrease the number of tracks by 1 and (2) change the type of track to `arrayQueue`. The time needed to perform the rearrangement is $O(\text{numberOfCars} * k)$. We can use AVL trees (see Chapter 15) to reduce this time to $O(\text{numberOfCars} * \log k)$.

```
void outputFromHoldingTrack()
{// output the smallest car from the holding tracks
    // pop smallestCar from itsTrack
    track[itsTrack].pop();
    cout << "Move car " << smallestCar << " from holding track "
        << itsTrack << " to output track" << endl;
    *
    // find new smallestCar and itsTrack by checking all queue fronts
    smallestCar = numberOfCars + 2;
    for (int i = 1; i <= numberOfTracks; i++)
        if (!track[i].empty() && track[i].front() < smallestCar)
    {
        smallestCar = track[i].front();
        itsTrack = i;
    }
}
```

Program 9.6 Function to output a railroad car

Second Implementation

To animate the progress of the rearrangement algorithm, it is useful to keep the queues used in our first implementation. In an animation we wish to move all cars

```
bool putInHoldingTrack(int c)
{// Put car c into a holding track.
// Return false iff there is no feasible holding track for this car.

// find best holding track for car c
// initialize
int bestTrack = 0, // best track so far
    bestLast = 0; // last car in bestTrack

// scan tracks
for (int i = 1; i <= numberTracks; i++)
    if (!track[i].empty())
        {// track i not empty
            int lastCar = track[i].back();
            if (c > lastCar && lastCar > bestLast)
            {
                // track i has bigger car at its rear
                bestLast = lastCar;
                bestTrack = i;
            }
        }
    else // track i empty
        if (bestTrack == 0)
            bestTrack = i;

if (bestTrack == 0)
    return false; // no feasible track

// add c to bestTrack
track[bestTrack].push(c);
cout << "Move car " << c << " from input track "
    << "to holding track " << bestTrack << endl;

// update smallestCar and itsTrack if needed
if (c < smallestCar)
{
    smallestCar = c;
    itsTrack = bestTrack;
}

return true;
}
```

Program 9.7 Function to put a railroad car into a holding track

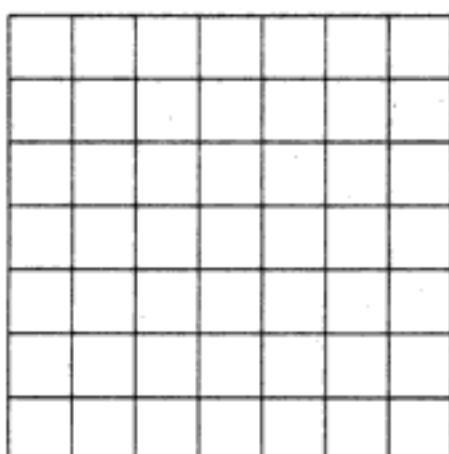
in a holding track one position right after the front car has been removed from the holding track. This is relatively easy to do when we have a list (in our case a queue) of the remaining cars in the holding track.

If our objective is simply to output the sequence of moves necessary to accomplish the rearrangement, then we need to know only the last member of each holding track (or queue) and the current track of each car. If holding track i is empty, let $\text{lastCar}[i]$ be 0; otherwise let it be the label/number of the last car in track i . If car i is in the input track, let $\text{whichTrack}[i]$ be 0; otherwise let it be the holding track car i is (was) in. Initially $\text{lastCar}[i] = 0$, $1 \leq i < k$, and $\text{whichTrack}[i] = 0$, $1 \leq i \leq n$. Using these variables and no queues, we can produce the same output as our first implementation produced. The code for the no-queue implementation is available from the Web site file `railroadWithNoQueues.cpp`.

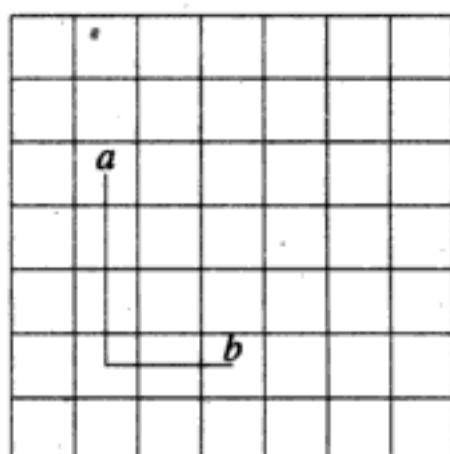
9.5.2 Wire Routing

Problem Description

As noted in Section 8.5.6, our solution to the rat-in-a-maze problem does not always find a shortest path from maze entrance to exit. The problem of finding a shortest path in a grid has many applications (besides the rat-in-a-maze problem). For example, a common approach to the wire-routing problem for electrical circuits is to impose a grid over the wire-routing region. The grid divides the routing region into an $n \times m$ array of squares much like a maze (Figure 9.12(a)). A wire runs from the midpoint of one square a to the midpoint of another b . In doing so, the wire may make right-angle turns (Figure 9.12(b)). Grid squares that already have a wire through them are blocked. To minimize signal delay, we wish to route the wire using a shortest path between a and b .



(a) A 7×7 grid

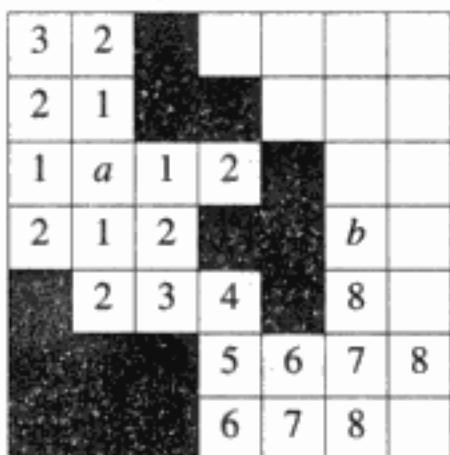


(b) A wire between a and b

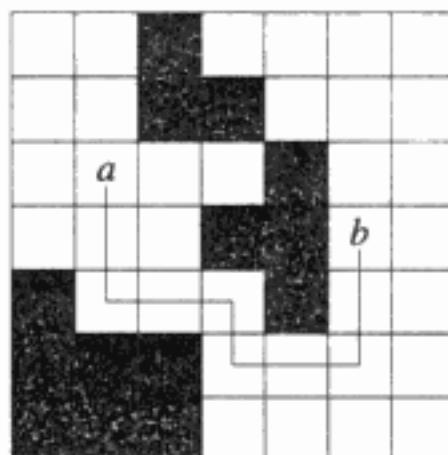
Figure 9.12 Wire-routing example

Solution Strategy

The following discussion assumes that you are familiar with the rat-in-a-maze development of Section 8.5.6. If not, you should review this development before proceeding. The shortest path between grid positions a and b is found in two passes—a distance-labeling pass and a path-identification pass. In the distance-labeling pass, we begin at position a and label its reachable neighbors 1 (i.e., they are distance 1 from a). Next the reachable neighbors of squares labeled 1 are labeled 2. This labeling process is continued until we either reach b or have no more reachable neighbors. Figure 9.13(a) shows the result of the distance-labeling pass for the case $a = (3,2)$ and $b = (4,6)$. The shaded squares are blocked squares.



(a) Distance labeling



(b) Wire path

Figure 9.13 Wire routing

Once we have reached b , we can label it with its distance (9 in the case of Figure 9.13(a)). The distance-labeling pass is followed by the path-identification pass in which we begin at b and move to any one its neighbors labeled 1 less than b 's label. Such a neighbor must exist as each grid's label is 1 more than that of at least one of its neighbors. In the case of Figure 9.13(a), we move from b to (5, 6). From here we move to one of its neighbors whose label is 1 less and so on until we reach a . In the example of Figure 9.13(a), from (5, 6) we move to (6, 6) and then to (6, 5), (6, 4), (5, 4), and so on. Figure 9.13(b) shows the constructed path, which is a shortest path between (3, 2) and (4, 6). Notice that the shortest path between (3, 2) and (4, 6) is not unique; (3, 2), (3, 3), (4, 3), (5, 3), (5, 4), (6, 4), (6, 5), (6, 6), (5, 6), (4, 6) is another shortest path.

C++ Implementation

Now let us take the strategy outlined above and obtain C++ code to find a shortest path in a grid. We will use many ideas from the rat-in-a-maze solution of Section 8.5.6. An $m \times m$ grid is represented as a two-dimensional array `grid` with a 0 representing an open position and a 1 a blocked position. The grid is surrounded by a “wall” of 1s; the array `offsets` helps us move from a position to its neighbors; and a queue keeps track of labeled grid positions whose neighbors have not been examined.

To implement the distance-labeling pass, we can either use an additional two-dimensional array for the distances or we can overload the use of the array `grid`. In practice, wire-routing grids do get large enough to tax the memory of even the most memory-rich computers. Therefore, the use of a second array is not recommended. When we overload the use of the array `grid`, we have a conflict between our use of the label 1 to designate a blocked grid position and our use of the label 1 for a position that is a unit distance from the start position `a`. To resolve this conflict, we increase all distance labels by 2. So `grid[i][j] = 1` for a blocked position; `grid[i][j] > 1` for a position whose distance from the start position is `grid[i][j] - 2`; and `grid[i][j] = 0` for an unblocked and unreached position.

Program 9.8 gives the code. `grid`, `size`, `pathLength`, `q`, `start`, `finish` and `path` are global variables.

Our code assumes that the positions `start` and `finish` are not blocked. The code begins with a check to see whether the `start` and `finish` positions are the same. In this case the path length is 0, and the code terminates. Otherwise, we set up a wall of blocked positions around the grid, initialize the offset array, and label the start position with a distance of 2. Using the queue `q` and beginning at position `start`, we move to reachable grid positions that are distance 1 from the start and then to those that are distance 2, and so on until we either reach the `finish` position or are unable to move to a new, unblocked position. In the latter case there is no path to the `finish` position. In the former case the `finish` position is labeled by its distance value.

If we reach the `finish` position, the path is reconstructed using the distance labels. The positions on the path (except for `start`) are stored in the array `path`.

Complexity

Since no grid position can get on the queue more than once, it takes $O(m^2)$ time (for an $m \times m$ grid) to complete the distance-labeling phase. The time needed for the path-construction phase is $O(\text{length of the shortest path})$.

```
bool findPath()
{// Find a shortest path from start to finish.
// Return true if successful, false if impossible.

    if ((start.row == finish.row) && (start.col == finish.col))
        // start == finish
        pathLength = 0;
        return true;
    }

    // initialize offsets
    position offset[4];
    offset[0].row = 0; offset[0].col = 1;    // right
    offset[1].row = 1; offset[1].col = 0;    // down
    offset[2].row = 0; offset[2].col = -1;   // left
    offset[3].row = -1; offset[3].col = 0;   // up

    // initialize wall of blocks around the grid
    for (int i = 0; i <= size + 1; i++)
    {
        grid[0][i] = grid[size + 1][i] = 1; // bottom and top
        grid[i][0] = grid[i][size + 1] = 1; // left and right
    }

    position here = start;
    grid[start.row][start.col] = 2; // block
    int numOfNbrs = 4; // neighbors of a grid position

    // label reachable grid positions
    arrayQueue<position> q;
    position nbr;
    do
    { // label neighbors of here
        for (int i = 0; i < numOfNbrs; i++)
            // check out neighbors of here
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if (grid[nbr.row][nbr.col] == 0)
                // unlabeled nbr, label it
                grid[nbr.row][nbr.col]
                    = grid[here.row][here.col] + 1;
```

Program 9.8 Find a wire route (continues)

```
        if ((nbr.row == finish.row) &&
            (nbr.col == finish.col)) break; // done
        // put on queue for later expansion
        q.push(nbr);
    }
}

// have we reached finish?
if ((nbr.row == finish.row) &&
    (nbr.col == finish.col)) break; // done

// finish not reached, can we move to a nbr?
if (q.empty())
    return false; // no path
here = q.front(); // get next position
q.pop();
} while(true);

// construct path
pathLength = grid[finish.row][finish.col] - 2;
path = new position [pathLength];

// trace backwards from finish
here = finish;
for (int j = pathLength - 1; j >= 0; j--)
{
    path[j] = here;
    // find predecessor position
    for (int i = 0; i < numNbrs; i++)
    {
        nbr.row = here.row + offset[i].row;
        nbr.col = here.col + offset[i].col;
        if (grid[nbr.row][nbr.col] == j + 2) break;
    }
    here = nbr; // move to predecessor
}

return true;
}
```

Program 9.8 Find a wire route (concluded)

9.5.3 Image-Component Labeling

Problem Description

A digitized image is an $m \times m$ matrix of pixels. In a binary image each pixel is either 0 or 1. A 0 pixel represents image background, while a 1 represents a point on an image component. We will refer to pixels whose value is 1 as component pixels. Two pixels are adjacent if one is to the left, above, right, or below the other. Two component pixels that are adjacent are pixels of the same image component. The objective of component labeling is to label the component pixels so that two pixels get the same label iff they are pixels of the same image component.

Consider Figure 9.14(a) that shows a 7×7 image. The blank squares represent background pixels, and the 1s represent component pixels. Pixels (1, 3) and (2, 3) are pixels of the same component because they are adjacent. Since component pixels (2, 3) and (2, 4) are adjacent, they are also of the same component. Hence the three pixels (1, 3), (2, 3), and (2, 4) are from the same component. Since no other image pixels are adjacent to these three pixels, these three define an image component. The image of Figure 9.14(a) has four components. The first component is defined by the pixel set (1, 3), (2, 3), (2, 4); the second is (3, 5), (4, 4), (4, 5), (5, 5); the third is (5, 2), (6, 1), (6, 2), (6, 3), (7, 1), (7, 2), (7, 3); and the fourth is (5, 7), (6, 7), (7, 6), (7, 7). In Figure 9.14(b) the component pixels have been given labels so that two pixels have the same label iff they are part of the same component. We use the numbers 2, 3, 4, ... as component identifiers; there is no component numbered 1 because 1 designates an unlabeled component pixel.

		1				
		1	1			
			1			
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(a) A 7×7 image

		2				
		2	2			
				3		
				3	3	
				4		3
4	4	4				5
4	4	4			5	5

(b) Labeled components

Figure 9.14 Image-component labeling

Solution Strategy

The components are determined by scanning the pixels by rows and within rows by columns. When an unlabeled component pixel is encountered, it is given a component identifier/label. This pixel forms the seed of a new component. We determine the remaining pixels in the component by identifying and labeling all component pixels that are adjacent to the seed. Call the pixels that are adjacent to the seed the distance 1 pixels. Then unlabeled component pixels that are adjacent to the distance 1 pixels are identified and labeled. These newly labeled pixels are the distance 2 pixels. Then unlabeled component pixels adjacent to the distance 2 pixels are identified and labeled. This process continues until no new unlabeled adjacent component pixels are found.

C++ Implementation

Our program to label component pixels uses much of the development used for the wire-routing problem. To move around the image with ease, we surround the image with a wall of blank (i.e., 0) pixels. We use the `offset` array to determine the pixels adjacent to a given pixel.

The labeling process used in the component-labeling problem is very similar to the process used to label squares in a wiring grid by their distance from the start square. This similarity results in a component-labeling code (Program 9.9) that is similar to Program 9.8.

Program 9.9 begins by setting up a wall of background (0) pixels around the image and initializing the array of neighbor/adjacent pixel offsets. The next two `for` loops scan the image for a seed for the next component. The seed is an unmarked component pixel. For such a pixel, `pixel[r][c]` is 1. The seed is assigned a component label by changing `pixel[r][c]` from 1 to a component identifier/label (`id`). Then with the help of a queue (a stack can be used instead), the remaining pixels in this component are identified. By the time the method `labelComponents` terminates, all component pixels have been assigned a label.

Complexity

It takes $\Theta(m)$ time to initialize the wall of background pixels and $\Theta(1)$ time to initialize `offsets`. Although the condition `pixel[r][c] == 1` is checked m^2 times, it is true only as many times as the number of components in the image. For each component $O(\text{number of pixels in component})$ time is spent identifying and labeling its pixels (other than the component seed). Since no pixel is in two or more components, the total time spent identifying and labeling nonseed component pixels is $O(\text{number of component pixels in image})$. Since the number of component pixels equals the number of pixels with value 1 in the input image and since this number is at most m^2 , the overall time complexity of `labelComponents` is $O(m^2)$.

```
void labelComponents()
{// Label the components.

    // initialize offsets
    position offset[4];
    offset[0].row = 0; offset[0].col = 1;    // right
    offset[1].row = 1; offset[1].col = 0;    // down
    offset[2].row = 0; offset[2].col = -1;   // left
    offset[3].row = -1; offset[3].col = 0;   // up

    // initialize wall of 0 pixels
    for (int i = 0; i <= size + 1; i++)
    {
        pixel[0][i] = pixel[size + 1][i] = 0; // bottom and top
        pixel[i][0] = pixel[i][size + 1] = 0; // left and right
    }

    int numOfNbrs = 4; // neighbors of a pixel position

    // scan all pixels labeling components
    arrayQueue<position> q;
    position here, nbr;
    int id = 1; // component id
    for (int r = 1; r <= size; r++)           // row r of image
        for (int c = 1; c <= size; c++)       // column c of image
            if (pixel[r][c] == 1)
                {// new component
                    pixel[r][c] = ++id; // get next id
                    here.row = r;
                    here.col = c;

                    while (true)
                        {// find rest of component
                            for (int i = 0; i < numOfNbrs; i++)
                                {// check all neighbors of here
                                    nbr.row = here.row + offset[i].row;
                                    nbr.col = here.col + offset[i].col;
                                    if (pixel[nbr.row][nbr.col] == 1)
                                        {// pixel is part of current component
                                            pixel[nbr.row][nbr.col] = id;
                                            q.push(nbr);
                                        }
                                }
                        }
                }
    }
}
```

Program 9.9 Component labeling (continues)

```
// any unexplored pixels in component?  
if (q.empty()) break;  
here = q.front(); // a component pixel  
q.pop();  
}  
  
} // end of if, for c, and for r  
}
```

Program 9.9 Component labeling (concluded)

9.5.4 Machine Shop Simulation

Problem Description

A machine shop (or factory or plant) comprises m machines or workstations. The machine shop works on jobs, and each job comprises several tasks. Each machine can process one task of one job at any time, and different machines perform different tasks. Once a machine begins to process a task, it continues processing that task until the task completes.

Example 9.2 A sheet metal plant might have one machine (or station) for each of the following tasks: design; cut the sheet metal to size; drill holes; cut holes; trim edges; shape the metal; and seal seams. Each of these machines/stations can work on one task at a time.

Each job includes several tasks. For example, to fabricate the heating and air-conditioning ducts for a new house, we would need to spend some time in the design phase, and then some time cutting the sheet metal stock to the right size pieces. We need to drill or cut the holes (depending on their size), shape the cut pieces into ducts, seal the seams, and trim any rough edges. ■

For each task of a job, there is a task time (i.e., how long does it take) and a machine on which it is to be performed. The tasks of a job are to be performed in a specified order. So a job goes first to the machine for its first task. When this first task is complete, the job goes to the machine for its second task, and so on until its last task completes. When a job arrives at a machine, the job may have to wait because the machine might be busy. In fact, several jobs may already be waiting for that machine.

Each machine in our machine shop can be in one of three states: active, idle, and change over. In the active state the machine is working on a task of some job; in the idle state it is doing nothing; and in the change-over state the machine has completed a task and is preparing for a new task. In the change-over state, the machine operator might, for example, clean the machine, put away tools used for the

last task, and take a break. The time each machine must spend in its change-over state depends on the machine.

When a machine becomes available for a new job, it will need to pick one of the waiting jobs to work on. We assume that each machine serves its waiting jobs in a FIFO manner, and so the waiting jobs at each machine form a (FIFO) queue. Other assumptions for the selection of the next job are possible. For example, the next job may be selected by priority. Each job has a priority, and when a machine becomes free, the waiting job with highest priority is selected.

The time at which a job's last task completes is called its **finish time**. The **length** of a job is the sum of its task times. If a job of length l arrives at the machine shop at time 0 and completes at time f , then it must have spent exactly $f - l$ amount of time waiting in machine queues. To keep customers happy, it is desirable to minimize the time a job spends waiting in machine queues. Machine shop performance can be improved if we know how much time jobs spend waiting and which machines are contributing most to this wait.

How the Simulation Works

When simulating a machine shop, we follow the jobs from machine to machine without physically performing the tasks. We simulate time by using a simulated clock that is advanced each time a task completes or a new job enters the machine shop. As tasks complete, new tasks are scheduled. Each time a task completes or a new job enters the shop, we say that an **event** has occurred. In addition, a **start event** initiates the simulation. When two or more events occur at the same time, we arbitrarily order these events. Figure 9.15 describes how a simulation works.

Example 9.3 Consider a machine shop that has $m = 3$ machines and $n = 4$ jobs. We assume that all four jobs are available at time 0 and that no new jobs become available during the simulation. The simulation will continue until all jobs have completed.

The three machines, M_1 , M_2 , and M_3 , have a change-over time of 2, 3, and 1, respectively. So when a task completes, machine 1 must wait two time units before starting another, machine 2 must wait three time units, and machine 3 must wait one time unit. Figure 9.16(a) gives the characteristics of the four jobs. Job 1, for example, has three tasks. Each task is specified as a pair of the form (machine, time). The first task of job 1 is to be done on M_1 and takes two time units, the second is to be done on M_2 and takes four time units, the third is to be done on M_1 and takes one time unit. The job lengths (i.e., the sum of their task times) are 7, 6, 8, and 4, respectively.

Figure 9.16(b) shows the machine shop simulation. Initially, the four jobs are placed into queues corresponding to their first tasks. The first task for jobs 1 and 3 are to be done on M_1 , so these jobs are placed on the queue for M_1 . The first tasks for jobs 2 and 4 are to be done on M_3 . Consequently, these jobs begin on the queue for M_3 . The queue for M_2 is empty. At the start all three machines are idle.

```

// initialize
input the data;
create the job queues at each machine;
schedule first job in each machine queue;

// do the simulation
while (an unfinished job remains)
{
    determine the next event;
    if (the next event is the completion of a machine change over)
        schedule the next job (if any) from this machine's queue;
    else
        { // a job task has completed
            put the machine that finished the job task into its change-over state;
            move the job whose task has finished to the machine for its next task
            (if any);
        }
}

```

Figure 9.15 The mechanics of simulation

We use the symbol I to indicate that the machines have no active job at this time. Since no machine is active, the time at which they will finish their current active task is undefined and denoted by the symbol L (large time).

The simulation begins at time 0. That is, the first event, the start event, occurs at time 0. At this time the first job in each machine queue is scheduled on the corresponding machine. Job 1's first task is scheduled on M_1 , and job 2's first task on M_3 . The queue for M_1 now contains job 3 only, while that for M_3 contains job 4 only. The queue for M_2 remains empty. Job 1 becomes the active job on M_1 , and job 2 the active job on M_3 . M_2 remains idle. The finish time for M_1 becomes 2 (current time of 0 plus task time of 2), and the finish time for M_3 becomes 4.

The next event occurs at time 2. This time is determined by finding the minimum of the machine finish times. At time 2 machine M_1 completes its active task. This task is a job 1 task. Job 1 is moved to machine M_2 for the next task. Since M_2 is idle, the processing of job 1's second task begins immediately. This task will complete at time 6 (current time of 2 plus task time of 4). M_1 goes into its change-over state and will remain in this state for two time units. Its active job is set to C (change over), and its finish time is set to 4.

At time 4 both M_1 and M_3 complete their active tasks. As machine M_1 completes a change-over task, that machine begins a new job; selecting the first job, job 3, from its queue. Since the task length for job 3's next task is 4, the task will complete at time 8 and the finish time for M_1 becomes 8. The next task for job 2,

Job#	#Tasks	Tasks	Length
1	3	(1,2) (2,4) (1,1)	7
2	2	(3,4) (1,2)	6
3	2	(1,4) (2,4)	8
4	2	(3,1) (2,3)	4

(a) Job characteristics

Time	Machine Queues			Active Jobs			Finish Times		
	M1	M2	M3	M1	M2	M3	M1	M2	M3
Init	1,3	—	2,4	I	I	I	L	L	L
0	3	—	4	1	I	2	2	L	4
2	3	—	4	C	1	2	4	6	4
4	2	—	4	3	1	C	8	6	5
5	2	—	—	3	1	4	8	6	6
6	2,1	4	—	3	C	C	8	9	7
7	2,1	4	—	3	C	I	8	9	L
8	2,1	4,3	—	C	C	I	10	9	L
9	2,1	3	—	C	4	I	10	12	L
10	1	3	—	2	4	I	12	12	L
12	1	3	—	C	C	I	14	15	L
14	—	3	—	1	C	I	15	15	L
15	—	—	—	C	3	I	17	19	L
16	—	—	—	C	3	I	17	19	L
17	—	—	—	I	3	I	L	19	L

(b) Simulation

Job#	Finish Time	Wait Time
1	15	8
2	12	6
3	19	11
4	12	8
Total	58	33

(c) Finish and wait times

Figure 9.16 Machine shop simulation example

which just completed its first task on machine M_3 , needs to be done on M_1 . Since M_1 is busy, job 2 is added to M_1 's job queue. M_3 moves into its change-over state and completes this change-over task at time 5. You should now be able to follow

the remaining sequence of events.

Figure 9.16(c) gives the finish and wait times. Since the length of job 2 is 6 and its finish time 12, job 2 must have spent a total of $12 - 6 = 6$ time units waiting in machine queues. Similarly, job 4 must have spent $12 - 4 = 8$ time units waiting in queues.

We may determine the distribution of the 33 units of total wait time across the three machines. For example, job 4 joined the queue for M_3 at time 0 and did not become active until time 5. So this job waited at M_3 for five time units. No other job experienced a wait at M_3 . The total wait time at M_3 was, therefore, five time units. Going through Figure 9.16(b), we can compute the wait times for M_1 and M_2 . The numbers are 18 and 10, respectively. As expected the sum of the job wait times (33) equals the sum of the machine wait times. ■

Benefits of Simulating a Machine Shop

Why do we want to simulate a machine shop? Here are some reasons:

- By simulating the shop, we can identify bottleneck machines/stations. If we determine that the paint station is going to be a bottleneck for the current mix of jobs, we can increase the number of paint stations in operation for the next few shifts. Similarly, if our simulation determines that the wait time at the drill station will be excessive in the next shift, we can schedule more drill station operators and put more drilling machines to work. Therefore, the simulator can be used for short-term operator-scheduling decisions.
- Using a machine shop simulator, we can answer questions such as, How is average wait time affected if we replace a certain machine with a more expensive but more effective machine? So the simulator can be used to help make expansion/modernization decisions at the factory.
- When customers arrive at the plant, they would like a fairly accurate estimate of when their jobs will complete. Such an estimate may be obtained by using a machine shop simulator.

High-Level Simulator Design

In designing our simulator, we will assume that all jobs are available initially (i.e., no jobs enter the shop during the simulation). Further, we assume that the simulation is to be run until all jobs complete.

The simulator is implemented as the class `machineShopSimulator`. Since the simulator is a fairly complex program, we break it into modules. The tasks to be performed by the simulator are input the data and put the jobs into the queues for their first tasks; perform the start event (i.e., do the initial loading of jobs onto the machines); run through all the events (i.e., perform the actual simulation); and

output the machine wait times. We will have one C++ function for each task. Program 9.10 gives the main function. The variable `largeTime` is a global variable that denotes a time that is larger than any permissible simulated time; that is all tasks of all jobs must complete before the time `largeTime`.

```
void main()
{
    inputData();          // get machine and job data
    startShop();          // initial machine loading
    simulate();           // run all jobs through shop
    outputStatistics();   // output machine wait times
}
```

Program 9.10 Main function for machine shop simulation

The Struct Task

Before we can develop the code for the four functions invoked by Program 9.10, we must develop representations for the data objects that are needed. These objects include tasks, jobs, machines, and an event list. We define a struct for the first three of these data object and a class for the third.

Each task has two components: `machine` (the machine on which it is to be performed) and `time` (the time needed to complete the task). Program 9.11 gives the struct `task`. Since machines are assumed to be have integer labels, `machine` is of type `int`. We will assume that all times are integral.

```
struct task
{
    int machine;
    int time;

    task(int theMachine = 0, int theTime = 0)
    {
        machine = theMachine;
        time = theTime;
    }
};
```

Program 9.11 The struct task

The Struct job

Each job has a list of associated tasks that are performed in list order. Consequently, the task list may be represented as a queue `taskQ`. To determine the total wait time experienced by a job, we need to know its length and finish time. The finish time is determined by the event clock, while the job length is the sum of task times. To determine a job's length, we associate a data member `length` with it. Program 9.12 gives the struct `job`.

```
struct job
{
    arrayQueue<task> taskQ;      // this job's tasks
    int length;                  // sum of scheduled task times
    int arrivalTime;             // arrival time at current queue
    int id;                      // job identifier

    job(int theId = 0)
    {
        id = theId;
        length = 0;
        arrivalTime = 0;
    }

    void addTask(int theMachine, int theTime)
    {
        task theTask(theMachine, theTime);
        taskQ.push(theTask);
    }

    int removeNextTask()
    // Remove next task of job and return its time.
    // Also update length.

        int theTime = taskQ.front().time;
        taskQ.pop();
        length += theTime;
        return theTime;
    }
};
```

Program 9.12 The struct job

The data member `arrivalTime` records the time at which a job enters its current

machine queue and determines the time the job waits in this queue. The job identifier is stored in `id` and is used only when outputting the total wait time encountered by this job.

The method `addTask` adds a task to the job's task queue. The task is to be performed on machine `theMachine` and takes `theTime` time. This method is used only during data input. The method `removeNextTask` is used when a job is moved from a machine queue to active status. At this time the job's first task is removed from the task queue (the task queue maintains a list of tasks yet to be scheduled on machines), the job length is incremented by the task time, and the task time is returned. The data member `length` becomes equal to the job length when we schedule the last task for the job.

The Struct `machine`

Each machine has a change-over time, an active job, and a queue of waiting jobs. Since each job can be in at most one machine queue at any time, the total space needed for all queues is bounded by the number of jobs. However, the distribution of jobs over the machine queues changes as the simulation proceeds. It is possible to have a few very long queues at one time. These queues might become very short later, and some other queues become long. By using linked queues, we limit the space required for the machine queues to that required for n nodes where n is the number of jobs.

Program 9.13 gives the struct `machine`. The data members `jobQ`, `changeTime`, `totalWait`, `numTasks`, and `activeJob`, respectively, denote the queue of waiting jobs, the change-over time for the machine, the total time jobs have spent waiting at this machine, the number of tasks processed by the machine, and the currently active job. The currently active job is `NULL` whenever the machine is idle or in its change-over state.

The Class `eventList`

We store the finish times of all machines in an event list. To go from one event to the next, we need to determine the minimum of these finish times. Our simulator also needs an operation that sets the finish time of a particular machine. This operation has to be done each time a new job is scheduled on a machine. When a machine becomes idle, its finish time is set to the large number `largeTime`. Program 9.14 gives the class `eventList` that implements the event list as a one-dimensional array `finishTime`, with `finishTime[p]` being the finish time of machine `p`.

The method `nextEventMachine` returns the machine that completes its active task first. The time at which machine `p` finishes its active task can be determined by invoking the method `nextEventTime(p)`. For a machine shop with m machines, it takes $\Theta(m)$ time to find the minimum of the finish times, so the complexity of `nextEventMachine` is $\Theta(m)$. The method to set the finish time of a machine, `setFinishTime`, runs in $\Theta(1)$ time. In Chapter 13 we will see two data

```
struct machine
{
    arrayQueue<job*> jobQ;
                    // queue of waiting jobs for this machine
    int changeTime; // machine change-over time
    int totalWait;  // total delay at this machine
    int numTasks;   // number of tasks processed on this machine
    job* activeJob; // job currently active on this machine

    machine()
    {
        totalWait = 0;
        numTasks = 0;
        activeJob = NULL;
    }
};
```

Program 9.13 The struct machine

structures—heaps and leftist trees—that may also represent an event list. When we use either of these data structures, the complexity of both `nextEventMachine` and `setFinishTime` becomes $O(\log m)$. If the total number of tasks across all jobs is `numTasks`, then, in a successful simulation run, our simulator will invoke `nextEventMachine` and `setFinishTime` $\Theta(\text{numTasks})$ times each. Using the event list implementation of Program 9.14, these invocations take a total of $\Theta(\text{numTasks} * m)$ time; using one of the data structures of Chapter 13, the invocations take $O(\text{numTasks} * \log m)$ time. Even though the data structures of Chapter 13 are more complex, they result in a faster simulation when the number of machines m is suitably large.

Global Variables

Program 9.15 gives the global variables used by our code. The significance of most of these variables is self-evident. `timeNow` is the simulated clock and records the current time. Each time an event occurs, it is updated to the event time. `largeTime` is a time that exceeds the finish time of the last job and denotes the finish time of an idle machine.

The Function `inputData`

The code for the function `inputData` (Program 9.16) begins by inputting the number of machines and jobs in the shop. Next we create the initial event list `eList`,

```

class eventList
{
    public:
        eventList(int theNumMachines, int theLargeTime)
        {// Initialize finish times for m machines.
            if (theNumMachines < 1)
                throw illegalParameterValue
                    ("number of machines must be >= 1");
            numMachines = theNumMachines;
            finishTime = new int [numMachines + 1];

            // all machines are idle, initialize with
            // large finish time
            for (int i = 1; i <= numMachines; i++)
                finishTime[i] = theLargeTime;
        }

        int nextEventMachine()
        {// Return machine for next event.

            // find first machine to finish, this is the
            // machine with smallest finish time
            int p = 1;
            int t = finishTime[1];
            for (int i = 2; i <= numMachines; i++)
                if (finishTime[i] < t)
                    {// i finishes earlier
                        p = i;
                        t = finishTime[i];
                    }
            return p;
        }

        int nextEventTime(int theMachine)
        {return finishTime[theMachine];}

        void setFinishTime(int theMachine, int theTime)
        {finishTime[theMachine] = theTime;}
    private:
        int* finishTime; // finish time array
        int numMachines; // number of machines
};


```

Program 9.14 The class eventList

```
// global variables
int timeNow;           // current time
int numMachines;        // number of machines
int numJobs;            // number of jobs
eventList* eList;       // pointer to event list
machine* mArray;        // array of machines
int largeTime = 10000;   // all machines finish before this
```

Program 9.15 Global variables for machine shop simulation

```
void inputData()
{// Input machine shop data.

    cout << "Enter number of machines and jobs" << endl;
    cin >> numMachines >> numJobs;
    if (numMachines < 1 || numJobs < 1)
        throw illegalInputData
            ("number of machines and jobs must be >= 1");

    // create event and machine queues
    eList = new eventList(numMachines, largeTime);
    mArray = new machine [numMachines + 1];

    // input the change-over times
    cout << "Enter change-over times for machines" << endl;
    int ct;
    for (int j = 1; j <= numMachines; j++)
    {
        cin >> ct;
        if (ct < 0)
            throw illegalInputData("change-over time must be >= 0");
        mArray[j].changeTime = ct;
    }
}
```

Program 9.16 Code to input machine shop data (continues)

with finish times equal to `largeTime` for each machine, and the array `mArray` of machines. Then we input the change-over times for the machines. Next we input the jobs one by one. For each job we first input the number of tasks it has, and then we input the tasks as pairs of the form (machine, time). The machine for the

```

// input the jobs
job* theJob;
int numTasks, firstMachine, theMachine, theTaskTime;
for (int i = 1; i <= numJobs; i++)
{
    cout << "Enter number of tasks for job " << i << endl;
    cin >> numTasks;
    firstMachine = 0; // machine for first task
    if (numTasks < 1)
        throw illegalInputData("each job must have > 1 task");

    // create the job
    theJob = new job(i);
    cout << "Enter the tasks (machine, time)"
        << " in process order" << endl;
    for (int j = 1; j <= numTasks; j++)
    { // get tasks for job i
        cin >> theMachine >> theTaskTime;
        if (theMachine < 1 || theMachine > numMachines
            || theTaskTime < 1)
            throw illegalInputData("bad machine number or task time");
        if (j == 1)
            firstMachine = theMachine; // job's first machine
        theJob->addTask(theMachine, theTaskTime); // add to
    } // task queue
    mArray[firstMachine].jobQ.push(theJob);
}
}

```

Program 9.16 Code to input machine shop data (concluded)

first task of the job is recorded in the variable `firstMachine`. When all tasks of a job have been input, the job is added to the queue for the first task's machine.

The Functions `startShop` and `changeState`

To start the simulation, we need to move the first job from each machine's job queue to the machine and commence processing. Since each machine is initialized in its idle state, we perform the initial loading in the same way as we change a machine from its idle state, which may happen during simulation, to an active state. The method `changeState(i)` performs this change over for machine `i`. The method to start the shop, Program 9.17, needs merely invoke `changeState` for each machine.

```
void startShop()
{// Load first jobs onto each machine.
    for (int p = 1; p <= numMachines; p++)
        changeState(p);
}
```

Program 9.17 Initial loading of machines

Program 9.18 gives the code for `changeState`. If machine `theMachine` is idle or in its change-over state, `changeState` returns `NULL`. Otherwise, it returns the job that `theMachine` has been working on. Additionally, `changeState(theMachine)` changes the state of machine `theMachine`. If machine `theMachine` was previously idle or in its change-over state, then it begins to process the next job on its queue. If that queue is empty, the machine's new state is "idle." If machine `theMachine` was previously processing a job, machine `theMachine` moves into its change-over state.

If `mArray[theMachine].activeJob` is `NULL`, then machine `theMachine` is either in its idle or change-over state; the job, `lastJob`, to return is `NULL`. If the job queue is empty, the machine moves into its idle state and its finish time is set to `largeTime`. If its job queue is not empty, the first job is removed from the queue and becomes machine `theMachine`'s active job. The time this job has spent waiting in machine `theMachine`'s queue is added to the total wait time for this machine, and the number of tasks processed by the machine incremented by 1. Next the task that this machine is going to work on is deleted from the job's task list, and the finish time of the machine is set to the time at which the new task will complete.

If `mArray[theMachine].activeJob` is not `NULL`, the machine has been working on a job whose task has just completed. Since this job is to be returned, we save it in `lastJob`. The machine should now move into its change-over state and remain in that state for `changeTime` time units.

The Functions `simulate` and `moveToNextMachine`

The function `simulate`, Program 9.19, cycles through all shop events until the last job completes. `numJobs` is the number of incomplete jobs, so the `while` loop of Program 9.19 terminates when no incomplete jobs remain. In each iteration of the `while` loop, the time for the next event is determined, and the clock time `timeNow` updated to this event time. A change-job operation is performed on the machine `nextToFinish` on which the event occurred. If this machine has just finished a task of a job (`theJob` is not `NULL`), job `theJob` moves to the machine on which its next task is to be performed. The function `moveToNextMachine` performs this move. If there is no next task for job `theJob`, the job has completed, function `moveToNextMachine` returns `false`, and `numJobs` is decremented by 1.

```

job* changeState(int theMachine)
{// Task on theMachine has finished, schedule next one.
 // Return last job run on this machine.
 job* lastJob;
 if (mArray[theMachine].activeJob == NULL)
 { // in idle or change-over state
     lastJob = NULL;
     // wait over, ready for new job
     if (mArray[theMachine].jobQ.empty()) // no waiting job
         eList->setFinishTime(theMachine, largeTime);
     else
     { // take job off the queue and work on it
         mArray[theMachine].activeJob =
             mArray[theMachine].jobQ.front();
         mArray[theMachine].jobQ.pop();
         mArray[theMachine].totalWait +=
             timeNow - mArray[theMachine].activeJob->arrivalTime;
         mArray[theMachine].numTasks++;
         int t = mArray[theMachine].activeJob->removeNextTask();
         eList->setFinishTime(theMachine, timeNow + t);
     }
 }
 else
 { // task has just finished on theMachine
 // schedule change-over time
     lastJob = mArray[theMachine].activeJob;
     mArray[theMachine].activeJob = NULL;
     eList->setFinishTime(theMachine, timeNow +
         mArray[theMachine].changeTime);
 }

 return lastJob;
}

```

Program 9.18 Code to change the active job at a machine

The function `moveToNextMachine` (Program 9.20) first checks to see whether any unprocessed tasks remain for the job `theJob`. If not, the job has completed and its finish time and wait time are output. The method returns `false` to indicate there was no next machine for this job.

When the job `theJob` to be moved has a next task, the machine `p` for this task

```
void simulate()
{// Process all jobs to completion.
    while (numJobs > 0)
        {// at least one job left
            int nextToFinish = eList->nextEventMachine();
            timeNow = eList->nextEventTime(nextToFinish);
            // change job on machine nextToFinish
            job* theJob = changeState(nextToFinish);
            // move theJob to its next machine
            // decrement numJobs if theJob has finished
            if (theJob != NULL && !moveToNextMachine(theJob))
                numJobs--;
        }
}
```

Program 9.19 Run all jobs through their machines

is determined and the job is added to this machine's queue of waiting jobs. In case machine p is idle, `changeState` is invoked to change the state of machine p so that machine p begins immediately to process the next task of `theJob`.

The Function `outputStatistics`

Since both the time at which a job finishes and the time a job spends waiting in machine queues are output by `moveToNextMachine`, `outputStatistics` needs to output only the time at which the machine shop completes all jobs (this time is also the time at which the last job completed and has been output by `moveToNextMachine`) and the statistics (total wait time and number of tasks processed) for each machine. Program 9.21 gives the code.

EXERCISES

19. Which applications from this section can use a stack instead of a queue without affecting the correctness of the program?
20. (a) You have a railroad shunting yard with three shunting tracks that operate as queues. The initial ordering of cars is 3, 1, 7, 6, 2, 8, 5, 4. Draw figures similar to Figures 9.11 to show the configuration of the shunting tracks, the input track, and the output track following each car move made by the solution of Section 9.5.1.
(b) Do part (a) for the case when the number of shunting tracks is 2.

```
bool moveToNextMachine(job* theJob)
{// Move theJob to machine for its next task.
// Return false iff no next task.

    if (theJob->taskQ.empty())
        {// no next task
            cout << "Job " << theJob->id << " has completed at "
                << timeNow << " Total wait was "
                << (timeNow - theJob->length) << endl;
            return false;
        }
    else
        {// theJob has a next task
            // get machine for next task
            int p = theJob->taskQ.front().machine;
            // put on machine p's wait queue
            mArray[p].jobQ.push(theJob);
            theJob->arrivalTime = timeNow;
            // if p idle, schedule immediately
            if (eList->nextEventTime(p) == largeTime)
                // machine is idle
                changeState(p);

            return true;
        }
}
```

Program 9.20 Move a job to the machine for the next task

21. Does Program 9.6 successfully rearrange all input car permutations that can be rearranged using k holding tracks that operate as queues? Prove your answer.
22. Rewrite Program 9.6 under the assumption that at most s_i cars can be in holding track i at any time. Reserve the track with smallest s_i for direct input to output moves.
23. Can you eliminate the use of queues and, instead, use the strategy of the second implementation of the railroad car problem when you have to display the state of the holding tracks following each move of a railroad car? Justify your answer.
24. Is it possible to solve the problem of Section 8.5.3 without the use of a stack

```
void outputStatistics()
{// Output wait times at machines.
    cout << "Finish time = " << timeNow << endl;
    for (int p = 1; p <= numMachines; p++)
    {
        cout << "Machine " << p << " completed "
            << mArray[p].numTasks << " tasks" << endl;
        cout << "The total wait time was "
            << mArray[p].totalWait << endl;
        cout << endl;
    }
}
```

Program 9.21 Output the wait times at each machine

(see second implementation of Section 9.5.1)? If so, develop and test such a program.

25. Consider the wire-routing grid of Figure 9.13(a). You are to route a wire between (1, 4) and (2, 2). Label all grid positions that are reached in the distance-labeling pass by their distance value. Now use the methodology of the path-identification pass to mark the shortest wire path.
26. Develop a complete C++ program for wire routing. Your program should include a `welcome` method that displays the program name and functionality; a method to input the wire grid size, blocked and unblocked grid positions, and wire endpoints; the method `findPath` (Program 9.8); and a method to output the input grid with the wire path shown. Test your code.
27. In a typical wire-routing application, several wires are routed in sequence. After a path has been found for one wire, the grid positions used by this path are blocked and we proceed to find a path for the next wire. When the array `grid` is overloaded to designate both blocked and unblocked positions as well as distances, we must clean the grid (i.e., set all grid positions that are on the wire path to 1 and all remaining positions with a label > 1 to 0) before we can begin on the next wire. Write a method to clean the grid. Do so by first restoring the grid to its initial state, using a process similar to that used in the distance-labeling pass. Then write code to block the positions on the wire path just found. This way the complexity of the cleanup pass is the same as that of the distance-labeling pass.
28. Develop a complete C++ program for image-component labeling. Your program should include a `welcome` function that displays the program name and

functionality; a function to input the image size and binary image; the function `labelComponents` (Program 9.9); and a function to output the image using a different color for pixels that are in different components. Test your code.

29. Rewrite function `labelComponents` using a stack. What are the relative merits/demerits of using a stack rather than a queue for this method?
30. Can we replace the stack in Program 8.6 with a queue? Why?
31. Can we replace the stack in Program 8.13 with a queue? Why?
32. Can we replace the stack in Program 8.14 with a queue? Why?
33. Can we replace the stack in Program 8.15 with a queue? Why?
34. Write an enhanced machine shop simulator that allows you to specify a minimum wait time between successive tasks of the same job. Your simulator must move a job into a wait state following the completion of each task (including the last one). Therefore, a job is placed on its next queue as soon as a task is complete. Upon arriving in this queue, the job enters its wait state. When a machine is ready to start a new task, it must bypass jobs at the front of the queue that are still in a wait state. The bypassed jobs could, for example, be moved to the end of the queue.
35. Write an enhanced machine shop simulator that allows jobs to enter the shop during simulation. The simulation stops at a specified time. Jobs that have not been completed by this time remain incomplete.

9.6 REFERENCES AND SELECTED READINGS

The wire-routing algorithm of Section 9.5.2 is known as Lee's router. The book *Algorithms for VLSI Physical Design Automation*, 2nd edition, by N. Sherwani, Kluwer Academic Publishers, Boston, 1995, contains a detailed discussion of this and other routing algorithms.

CHAPTER 10

SKIP LISTS AND HASHING

BIRD'S-EYE VIEW

Although a sorted array of n elements can be searched in $O(\log n)$ time with the binary search method, the search operation on a sorted chain takes $O(n)$ time. We can improve the expected performance of a sorted chain by placing additional pointers in some or all of the chain nodes. These pointers permit us to skip over several nodes of the chain during a search. Thus it is no longer necessary to examine all chain nodes from left to right during a search.

Chains augmented with additional forward pointers are called **skip lists**. Skip lists employ a randomization technique to determine which chain nodes are to be augmented by additional forward pointers and how many additional pointers are to be placed in the node. Using this randomization technique, skip lists deliver an expected performance of $O(\log n)$ for the search, insert, and delete operations. However, the worst-case performance is $\Theta(n)$.

Hashing is another randomization scheme that may be used to search, insert, and delete elements. It provides improved expected performance, $\Theta(1)$, over skip lists but has the same worst-case performance, $\Theta(n)$. Despite this performance, skip lists have an advantage over hashing in applications where we need to frequently output all elements in sorted order and/or search by element rank (e.g., find the 10th-smallest element). These latter two operations can be performed more efficiently with skip lists.

The asymptotic performance of sorted arrays, sorted chains, skip lists, and hash tables is summarized in the following table.

Method	Worst Case			Expected		
	Search	Insert	Erase	Search	Insert	Erase
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
sorted chain	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
skip lists	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
hash tables	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

The C++ STL has several container classes that employ hashing—`hash_map`, `hash_multimap`, `hash_multiset`, and `hash_set`.

One application of hashing is developed in this chapter. This application is text compression and decompression. The program we develop is based on the popular Lempel-Ziv-Welch algorithm.

10.1 DICTIONARIES

A **dictionary** is a collection of pairs of the form (k, v) , where k is a **key** and v is the **value** associated with the key k (equivalently, v is the value whose key is k). No two pairs in a dictionary have the same key.

The following operations are performed on a dictionary:

- Determine whether or not the dictionary is empty.
- Determine the dictionary size (i.e., number of pairs).
- Find the pair with a specified key.
- Insert a pair into the dictionary.
- Delete or erase the pair with a specified key.

Example 10.1 The class list for the data structures course is a dictionary with as many pairs as students registered for the course. When a new student registers, a pair/record corresponding to this student is inserted into the dictionary; when a student drops the course, his/her record may be deleted. During the course the instructor may query the dictionary to determine the record corresponding to a particular student and make changes to the record (for example, add/change test or assignment scores). The student name may be used as the key; the remaining information in a record is the value associated with the key. ■

A **dictionary with duplicates** is similar to a dictionary as defined above. However, it permits two or more $(key, value)$ pairs to have the same **key**.

Example 10.2 A word dictionary is a collection of pairs; each pair comprises a word and its value. The value of a word includes the meaning of the word, the pronunciation, etymologies, and so on. Webster's dictionary contains a pair (or entry) for the word *date*. Part of this pair reads "date, the point of time at which a transaction or event takes place." For this pair *date* is the key. Webster's dictionary actually has several pairs with the key *date*. Abbreviated forms of these pairs are "date, the oblong fruit of a palm" and "date, to assign a chronology record." The dictionary publisher inserts new pairs into the word dictionary as new words are created and as words take on new meanings; the publisher also deletes pairs that are no longer required. Users of a word dictionary, that is you and I, generally only search the dictionary for pairs with a given key. Occasionally, we might jot down a new entry into our copy of a dictionary. In data structures terminology a word dictionary is a dictionary with duplicates—it is a collection of pairs, each pair has a key, the keys need not be distinct, and the operations you perform on the pair collection are find, insert, and erase. Although the printed form of a word dictionary lists the pairs in alphabetical order of the keys, this arrangement is not

required by an electronic dictionary. In a computer you can store the pairs any way you like.

A telephone directory is another example of a dictionary with duplicates. ■

In a dictionary with duplicates, we need a rule to eliminate the ambiguity in the find and erase operations. That is, if we are to find (or erase) a pair with a specified key, then which of the several pairs with this key is to be returned (or erased)? Two possibilities for the find operation are (1) find any pair that has the specified key and (2) find all pairs that have the specified key. For the erase operation, we may require that the user be presented with all pairs that have the specified key; the user must select which pair(s) to erase. Alternatively, we may arbitrarily erase any one pair that has the specified key or erase all pairs with the specified key.

In the case of both dictionaries and dictionaries with duplicates, some applications require a different form of the erase operation in which all pairs inserted after a particular time are to be removed.

Example 10.3 A compiler uses a dictionary with duplicates, called the **symbol table**, of user-defined identifiers. When an identifier is defined, a pair (*key, value*) is created for it and inserted into the symbol table. The identifier is the *key* and information such as identifier type (*int, float*, etc.) and (relative) memory address for the value of the identifier comprise the *value* component of the pair. Since the same identifier name may be defined more than once (in different program blocks), the symbol table must be able to hold multiple pairs with the same key. A search should return the most recently inserted pair with the given key. Deletions are done only when the end of a program block is reached. All pairs inserted after the start of that block are to be deleted. ■

The find operation allows you to retrieve dictionary pairs **randomly**, by providing the key of the pair you want. Some dictionary applications require an additional access mode—**sequential access**—in which pairs are retrieved one by one in ascending order of keys. Sequential access requires an iterator that can sequence through the pairs in the dictionary in ascending order of their keys. All dictionary implementations developed in this chapter (other than hash tables) are suitable for both random and sequential access.

EXERCISES

1. Look in a word dictionary and find a word (other than *date*) for which there are multiple entries. Try to find a word that has more than three entries.
2. Give three real-world applications of dictionaries and/or dictionaries with duplicates. Do not repeat the ones given in this section. Explain which dictionary operations are used in each of your applications.
3. Give an application of a dictionary or dictionary with duplicates in which sequential access is desired.

10.2 THE ABSTRACT DATA TYPE

The abstract data type *Dictionary* is specified in ADT 10.1. In this specification, *p.first* and *p.second*, respectively, refer to the first and second components of the pair *p*. The first component of a pair is its key, and the second is the value. When the dictionary contains no pair with key *p.first*, *insert(p)* inserts the pair *p* into the dictionary; when the dictionary already contains a pair with key *p.first*, the old value associated with *p.first* is replaced by the new value *p.second*. This behavior of the *insert* operation agrees with the behavior of the *insert* operation of the STL class `hash_map`.

```
AbstractDataType Dictionary
{
    instances
        collection of pairs with distinct keys
    operations
        empty() : return true iff the dictionary is empty;
        size() : return the number of pairs in the dictionary;
        find(k) : return the pair with key k;
        insert(p) : insert the pair p into the dictionary;
        erase(k) : delete the pair with key k;
}
```

ADT 10.1 Dictionary abstract data type

Program 10.1 gives the C++ abstract class that corresponds to ADT 10.1. We require the *find* method to return a pointer to the matching pair rather than the pair itself. This behavior conforms to that of the *find* method of the STL container class `hash_map`.

In this chapter we do not explicitly develop representations for dictionaries with duplicates. However, the representations developed for dictionaries without duplicates may be adapted to the case when duplicate entries are permitted. The STL class `hash_multimap` represents a dictionary with duplicates.

EXERCISE

4. List the methods included in the C++ STL class `hash_map` that are not included in our abstract class `dictionary`? What does each new method do?

```
template<class K, class E>
class dictionary
{
public:
    virtual ~dictionary() {}
    virtual bool empty() const = 0;
        // return true iff dictionary is empty
    virtual int size() const = 0;
        // return number of pairs in dictionary
    virtual pair<const K, E>* find(const K&) const = 0;
        // return pointer to matching pair
    virtual void erase(const K&) = 0;
        // remove matching pair
    virtual void insert(const pair<const K, E>&) = 0;
        // insert a (key, value) pair into the dictionary
};
```

Program 10.1 The abstract class dictionary

10.3 LINEAR LIST REPRESENTATION

A dictionary may be maintained as an ordered linear list (p_0, p_1, \dots) where the p_i s are the dictionary pairs in ascending order of key. To facilitate this representation, we may define two classes `sortedArrayList` and `sortedChain`. The first uses an array representation of a linear list (see Section 5.3), while the latter uses a linked representation (see Section 6.1).

Exercise 5 asks you to develop the class `sortedArrayList`. We note that you can search a `sortedArrayList` using the binary search method. So the `find` operation takes $O(\log n)$ time for an n -pair dictionary. To make an insertion, we need to verify that the dictionary doesn't already contain a pair with the same key. This verification is done by performing a search (i.e., a `find`). Following this search, the insertion may be done in $O(n)$ additional time, as $O(n)$ pairs must be moved to make room for the new pair. Each `erase` is done by first searching for the pair to be erased and then erasing it. Following the search, the erasing takes $O(n)$ time as $O(n)$ pairs must be moved to fill up the vacancy left by the erased pair.

Programs 10.2, 10.3, and 10.4 give the `find`, `insert`, and `erase` methods of the class `sortedChain`. The nodes in `sortedChain` are instances of `pairNode`. Each instance of `pairNode`, like each instance of `chainNode` (Program 6.1), has an `element` and `next` field; the datatype of these fields are, respectively, `pair<const K, E>` and `pairNode<K, E>*`.

Using either the class `sortedArrayList` or `sortedChain` and the corresponding iterator methods, we can provide sequential access to the dictionary pairs; pairs can

```
template<class K, class E>
pair<const K,E>* sortedChain<K,E>::find(const K& theKey) const
// Return pointer to matching pair.
// Return NULL if no matching pair.
pairNode<K,E>* currentNode = firstNode;

// search for match with theKey
while (currentNode != NULL &&
       currentNode->element.first != theKey)
    currentNode = currentNode->next;

// verify match
if (currentNode != NULL && currentNode->element.first == theKey)
    // yes, found match
    return &currentNode->element;

// no match
return NULL;
}
```

Program 10.2 The method `sortedChain<K,E>::find`

be examined in ascending order of keys at a cost of $\Theta(1)$ time per pair.

EXERCISES

5. Develop the C++ class `sortedArrayList` that uses an array representation. Provide the same member methods as provided in the class `sortedChain`. Write and test code for all methods.
6. Modify the class `sortedChain` to use a chain that has both a header node and a tail node. Use the tail node to simplify your code by placing the key being searched for, inserted, or erased into the tail node at the start of the operation.

10.4 SKIP LIST REPRESENTATION (OPTIONAL)

10.4.1 The Ideal Case

A search in an n -pair dictionary that is represented as a sorted chain requires up to n key comparisons. The number of comparisons can be reduced to $n/2 + 1$ if we

```

template<class K, class E>
void sortedChain<K,E>::insert(const pair<const K, E>& thePair)
{// Insert thePair into the dictionary. Overwrite existing
// pair, if any, with same key.
pairNode<K,E> *p = firstNode,
    *tp = NULL; // tp trails p

// move tp so that thePair can be inserted after tp
while (p != NULL && p->element.first < thePair.first)
{
    tp = p;
    p = p->next;
}

// check if there is a matching pair
if (p != NULL && p->element.first == thePair.first)
{// replace old value
    p->element.second = thePair.second;
    return;
}

// no match, set up node for thePair
pairNode<K,E> *newNode = new pairNode<K,E>(thePair, p);

// insert newNode just after tp
if (tp == NULL) firstNode = newNode;
else tp->next = newNode;

dSize++;
return;
}

```

Program 10.3 The method `sortedChain<K,E>::insert`

keep a pointer to the middle pair. Now to search for a pair, we first compare with the middle one. If we are looking for a pair with smaller key, we need search only the left half of the sorted chain. If we are looking for a larger key, we need compare only the right half of the chain.

Example 10.4 Consider the seven-pair sorted chain of Figure 10.1(a). This sorted chain has been augmented by a header node and a tail node. The number inside a node is its key. A search of this chain may involve up to seven key comparisons. We

```

template<class K, class E>
void sortedChain<K,E>::erase(const K& theKey)
{// Delete the pair, if any, whose key equals theKey.
    pairNode<K,E> *p = firstNode,
        *tp = NULL; // tp trails p

    // search for match with theKey
    while (p != NULL && p->element.first < theKey)
    {
        tp = p;
        p = p->next;
    }

    // verify match
    if (p != NULL && p->element.first == theKey)
    {// found a match
        // remove p from the chain
        if (tp == NULL) firstNode = p->next; // p is first node
        else tp->next = p->next;

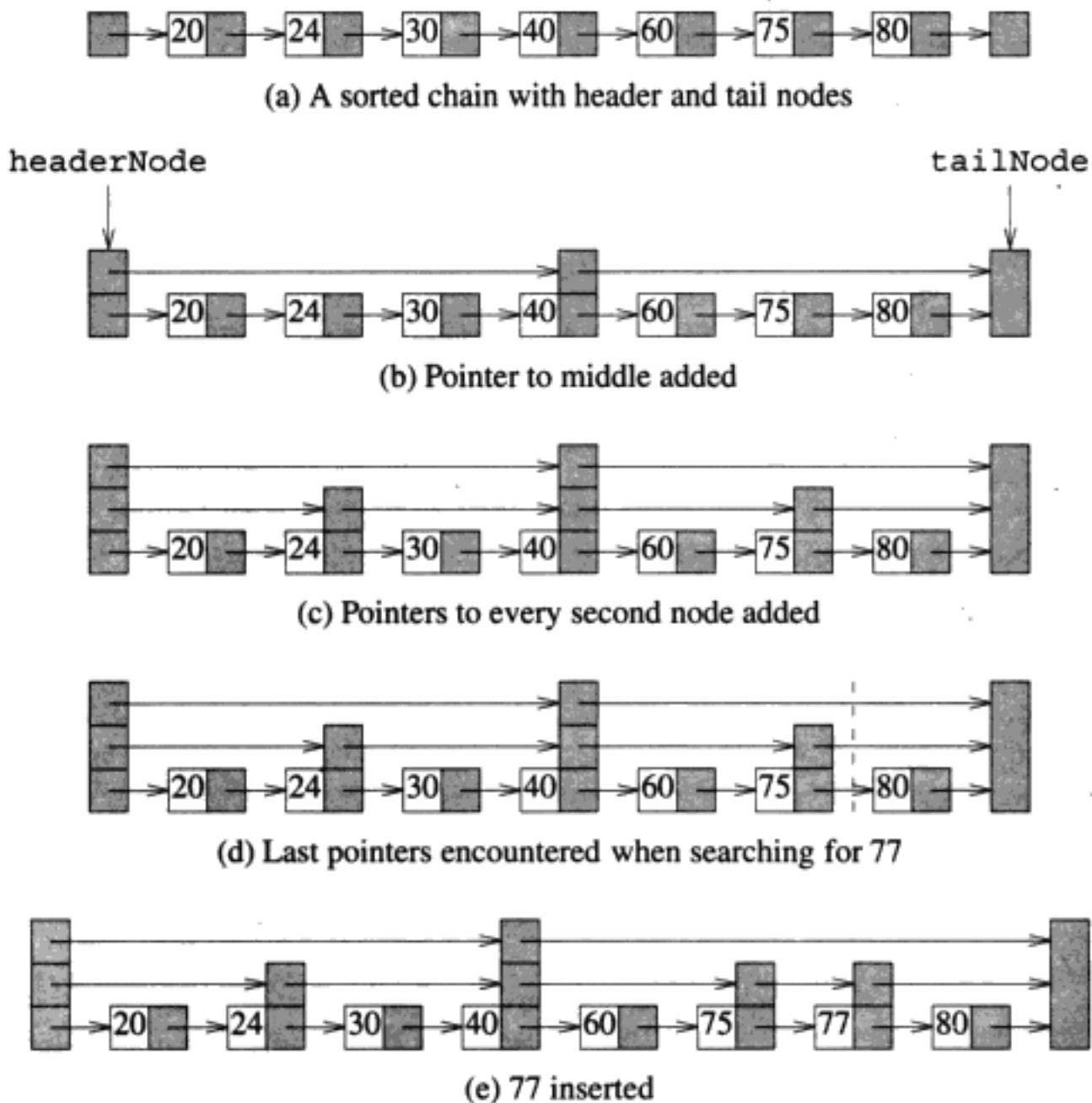
        delete p;
        dSize--;
    }
}

```

Program 10.4 The method sortedChain<K,E>::erase

can reduce this worst-case number of comparisons to 4 by keeping a pointer to the middle pair as in Figure 10.1(b). Now to search for a key, we first compare with the key of the middle pair and then, depending on the outcome, compare with either the left or right half of the chain. If we are looking for a pair with key 26, then we begin by comparing 26 with the middle key 40. Since $26 < 40$, we need not examine the pairs to the right of 40. If we are searching for a pair with key 75, then we can limit the search to the pairs that follow 40. ■

We can reduce the worst-case number of key comparisons by keeping pointers to the middle pairs of each half as in Figure 10.1(c). The level 0 chain is essentially that of Figure 10.1(a) and includes all seven pairs of the dictionary. The level 1 chain includes the second, fourth, and sixth pairs, while the level 2 chain includes only the fourth pair. To search for a pair with key 30, we begin with a comparison against the middle key. This key is found in $\Theta(1)$ time using the level 2 chain. Since $30 < 40$, the search continues by examining the middle key of the left half. This

**Figure 10.1** Fast searching of a sorted chain

key is also found in $\Theta(1)$ time using the level 1 chain. Since $30 > 24$, we continue the search by dropping into the level 0 chain and comparing with the next key in this chain.

As another example, consider the search for a pair with key 77. The first comparison is with 40. Since $77 > 40$, we drop into the level 1 chain and compare with the key (75) in this chain that comes just after 40. Since $77 > 75$, we drop into the level 0 chain and compare with the key (80) in this chain that comes just after 75.

At this time we know that 77 is not in the dictionary. Using the three-chain structure of Figure 10.1(c), we can perform all searches using at most three comparisons. The three-chain structure allows us to perform a binary search in the sorted chain.

For general n the level 0 chain includes all pairs; the level 1 chain includes every second pair; the level 2 chain every fourth pair; and the level i chain every 2^i th pair. We will say that a pair is a **level i pair** iff it is in the chains for levels 0 through i and it is not on the level $i + 1$ chain (in case this chain exists). In Figure 10.1(c), 40 is the (key of the) only level 2 pair; 24 and 75 are the level 1 pairs; and 20, 30, 60, and 80 are the level 0 pairs.

We will use the term **skip list** to refer to a structure such as that of Figure 10.1(c). In such a structure we have a hierarchy of chains. The level 0 chain is a sorted chain of all pairs. The level 1 chain is also a sorted chain that comprises some subset of the pairs on the level 0 chain. In general, the level i chain comprises a subset of the pairs in the level $i - 1$ chain. The skip list of Figure 10.1(c) has a very regular structure in that the level i chain comprises every other pair of the level $i - 1$ chain.

10.4.2 Insertions and Deletions

When insertions and deletions occur, we cannot maintain the regular structure of Figure 10.1(c) without doing $O(n)$ work. We can attempt to approximate this structure in the face of insertions by noting that in the regular structure, $n/2^i$ pairs are level i pairs. When an insertion is made, the pair level is i with probability $1/2^i$. We can actually allow for any probability to be used when making this determination. Therefore, we can assign the newly inserted pair at level i with probability p^i . Figure 10.1(c) corresponds to the case $p = 0.5$. For general p the number of chain levels is $\lfloor \log_{1/p} n \rfloor + 1$. In this case a regular skip list structure has the property that the level i chain comprises every $1/p$ th pair of the level $i - 1$ chain.

Suppose we are to insert a pair with key 77. We first search to make sure that no pair with this key is present. During this search the last level 2 pointer seen is associated with key 40, the last level 1 pointer seen is associated with key 75, and the last level 0 pointer seen is associated with key 75. These pointers are cut by the broken line of Figure 10.1(d). The new pair is to be inserted between the pairs with keys 75 and 80 at the position shown by the broken line of Figure 10.1(d).

To make the insertion, we need to assign a level to the new pair. This assignment can be made by using a random number generator as described later. If the new pair is a level i pair, then only the level 0 through level i pointers cut by the broken line are affected. Figure 10.1(e) shows the list structure following the insertion of 77 as a level 1 pair.

We have no control over the structure that is left following a deletion. To delete the 77 from the skip list structure of Figure 10.1(e), we first search for 77. The last pointers encountered in the chains are the level 2 pointer in the node with 40 and the level 1 and level 0 pointers in the node with 75. Of these pointers only the

level 0 and level 1 pointers are to be changed, as 77 is a level 1 pair. When these pointers are changed to point to the pair after 77 in their respective chains, we get the structure of Figure 10.1(d).

10.4.3 Assigning Levels

The basis of level assignment is the observation that in a regular skip list structure, a fraction p of the pairs on the level $i - 1$ chain are also on the level i chain. Therefore, the probability that a pair that is on the level $i - 1$ chain is also on the level i chain is p . Suppose we have a uniform random number generator that generates real numbers between 0 and 1. Then the probability that the next random number is $\leq p$ is p . Consequently, if the next random number is $\leq p$, then the new pair should be on the level 1 chain. Now we need to decide whether it should also be on the level 2 chain. To make this decision, we simply generate another random number. If the new random number is $\leq p$, then the pair is also on the level 2 chain. We can continue this process until a random number $> p$ is generated.

A potential shortcoming of this way of assigning levels is that some pairs may be assigned a very large level, resulting in a number of chains far in excess of $\log_{1/p} N$ where N is the maximum number of pairs expected in the dictionary. To prevent this possibility, we can set an upper limit to the assignable level number. In a regular skip list structure with N pairs, the maximum level `maxLevel` is

$$\lceil \log_{1/p} N \rceil - 1 \quad (10.1)$$

We can use this value as the upper limit.

Another shortcoming is that even with the use of an upper limit as above, we may find ourselves in a situation where, for example, we have 3 chains just before the insertion and 10 just after. In this case the new pair was assigned the level 9 even though there were no pairs at levels 3 through 8 prior to the insertion. In other words, prior to and following the insertion, there are no level 3, 4, ..., 8 pairs. Since there is no immediate benefit to having these empty levels, we may alter the level assignment of the pair to 3.

Example 10.5 We are using a skip list to represent a dictionary that will have no more than 1024 pairs. We have decided to use $p = 0.5$, so `maxLevel` is $\log_2 1024 - 1 = 9$.

Suppose we start with an empty dictionary that is represented by a skip list structure that has a header and a tail. The header has 10 pointers, one for each of the 10 chains we might have. Each pointer goes from the header to the tail.

When the first pair is inserted, it is assigned a level. The permissible levels are 0 through 9 (`maxLevel`). If the level assigned is 9, then to insert the first pair, we will need to change nine chain pointers. On the other hand, as we have no level 0, level 1, ..., level 8 pairs, we may alter the level assignment to 0 and change only one chain pointer. ■

An alternative way to assign levels is to divide the range of values the random number generator outputs into segments. The first segment contains $1 - 1/p$ of the range, the second $1/p - 1/p^2$ of the range, and so on. If the random number generated falls in the i th segment, the pair to be inserted is a level $i - 1$ pair.

10.4.4 The Struct skipNode

The header node of a skip list structure needs sufficient pointer fields for the maximum number of level chains that might be constructed. The tail node needs no pointer field. Each node that contains a dictionary pair needs an **element** field for the pair and a number of pointer fields that is one more than its level number. The struct **skipNode** of Program 10.5 can meet the needs of all kinds of nodes.

```
template <class K, class E>
struct skipNode
{
    typedef pair<const K, E> pairType;

    pairType element;
    skipNode<K,E> **next; // 1D array of pointers

    skipNode(const pairType& thePair, int size)
        :element(thePair){next = new skipNode<K,E>* [size];}
};
```

Program 10.5 The struct skipNode

The pointer fields are represented by the array **next** with **next[i]** being the pointer for the level i chain. The constructor places the dictionary pair into **element** and allocates space for the array of pointers. When the constructor is invoked, the value of **size** should be **lev + 1** for a level **lev** pair.

10.4.5 The Class skipList

The Data Members of skipList

Program 10.6 gives the data members of the class **skipList**. The significance of each data member should be clear from its name and the attached comment.

The Constructor for skipList

Program 10.7 gives the constructor. **largeKey** is a value larger than the key of any pair to be kept in the dictionary. The value **largeKey** is used in the tail node. **maxPairs** is the maximum number of pairs the dictionary is to hold. Although our

```

float cutOff;           // used to decide level number
int levels;            // max current nonempty chain
int dSize;             // number of pairs in dictionary
int maxLevel;          // max permissible chain level
K tailKey;             // a large key
skipNode<K,E>* headerNode; // header node pointer
skipNode<K,E>* tailNode;  // tail node pointer
skipNode<K,E>** last;    // last[i] = last node seen on level i

```

Program 10.6 The data members of skipList

```

template<class K, class E>
skipList<K,E>::skipList(K largeKey, int maxPairs, float prob)
{// Constructor for skip lists with keys smaller than largeKey and
// size at most maxPairs. 0 < prob < 1.
    cutOff = prob * RAND_MAX;
    maxLevel = (int) ceil(logf((float) maxPairs) / logf(1/prob)) - 1;
    levels = 0; // initial number of levels
    dSize = 0;
    tailKey = largeKey;

    // create header & tail nodes and last array
    pair<K,E> tailPair;
    tailPair.first = tailKey;
    headerNode = new skipNode<K,E> (tailPair, maxLevel + 1);
    tailNode = new skipNode<K,E> (tailPair, 0);
    last = new skipNode<K,E> *[maxLevel+1];

    // header points to tail at all levels as lists are empty
    for (int i = 0; i <= maxLevel; i++)
        headerNode->next[i] = tailNode;
}

```

Program 10.7 Constructor

codes permit more pairs than `maxPairs`, the expected performance is better if the number of pairs does not exceed `maxPairs`, as the number of chains is limited by substituting `maxPairs` for N in Equation 10.1. `prob` is the probability that a pair in the level $i - 1$ chain is also in the level i chain. The constructor initializes the data members of `skipList` to the values stated earlier. The constructor also allocates

space for the header and tail nodes and the array `last` that is used to keep track of the last node encountered in each chain during the search phase that precedes an `insert` and `erase`; the skip list is initialized to the empty configuration in which we have `maxLevel+1` pointers from the header node to the tail node. The complexity of the constructor is $O(\text{maxLevel})$.

The Method `skipList<K,E>::find`

Program 10.8 gives the code for the method `find`. The method returns `NULL` in case no pair with key `theKey` is in the dictionary. When the dictionary contains a pair with key `theKey`, a pointer to this pair is returned.

```
template<class K, class E>
pair<const K,E>* skipList<K,E>::find(const K& theKey) const
{// Return pointer to matching pair.
// Return NULL if no matching pair.
    if (theKey >= tailKey)
        return NULL; // no matching pair possible

    // position beforeNode just before possible node with theKey
    skipNode<K,E>* beforeNode = headerNode;
    for (int i = levels; i >= 0; i--)           // go down levels
        // follow level i pointers
        while (beforeNode->next[i]->element.first < theKey)
            beforeNode = beforeNode->next[i];

    // check if next node has theKey
    if (beforeNode->next[0]->element.first == theKey)
        return &beforeNode->next[0]->element;

    return NULL; // no matching pair
}
```

Program 10.8 The method `skipList<K,E>::find`

`find` begins with the highest level chain—the level `levels` chain—that contains a pair and works its way down to the level 0 chain. At each level we advance as close to the pair being searched as possible without advancing to the right of the pair. Although we can terminate the search at level `i` if we reach a pair whose key equals `theKey`, the additional comparison needed to test for equality isn't justified because most pairs are expected to be only in the level 0 chain. When we exit from the `for` loop, we are positioned just to the left of the pair we seek. Comparing with

the next pair on the level 0 chain permits us to determine whether or not the pair we seek is in the structure.

The Method `skipList<K,E>::insert`

Before we can write code for the method to insert a pair into a skip list, we must write methods to assign a level number to a new pair and to search the skip list as is done by `find` but saving a pointer to the last node encountered at each level of the search. Programs 10.9 and 10.10 give these two methods.

```
template<class K, class E>
int skipList<K,E>::level() const
{// Return a random level number <= maxLevel.
    int lev = 0;
    while (rand() <= cutOff)
        lev++;
    return (lev <= maxLevel) ? lev : maxLevel;
}
```

Program 10.9 Method to assign a level number

Program 10.11 gives the code to insert the pair `thePair` into a skip list. `thePair` is not inserted if `largeKey ≤ thePair.first`. Also, if the skip list already has a pair whose key is `thePair.first`, the second component of this existing pair is changed to `thePair.second`.

The Method `skipList<K,E>::erase`

Program 10.12 gives the code to erase the dictionary pair, if any, with key `theKey`. The `while` loop updates `levels` so that there is at least one level `levels` pair unless the skip list is empty. In the latter case `levels` is set to 0.

Other Methods

The codes for other methods such as `size` and `empty`, and the iterator methods are similar to the codes for the corresponding methods of `chain`. Recall that the pairs in the chain for each level (excluding the header node, which has no key) are in ascending order from left to right. In particular, the level 0 chain contains all pairs in the dictionary in ascending order of their key. Therefore the `skipList` iterator is able to provide sequential access to the dictionary pairs in sorted order in $\Theta(1)$ time per pair accessed.

```
template<class K, class E>
pair<const K,E>* skipList<K,E>::find(const K& theKey) const
{// Return pointer to matching pair.
// Return NULL if no matching pair.
if (theKey >= tailKey)
    return NULL; // no matching pair possible

// position beforeNode just before possible node with theKey
skipNode<K,E>* beforeNode = headerNode;
for (int i = levels; i >= 0; i--) // go down levels
    // follow level i pointers
    while (beforeNode->next[i]->element.first < theKey)
        beforeNode = beforeNode->next[i];

// check if next node has theKey
if (beforeNode->next[0]->element.first == theKey)
    return &beforeNode->next[0]->element;

return NULL; // no matching pair
}
```

Program 10.10 Method to search a skip list and save the last node encountered at each level

10.4.6 Complexity of skipList Methods

The complexity of methods `find`, `insert`, and `erase` is $O(n+\maxLevel)$ where n is the number of pairs in the dictionary. In the worst case there may be only one level `maxLevel` pair, and the remaining pairs may all be level 0 pairs. Now $O(\maxLevel)$ time is spent on the level i chains for $i > 0$, and $O(n)$ time on the level 0 chain. Despite this poor worst-case performance, skip lists are a valuable representation method, as the expected complexity of methods `find`, `insert`, and `erase` is $O(\log n)$.

As for the space complexity, we note that in the worst case each pair might be a level `maxLevel` pair requiring `maxLevel+1` pointers. Therefore, in addition to the space needed to store the n pairs, we need space for $O(n*\maxLevel)$ pointers. On the average, however, only $n*p$ of the pairs are expected to be on the level 1 chain, $n*p^2$ on the level 2 chain, and $n*p^i$ on the level i chain. So the expected number of pointer fields (excluding those in the header and tail nodes) is $n \sum_i p^i = n/(1-p)$. So while the worst-case space requirements are large, the expected requirements are not. When $p = 0.5$, the expected space requirements (in addition to that for the dictionary pairs) is that for approximately $2n$ pointers!

```

template<class K, class E>
void skipList<K,E>::insert(const pair<const K, E>& thePair)
{// Insert thePair into the dictionary. Overwrite existing
// pair, if any, with same key.
    if (thePair.first >= tailKey) // key too large
    {ostringstream s;
     s << "Key = " << thePair.first << " Must be < " << tailKey;
     throw illegalParameterValue(s.str());
    }

    // see if element with theKey already present
    skipNode<K,E>* theNode = search(thePair.first);
    if (theNode->element.first == thePair.first)
    {// update theNode->element.second
        theNode->element.second = thePair.second;
        return;
    }

    // not present, determine level for new node
    int theLevel = level(); // level of new node
    // fix theLevel to be <= levels + 1
    if (theLevel > levels)
    {
        theLevel = ++levels;
        last[theLevel] = headerNode;
    }

    // get and insert new node just after theNode
    skipNode<K,E>* newNode = new skipNode<K,E>(thePair, theLevel + 1);
    for (int i = 0; i <= theLevel; i++)
    {// insert into level i chain
        newNode->next[i] = last[i]->next[i];
        last[i]->next[i] = newNode;
    }

    dSize++;
    return;
}

```

Program 10.11 Skip list insertion

```
template<class K, class E>
void skipList<K,E>::erase(const K& theKey)
{ // Delete the pair, if any, whose key equals theKey.
  if (theKey >= tailKey) // too large
    return;

  // see if matching element present
  skipNode<K,E>* theNode = search(theKey);
  if (theNode->element.first != theKey) // not present
    return;

  // delete node from skip list
  for (int i = 0; i <= levels &&
       last[i]->next[i] == theNode; i++)
    last[i]->next[i] = theNode->next[i];

  // update levels
  while (levels > 0 && headerNode->next[levels] == tailNode)
    levels--;

  delete theNode;
  dSize--;
}
```

Program 10.12 Erasing a pair from a skip list**EXERCISES**

7. Write a level allocation program that divides the range of random number values into segments as described in the text and then determines the level on the basis of which segment a random number falls into.
8. Modify the class **skipList** to allow for the presence of pairs that have the same key. Each chain is now in nondecreasing order of key from left to right. Test your code.
9. Extend the class **skipList** by including methods to erase the pair with smallest key and to erase the pair with largest key. What is the expected complexity of each method?

10.5 HASH TABLE REPRESENTATION

10.5.1 Ideal Hashing

Another possibility for the representation of a dictionary is to use **hashing**. This method uses a **hash function** to map dictionary pairs into positions in a table called the **hash table**. In the ideal situation, if pair p has the key k and f is the hash function, then p is stored in position $f(k)$ of the table. Assume for now that each position of the table can store at most one pair. To search for a pair with key k , we compute $f(k)$ and see whether a pair exists at position $f(k)$ of the table. If so, we have found the desired pair. If not, the dictionary contains no pair with the specified key k . In the former case the pair may be deleted (if desired) by making position $f(k)$ of the table empty. In the latter case the pair may be inserted by placing it in position $f(k)$.

Example 10.6 Consider the student records dictionary of Example 10.1. Suppose that instead of using student names as the key, we use student ID numbers, which are six-digit integers. For our class assume we will have at most 100 students and their ID numbers will be in the range 951000 and 952000. The function $f(k) = k - 951,000$ maps student IDs into table positions 0 through 1000. We may use an array `table[1001]` to store pointers to our dictionary pairs. This array is initialized so that `table[i]` is `NULL` for $0 \leq i \leq 1000$. To search for a pair with key k , we compute $f(k) = k - 951,000$. The pair (or more accurately, a pointer to the pair) is at `table[f(k)]` provided `table[f(k)]` is not `NULL`. If `table[f(k)]` is `null`, the dictionary contains no pair with key k . In the latter case the pair may be inserted at this position. In the former case the pair may be removed by setting `table[f(k)]` to `NULL`. ■

In the ideal situation just described, it takes $O(b)$ time to initialize an empty dictionary (b is the number of positions in the hash table) and $\Theta(1)$ time to perform a `find`, `insert`, or `erase` operation.

Although the ideal hashing solution just described may be used in many applications of a dictionary, in many other applications the range in key values is so large that a table either doesn't make sense or is impractical (or both).

Example 10.7 Suppose that in the class list example (Example 10.1) the student IDs are in the range [100000, 999999] and we are to use the hash function $f(k) = k - 100,000$. Since the value of $f()$ is in the range [0, 899,999], we need a table with 900,000 positions. It doesn't make sense to use a table this large for a class with only 100 students. Besides being terribly wasteful of space, it takes quite a bit of time to initialize the 900,000 array entries to `NULL`. ■

Example 10.8 [Converting Strings to Unique Numbers] Imagine you are maintaining a dictionary in which the keys are exactly three characters long. For example,

each key may be the initials in a name; the key for Mohandas Karamchand Gandhi would be MKG.

Since each character in C++ is 1 byte long, we could convert a three-character string into a long integer using the code of Program 10.13.

```
long threeToLong(string s)
{// Assume s.length() >= 3.
    // leftmost char
    long answer = s.at(0);

    // shift left 8 bits and add in next char
    answer = (answer << 8) + s.at(1);

    // shift left 8 bits and add in next char
    return (answer << 8) + s.at(2);
}
```

Program 10.13 Converting a three-character string to a long integer

When $s = abc$, $s.charAt(0) = a$, $s.charAt(1) = b$, and $s.charAt(2) = c$. If each of the characters a , b , and c is typecast into an integer, you get the numbers 97, 98, and 99, respectively. The left shifts (in Program 10.13) by 8 are done so that the bits of one character do not interfere with the bits of another character. Because of this shifting, different three-character strings convert to different long integers and it is possible to reconstruct s from `threeToLong(s)` (see Exercise 12).

Since a left shift by 8 is equivalent to multiplying by $2^8 = 256$, the computation performed by Program 10.13 when $s = abc$ is equivalent to computing $((97 * 256 + 98) * 256) + 99 = 6,382,179$.

Although Program 10.13 converts each three-character key into a unique long integer, the range of these long integers is $[0, 2^{24} - 1]$. ■

10.5.2 Hash Functions and Tables

Buckets and Home Buckets

When the key range is too large to use the ideal method described above, we use a hash table that has a number of positions that is smaller than the key range and a hash function $f(k)$ that maps several different keys into the same position of the hash table. Each position of the table is a **bucket**; $f(k)$ is the **home bucket** for the pair whose key is k ; and the number of buckets in a table equals the table length. Since a hash function may map several keys into the same bucket, we may consider designing buckets that can hold more than one pair. We consider two extremes in

this chapter. In the first, each bucket may hold just one pair and in the second, each bucket is a linear list of all pairs for which this bucket is the home bucket.

The Division Hash Function

Of the many hash functions that have been proposed, hashing by division is most common. In hashing by division, the hash function has the form

$$f(k) = k \% D \quad (10.2)$$

where k is the key, D is the length (i.e., number of buckets) of the hash table, and $\%$ is the modulo operator. The positions in the hash table are indexed 0 through $D - 1$. When $D = 11$, the home buckets for the keys 3, 22, 27, 40, 80, and 96 are $f(3) = 3$, $f(22) = 0$, $f(27) = 5$, 7, 3, and 8, respectively.

Other hash functions are described in this book's Web site.

Collisions and Overflows

Figure 10.2(a) shows a hash table with 11 buckets numbered 0 through 10, and each bucket has space for one pair. The figure shows only the key for each pair in the table. The divisor D that has been used is 11. The 80 is in position 3 because $80 \% 11 = 3$; the 40 is in position $40 \% 11 = 7$; and the 65 is in position $65 \% 11 = 10$. Each pair is in its home bucket. The remaining buckets in the hash table are empty.

Now suppose we wish to enter the key 58 into the table. The home bucket is $f(58) = 58 \% 11 = 3$. This bucket is already occupied by a different key. We say that a **collision** has occurred. A collision occurs whenever two different keys have the same home bucket. Since a bucket may be able to accommodate more than one pair, a collision may not be a problem. All pairs that have the same home bucket can be stored in their home bucket provided enough space is available in the home bucket. An **overflow** occurs when there isn't room in the home bucket for the new pair.

In our example each bucket can accommodate only one pair. So collisions and overflows occur at the same time. If we cannot put 58 into its home bucket, where should we put it? This question is answered by the overflow-handling mechanism in use. The most popular overflow-handling mechanism is linear probing (Section 10.5.3). Other mechanisms such as quadratic probing and double hashing are described in the Web site.

I Want a Good Hash Function

Although collisions may not give you a headache, they lead to overflows, which are guaranteed to cause a headache (unless a bucket can hold an unlimited number of pairs) as far as the insertion operation is concerned. The expected occurrence of

			80				40			65	
table[]	0	1	2	3	4	5	6	7	8	9	10
(a)											
		24	80	58			40			65	
table[]	0	1	2	3	4	5	6	7	8	9	10
(b)											
		24	80	58	35		40			65	
table[]	0	1	2	3	4	5	6	7	8	9	10
(c)											

Figure 10.2 Hash tables

collisions and overflows is minimized when approximately the same number of keys from the key range hashes into any bucket of the table. A **uniform hash function** provides such a distribution of home buckets.

Example 10.9 [Uniform Hash Function] Assume that our hash table has $b > 1$ buckets numbered 0 through $b - 1$. The hash function $f(k) = 0$ for all k is not a uniform hash function because all keys hash into the same bucket, bucket 0. When this hash function is used, we get the maximum number of collisions and overflows that are possible. Suppose $b = 11$ and the key range is $[0, 98]$. A uniform hash function will hash approximately 9 of these keys into each bucket; when the key range is $[0, 999]$, approximately 91 keys are hashed into each bucket.

The function $f(k) = k \% b$ is a uniform hash function for every key range $[0, r]$ where r is a positive integer. When $r = 20$ and $b = 11$, for example, some buckets get two keys and others get one; and when $r = 50$ and $b = 11$, some buckets get five keys and others get four. No matter what r and $b > 1$ are, when division is used, some buckets get $\lfloor r/b \rfloor$ keys and the remaining buckets get $\lceil r/b \rceil$ keys. Division yields a uniform hash function. ■

If people would select the set of keys in their dictionary uniformly from the key range, the use of a uniform hash function would result in a uniform assignment of dictionary keys to home buckets. Unfortunately, we haven't figured out how to make people select keys in this manner. In practice, dictionary applications

have keys that show some degree of correlation. For example, when the keys are integers, you might have a preponderance of odd keys or even keys, rather than an equal number of odd and even keys; when the keys are alphanumeric, you may have clusters of keys that have the same prefix or the same suffix. Because keys in real-world applications are not selected uniformly from the key range, some uniform hash functions work better than others in providing an approximately equal distribution of keys to home buckets. These uniform hash functions that actually give a good distribution of keys in practice are called **good hash functions**.

Example 10.10 [Selecting the Hash Function Divisor D] When using the hash function $f(k) = k \% D$ ($D = b$), some choices of D result in a good hash function and other choices result in a bad hash function. As noted above, all choices of D , $D > 1$, result in a uniform hash function.

Suppose that D is an even integer. Now $f(k)$ is even whenever k is even, and $f(k)$ is odd whenever k is odd. For example, $k \% 20$ is even whenever k is even and is odd whenever k is odd. If your application has a preponderance of even keys, then many more keys get even home buckets than get odd home buckets. We do not get a uniform assignment of home buckets. The same is true when your application has a preponderance of odd keys. This time many more keys get an odd home bucket than get an even home bucket. So choosing D to be even gives us a bad hash function.

When D is divisible by small odd numbers such as 3, 5, and 7, hashing by division does not distribute the use of home buckets uniformly in real-world dictionaries. Therefore, for division by hashing to be a good hash function, we must choose a divisor D that is neither even nor divisible by small odd numbers. *The ideal choice for D is a prime number. When you cannot find a prime number close to the table length you have in mind, you should choose D so that it is not divisible by any number between 2 and 19.* Other considerations for the choice of D are discussed in Sections 10.5.3 and 10.5.4. ■

Because of the correlation among keys in a dictionary application, you should choose uniform hash functions whose value depends on all bits of the key (as opposed to just the first few, last few, or middle few). The hash functions described in this book's Web site have this property. Therefore, these hash functions are good hash functions. When using the division hash function, dependence on all bits is obtained by using an odd value for D . Best results are obtained when D (and therefore the number of buckets b) is either a prime number or has *no prime factors less than 20*.

Division and Nonintegral Keys

To use the division hash function, keys need to be converted to nonnegative integers before $f(k)$ can be computed. Since all hash functions hash several keys into the same home bucket, it is not necessary for us to convert keys into unique nonnegative integers. It is ok for us to convert the strings *data*, *structures*, and *algorithms* into the same integer (say, 199).

Example 10.11 [Converting Strings to Integers] Program 10.13 cannot be extended to convert strings with more than four characters into a number because a long integer has only 32 bits. Since it is not necessary to convert strings into unique nonnegative integers, we can map every string, no matter how long, into a 16-bit integer. Program 10.14 shows you one way to do this.

```

int stringToInt(string s)
{// Convert s into a nonnegative int that depends on all
// characters of s.
    int length = (int) s.length();    // number of characters in s
    int answer = 0;
    if (length % 2 == 1)
        {// length is odd
            answer = s.at(length - 1);
            length--;
        }

        // length is now even
        for (int i = 0; i < length; i += 2)
            {// do two characters at a time
                answer += s.at(i);
                answer += ((int) s.at(i + 1)) << 8;
            }

        return (answer < 0) ? -answer : answer;
}

```

Program 10.14 Converting a string into a nonunique integer

Program 10.14 converts pairs of characters into a unique integer, using the technique of Program 10.13, and then sums these unique integers. Although it would have been easier to simply add all the characters together (rather than shift every other one by 16 bits), doing so would give us integers that are not much more than 8 bits long; strings that are eight characters long would produce integers up to 11 bits long. Shifting by 16 bits allows us to cover the entire range of integers even with strings that are two characters long. ■

The C++ STL provides specializations of the template class `hash<T>` that transform instances of type `T` into a nonnegative integer of type `size_t`. Program 10.15 shows a possible specialization of `hash<T>` for the case when `T` is the STL class `string`. This specialization is identical to that used in the SGI STL specialization `hash<char*>`.

```

template<>
class hash<string>
{
public:
    size_t operator()(const string theKey) const
    { // Convert theKey to a nonnegative integer.
        unsigned long hashValue = 0;
        int length = (int) theKey.length();
        for (int i = 0; i < length; i++)
            hashValue = 5 * hashValue + theKey.at(i);

        return size_t(hashValue);
    }
};


```

Program 10.15 The specialization `hash<string>`

10.5.3 Linear Probing

The Method

The easiest way to find a place to put 58 into the table of Figure 10.2(a) is to search the table for the next available bucket and then put 58 into it. This method of handling overflows is called **linear probing** (also referred to as linear open addressing).

The 58 gets inserted into position 4. Suppose that the next key to be inserted is 24. $24 \% 11$ is 2. This bucket is empty, and so the 24 is placed there. Our hash table now has the form shown in Figure 10.2(b). Let us attempt to insert the key 35 into this table. Its home bucket (2) is full. Using linear probing, this key is placed in the next available bucket, and the table of Figure 10.2(c) results. As a final example, consider inserting 98 into the table. Its home bucket (10) is full. The next available bucket is 0, and the insertion is made into this bucket. So the search for the next available bucket is made by regarding the table as circular!

Having seen how insertions are made when linear probing is used, we can devise a method to search such a table. The search begins at the home bucket $f(k)$ of the key k we are searching for and continues by examining successive buckets in the table (regarding the table as circular) until one of the following happens: (1) a bucket containing a pair with key k is reached, in which case we have found the pair we were searching for; (2) an empty bucket is reached; and (3) we return to the home bucket. In the latter two cases, the table contains no pair with key k .

The deletion of a pair must leave behind a table on which the search method just described works correctly. If we are to delete the pair with key 58 from the

table of Figure 10.2(c), we cannot simply make position 4 of the table `NULL`. Doing so will result in the search method failing to find the pair with key 35. A deletion may require us to move several pairs. The search for pairs to move begins just after the bucket vacated by the deleted pair and proceeds to successive buckets until we either reach an empty bucket or we return to the bucket from which the deletion took place. When pairs are moved up the table following a deletion, we must take care not to move a pair to a position before its home bucket because making such a pair move would cause the search for this pair to fail.

An alternative to this rather cumbersome deletion strategy is to introduce the field `neverUsed` in each bucket. When the table is initialized, this field is set to `true` for all buckets. When a pair is placed into a bucket, its `neverUsed` field is set to `false`. Now condition (2) for search termination is replaced by: a bucket with its `neverUsed` field equal to `true` is reached. We accomplish a removal by setting the table position occupied by the removed pair to `NULL`. A new pair may be inserted into the first empty bucket encountered during a search that begins at the pair's home bucket. Notice that in this alternative scheme, `neverUsed` is never reset to `true`. After a while all (or almost all) buckets have this field equal to `false`, and unsuccessful searches examine all buckets. To improve performance, we must reorganize the table when many empty buckets have their `neverUsed` field equal to `false`. This reorganization could, for example, involve reinserting all remaining pairs into an empty hash table.

C++ Implementation of Linear Probing

Program 10.16 gives the data members and the constructor for our hash table class `hashTable` that uses linear probing. Notice that the hash table is defined as a one-dimensional array `table[]` of type `pair<const K, E>*`.

Program 10.17 gives the method `search` of `hashTable`. This method returns a bucket `b` in the table that satisfies exactly one of the following: (1) `table[b]` points to a pair whose key is `theKey`; (2) no pair in the table has the key `theKey`, `table[b]` is `NULL`, and the pair with key `theKey` may be inserted into bucket `b` if desired; and (3) no pair in the table has the key `theKey`, `table[b]` has a key other than `theKey`, and the table is full.

Program 10.18 implements the method `hashTable<K,E>::find`.

Program 10.19 gives the implementation of the method `insert`. This code begins by invoking the method `search`. From the specification of `search`, if the returned bucket `b` is empty, then there is no pair in the table with key `thePair.first` and the pair `thePair` may be inserted into this bucket. If the returned bucket is not empty, then it either contains a pair with key `thePair.first` or the table is full. In the former case we change the second component of the pair stored in the bucket to `thePair.second`; in the latter, we throw an exception (increasing the table size is an alternative to throwing an exception; this alternative is considered in Exercise 25). Exercise 26 asks you to write code for the method `erase`.

```

// data members of hashTable
pair<const K, E>** table; // hash table
hash<K> hash;           // maps type K to nonnegative integer
int dSize;               // number of pairs in dictionary
int divisor;             // hash function divisor

// constructor
template<class K, class E>
hashTable<K,E>::hashTable(int theDivisor)
{
    divisor = theDivisor;
    dSize = 0;

    // allocate and initialize hash table array
    table = new pair<const K, E>* [divisor];
    for (int i = 0; i < divisor; i++)
        table[i] = NULL;
}

```

Program 10.16 Data members and constructor for hashTable

```

template<class K, class E>
int hashTable<K,E>::search(const K& theKey) const
{// Search an open addressed hash table for a pair with key theKey.
 // Return location of matching pair if found, otherwise return
 // location where a pair with key theKey may be inserted
 // provided the hash table is not full.

    int i = (int)hash(theKey) % divisor; // home bucket
    int j = i; // start at home bucket
    do
    {
        if (table[j] == NULL || table[j]->first == theKey)
            return j;
        j = (j + 1) % divisor; // next bucket
    } while (j != i); // returned to home bucket?

    return j; // table full
}

```

Program 10.17 The method hashTable<K,E>::search

```

template<class K, class E>
pair<const K,E*>* hashTable<K,E>::find(const K& theKey) const
{ // Return pointer to matching pair.
  // Return NULL if no matching pair.
  // search the table
  int b = search(theKey);

  // see if a match was found at table[b]
  if (table[b] == NULL || table[b]->first != theKey)
    return NULL;           // no match

  return table[b]; // matching pair
}

```

Program 10.18 The method `hashTable<K,E>::find`

Performance Analysis

We will analyze the time complexity only. Let b be the number of buckets in the hash table. When division with divisor D is used as the hash function, $b = D$. The time needed to initialize the table is $O(b)$. The worst-case insert and find time is $\Theta(n)$ when n pairs are present in the table. The worst case happens, for instance, when all n key values have the same home bucket. Comparing the worst-case complexity of hashing to that of the linear list method to maintain a dictionary, we see that both have the same worst-case complexity.

For average performance, however, hashing is considerably superior. Let U_n and S_n , respectively, denote the average number of buckets examined during an unsuccessful and a successful search when n is large. This average is defined over all possible sequences of n key values being inserted into the table. For linear probing, it can be shown that

$$U_n \approx \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right) \quad (10.3)$$

$$S_n \approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) \quad (10.4)$$

where $\alpha = n/b$ is the **loading factor**. Although Equation 10.3 is rather difficult to derive, Equation 10.4 can be derived from Equation 10.3 with modest effort (Exercise 21a).

From Equations 10.3 and 10.4, it follows that when $\alpha = 0.5$, an unsuccessful search will examine 2.5 buckets on the average and an average successful search

```

template<class K, class E>
void hashTable<K,E>::insert(const pair<const K, E>& thePair)
{// Insert thePair into the dictionary. Overwrite existing
// pair, if any, with same key.
// Throw hashTableFull exception in case table is full.
// search the table for a matching element
int b = search(thePair.first);

// check if matching element found
if (table[b] == NULL)
{
    // no matching element and table not full
    table[b] = new pair<const K,E> (thePair);
    dSize++;
}
else
{// check if duplicate or table full
    if (table[b]->first == thePair.first)
        // duplicate, change table[b]->second
        table[b]->second = thePair.second;
    }
else // table is full
    throw hashTableFull();
}
}

```

Program 10.19 The method `hashTable<K,E>::insert`

will examine 1.5 buckets. When $\alpha = 0.9$, these figures are 50.5 and 5.5. These figures, of course, assume that n is much larger than 51. When it is possible to work with small loading factors, the average performance of hashing with linear probing is significantly superior to that of the linear list method. Generally, when linear probing is used, we try to keep $\alpha \leq 0.75$.

Analysis of Random Probing

To give you a taste of what is involved in determining U_n and S_n , we derive U_n and S_n formulas for the random probing method to handle overflows. In random probing, when an overflow occurs, the search for a free bucket in which the new key is inserted is done in a random manner (in practice, a pseudorandom number generator is used so we can reproduce the bucket search sequence and use this sequence in subsequent searches for the inserted pair).

Our derivation of the formula for U_n makes use of the following result from probability theory.

Theorem 10.1 *Let p be the probability that a certain event occurs. The expected number of independent trials needed for that event to occur is $1/\alpha$.*

To get a feel for the validity of Theorem 10.1, suppose that you flip a coin. The probability that the coin lands heads up is $p = 1/2$. The number of times you expect to flip the coin before it lands heads up is $1/p = 2$. A die has six sides labeled 1 through 6. When you throw a die, the probability of drawing an odd number is $p = 1/2$. You expect to throw the die $1/p = 2$ times before drawing an odd number. The probability that a die throw draws a 6 is $p = 1/6$, so you expect to throw the die $1/p = 6$ times before drawing a 6.

The formula for U_n is derived as follows. When the loading density is α , the probability that any bucket is occupied is also α . Therefore, the probability that a bucket is empty is $p = 1 - \alpha$. In random probing an unsuccessful search looks for an empty bucket, using a sequence of independent trials. Therefore, the expected number of buckets examined is

$$U_n \approx \frac{1}{p} = \frac{1}{1 - \alpha} \quad (10.5)$$

The equation for S_n may be derived from that for U_n . Number the n pairs in the table $1, 2, \dots, n$ in the order they were inserted. When the i th pair is inserted, an unsuccessful search is done and the pair is inserted into the empty bucket where the unsuccessful search terminates. At the time the i th pair is inserted, the loading factor is $(i-1)/b$ where b is the number of buckets. From Equation 10.5 it follows that the expected number of buckets that are to be examined when searching for the i th pair is

$$\frac{1}{1 - \frac{i-1}{b}}$$

Assuming that each pair in the table is searched for with equal probability, we get

$$\begin{aligned} S_n &\approx \frac{1}{n} \sum_{i=1}^n \frac{1}{1 - \frac{i-1}{b}} \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1 - \frac{i}{b}} \\ &\approx \frac{1}{n} \int_{i=0}^{n-1} \frac{1}{1 - \frac{i}{b}} di \end{aligned}$$

$$\begin{aligned}
 &\approx \frac{1}{n} \int_{i=0}^n \frac{1}{1 - \frac{i}{b}} di \\
 &= -\frac{b}{n} \log_e(1 - i/b) \Big|_0^n \\
 &= -\frac{1}{\alpha} \log_e(1 - \alpha)
 \end{aligned} \tag{10.6}$$

Linear probing incurs a performance penalty relative to random probing as far as the number of examined buckets is concerned. For example, when $\alpha = 0.9$, an unsuccessful search using linear probing is expected to examine 50.5 buckets; when random probing is used, this expected number drops to 10. So why do we not use random probing? Here are two reasons:

- Our real interest is run time, not number of buckets examined. It takes more time to compute the next random number than it does to examine several buckets.
- Since random probing searches the table in a random fashion, it pays a run-time penalty because of the cache effect (Section 4.5). Therefore, even though random probing examines a smaller number of buckets than does linear probing, examining this smaller number of buckets actually takes more time except when the loading factor is close to 1.

Choosing a Divisor D

To determine D , we first determine what constitutes acceptable performance for unsuccessful and successful searches. Using the formulas for U_n and S_n , we can determine the largest α that can be used. From the value of n (or an estimate) and the computed value of α , we obtain the smallest permissible value for b . Next we find the smallest integer that is at least as large as this value of b and that either is a prime or has no factors smaller than 20. This integer is the value of D and b to use.

Example 10.12 We are to design a hash table for up to 1000 pairs. Successful searches should require no more than four bucket examinations on average, and unsuccessful searches should examine no more than 50.5 buckets on average. From the formula for U_n , we obtain $\alpha \leq 0.9$, and from that for S_n , we obtain $4 \geq 0.5 + 1/(2(1-\alpha))$ or $\alpha \leq 6/7$. Therefore, we require $\alpha \leq \min\{0.9, 6/7\} = 6/7$. Hence b should be at least $\lceil 7n/6 \rceil = 1167$. $b = D = 1171$ is a suitable choice. ■

Another way to compute D is to begin with a knowledge of the largest possible value for b as determined by the maximum amount of space available for the hash table. Now we find the largest D no larger than this largest value that is either a prime or has no factors smaller than 20. For instance, if we can allot at most 530 buckets to the table, then $23 * 23 = 529$ is the right choice for D and b .