SECOND EDITION

COMPUTER SCIENCE

HANDBOOK

EDITOR-IN-CHIEF

ALLEN B. TUCKER

CHAPMAN & HALL/CRC

Published in Cooperation with ACM, The Association for Computing Machinery

© 2004 by Taylor & Francis Group, LLC

Library of Congress Cataloging-in-Publication Data

Computer science handbook / editor-in-chief, Allen B. Tucker—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-360-X (alk. paper)
1.Computer science-Handbooks, manuals, etc. 2. Engineering—Hanbooks, manuals, etc.
I. Tucker, Allen B.

QA76.C54755 2004
004—dc22 2003068758

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior permission in writing from the publisher.

All rights reserved. Authorization to photocopy items for internal or personal use, or the personal or internal use of specific clients, may be granted by CRC Press LLC, provided that $1.50 per page photocopied is paid directly to Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923 USA. The fee code for users of the Transactional Reporting Service is ISBN 1-58488-360-X/04/$0.00+$1.50. The fee is subject to change without notice. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

The consent of CRC Press LLC does not extend to copying for general distribution, for promotion, for creating new works, or for resale. Specific permission must be obtained in writing from CRC Press LLC for such copying.

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation, without intent to infringe.

Visit the CRC Press Web site at www.crcpress.com

© 2004 by Chapman & Hall/CRC

No claim to original U.S. Government works

International Standard Book Number 1-58488-360-X
Library of Congress Card Number 2003068758
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

© 2004 by Taylor & Francis Group, LLC

Preface to the Second Edition

Purpose

The purpose of *The Computer Science Handbook* is to provide a single comprehensive reference for computer scientists, software engineers, and IT professionals who wish to broaden or deepen their understanding in a particular subfield of computer science. Our goal is to provide the most current information in each of the following eleven subfields in a form that is accessible to students, faculty, and professionals in computer science:

> algorithms, architecture, computational science, graphics, human-computer interaction, information management, intelligent systems, net-centric computing, operating systems, programming languages, and software engineering

Each of the eleven sections of the *Handbook* is dedicated to one of these subfields. In addition, the appendices provide useful information about professional organizations in computer science, standards, and languages. Different points of access to this rich collection of theory and practice are provided through the table of contents, two introductory chapters, a comprehensive subject index, and additional indexes.

A more complete overview of this *Handbook* can be found in Chapter 1, which summarizes the contents of each of the eleven sections. This chapter also provides a history of the evolution of computer science during the last 50 years, as well as its current status, and future prospects.

New Features

Since the first edition of the *Handbook* was published in 1997, enormous changes have taken place in the discipline of computer science. The goals of the second edition of the *Handbook* are to incorporate these changes by:

1. Broadening its reach across all 11 subject areas of the discipline, as they are defined in *Computing Curricula 2001* (the new standard taxonomy)
2. Including a heavier proportion of applied computing subject matter
3. Bringing up to date all the topical discussions that appeared in the first edition

This new edition was developed by the editor-in-chief and three editorial advisors, whereas the first edition was developed by the editor and ten advisors. Each edition represents the work of over 150 contributing authors who are recognized as experts in their various subfields of computer science.

Readers who are familiar with the first edition will notice the addition of many new chapters, reflecting the rapid emergence of new areas of research and applications since the first edition was published. Especially exciting are the addition of new chapters in the areas of computational science, information

© 2004 by Taylor & Francis Group, LLC

management, intelligent systems, net-centric computing, and software engineering. These chapters explore topics like cryptography, computational chemistry, computational astrophysics, human-centered software development, cognitive modeling, transaction processing, data compression, scripting languages, multimedia databases, event-driven programming, and software architecture.

Acknowledgments

A work of this magnitude cannot be completed without the efforts of many individuals. During the 2-year process that led to the first edition, I had the pleasure of knowing and working with ten very distinguished, talented, and dedicated editorial advisors:

Harold Abelson (MIT), Mikhail Atallah (Purdue), Keith Barker (Uconn), Kim Bruce (Williams), John Carroll (VPI), Steve Demurjian (Uconn), Donald House (Texas A&M), Raghu Ramakrishnan (Wisconsin), Eugene Spafford (Purdue), Joe Thompson (Mississippi State), and Peter Wegner (Brown).

For this edition, a new team of trusted and talented editorial advisors helped to reshape and revitalize the *Handbook* in valuable ways:

Robert Cupper (Allegheny), Fadi Deek (NJIT), Robert Noonan (William and Mary)

All of these persons provided valuable insights into the substantial design, authoring, reviewing, and production processes throughout the first eight years of this *Handbook*'s life, and I appreciate their work very much.

Of course, it is the chapter authors who have shared in these pages their enormous expertise across the wide range of subjects in computer science. Their hard work in preparing and updating their chapters is evident in the very high quality of the final product. The names of all chapter authors and their current professional affiliations are listed in the contributor list.

I want also to thank Bowdoin College for providing institutional support for this work. Personal thanks go especially to Craig McEwen, Sue Theberge, Matthew Jacobson-Carroll, Alice Morrow, and Aaron Olmstead at Bowdoin, for their various kinds of support as this project has evolved over the last eight years. Bob Stern, Helena Redshaw, Joette Lynch, and Robert Sims at CRC Press also deserve thanks for their vision, perseverance and support throughout this period.

Finally, the greatest thanks is always reserved for my wife Meg – my best friend and my love – for her eternal influence on my life and work.

Allen B. Tucker
Brunswick, Maine

© 2004 by Taylor & Francis Group, LLC

Editor-in-Chief

Allen B. Tucker is the Anne T. and Robert M. Bass Professor of Natural Sciences in the Department of Computer Science at Bowdoin College, where he has taught since 1988. Prior to that, he held similar positions at Colgate and Georgetown Universities. Overall, he has served eighteen years as a department chair and two years as an associate dean. At Colgate, he held the John D. and Catherine T. MacArthur Chair in Computer Science.

Professor Tucker earned a B.A. in mathematics from Wesleyan University in 1963 and an M.S. and Ph.D. in computer science from Northwestern University in 1970. He is the author or coauthor of several books and articles in the areas of programming languages, natural language processing, and computer science education. He has given many talks, panel discussions, and workshop presentations in these areas, and has served as a reviewer for various journals, NSF programs, and curriculum projects. He has also served as a consultant to colleges, universities, and other institutions in the areas of computer science curriculum, software design, programming languages, and natural language processing applications.

A Fellow of the ACM, Professor Tucker co-authored the 1986 Liberal Arts Model Curriculum in Computer Science and co-chaired the ACM/IEEE-CS Joint Curriculum Task Force that developed Computing Curricula 1991. For these and other related efforts, he received the ACM's 1991 Outstanding Contribution Award, shared the IEEE's 1991 Meritorious Service Award, and received the ACM SIGCSE's 2001 Award for Outstanding Contributions to Computer Science Education. In Spring 2001, he was a Fulbright Lecturer at the Ternopil Academy of National Economy (TANE) in Ukraine. Professor Tucker has been a member of the ACM, the NSF CISE Advisory Committee, the IEEE Computer Society, Computer Professionals for Social Responsibility, and the Liberal Arts Computer Science (LACS) Consortium.

© 2004 by Taylor & Francis Group, LLC

Contributors

Eric W. Allender
Rutgers University

James L. Alty
Loughborough University

Thomas E. Anderson
University of Washington

M. Pauline Baker
National Center for Supercomputing Applications

Steven Bellovin
AT&T Research Labs

Andrew P. Bernat
Computer Research Association

Brian N. Bershad
University of Washington

Christopher M. Bishop
Microsoft Research

Guy E. Blelloch
Carnegie Mellon University

Philippe Bonnet
University of Copenhagen

Jonathan P. Bowen
London South Bank University

Kim Bruce
Williams College

Steve Bryson
NASA Ames Research Center

Douglas C. Burger
University of Wisconsin at Madison

Colleen Bushell
National Center for Supercomputing Applications

Derek Buzasi
U.S. Air Force Academy

William L. Bynum
College of William and Mary

Bryan M. Cantrill
Sun Microsystems, Inc.

Luca Cardelli
Microsoft Research

David A. Caughy
Cornell University

Vijay Chandru
Indian Institute of Science

Steve J. Chapin
Syracuse University

Eric Chown
Bowdoin College

Jacques Cohen
Brandeis University

J.L. Cox
Brooklyn College, CUNY

Alan B. Craig
National Center for Supercomputing Applications

Maxime Crochemore
University of Marne-la-Vallée and King's College London

Robert D. Cupper
Allegheny College

Thomas Dean
Brown Univeristy

Fadi P. Deek
New Jersey Institute of Technology

Gerald DeJong
University of Illinois at Urbana-Champaign

Steven A. Demurjian Sr.
University of Connecticut

Peter J. Denning
Naval Postgraduate School

Angel Diaz
IBM Research

T.W. Doeppner Jr.
Brown University

© 2004 by Taylor & Francis Group, LLC

Henry Donato
College of Charleston

Chitra Dorai
IBM T.J. Watson
Research Center

Wolfgang Dzida
Pro Context GmbH

David S. Ebert
Purdue University

Raimund Ege
Florida International
University

Osama Eljabiri
New Jersey Institute
of Technology

David Ferbrache
U.K. Ministry of Defence

Raphael Finkel
University of Kentucky

John M. Fitzgerald
Adept Technology

Michael J. Flynn
Stanford University

Kenneth D. Forbus
Northwestern University

Stephanie Forrest
University of New Mexico

Michael J. Franklin
University of California
at Berkeley

John D. Gannon
University of Maryland

Carlo Ghezzi
Politecnico di Milano

Benjamin Goldberg
New York University

James R. Goodman
University of Wisconsin
at Madison

Jonathan Grudin
Microsoft Research

Gamil A. Guirgis
College of Charleston

Jon Hakkila
College of Charleston

Sandra Harper
College of Charleston

Frederick J. Heldrich
College of Charleston

Katherine G. Herbert
New Jersey Institute
of Technology

Michael G. Hinchey
NASA Goddard Space
Flight Center

Ken Hinckley
Microsoft Research

Donald H. House
Texas A&M University

Windsor W. Hsu
IBM Research

Daniel Huttenlocher
Cornell University

Yannis E. Ioannidis
University of Wisconsin

Robert J.K. Jacob
Tufts University

Sushil Jajodia
George Mason University

Mehdi Jazayeri
Technical University of Vienna

Tao Jiang
University of California

Michael J. Jipping
Hope College

Deborah G. Johnson
University of Virginia

Michael I. Jordan
University of California
at Berkeley

David R. Kaeli
Northeastern University

Erich Kaltófen
North Carolina State University

Subbarao Kambhampati
Arizona State University

Lakshmi Kantha
University of Colorado

Gregory M. Kapfhammer
Allegheny College

Jonathan Katz
University of Maryland

Arie Kaufman
State University of New York
at Stony Brook

Samir Khuller
University of Maryland

David Kieras
University of Michigan

David T. Kingsbury
Gordon and Betty Moore
Foundation

Danny Kopec
Brooklyn College, CUNY

Henry F. Korth
Lehigh University

© 2004 by Taylor & Francis Group, LLC

Kristin D. Krantzman
College of Charleston

Edward D. Lazowska
University of Washington

Thierry Lecroq
University of Rouen

D.T. Lee
Northwestern University

Miriam Leeser
Northeastern University

Henry M. Levy
University of Washington

Frank L. Lewis
University of Texas at Arlington

Ming Li
University of Waterloo

Ying Li
IBM T.J. Watson
Research Center

Jianghui Liu
New Jersey Institute
of Technology

Kai Liu
Alcatel Telecom

Kenneth C. Louden
San Jose State University

Michael C. Loui
University of Illinois at
Urbana-Champaign

James J. Lu
Emory University

Abby Mackness
Booz Allen Hamilton

Steve Maddock
University of Sheffield

Bruce M. Maggs
Carnegie Mellon University

Dino Mandrioli
Politecnico di Milano

M. Lynne Markus
Bentley College

Tony A. Marsland
University of Alberta

Edward J. McCluskey
Stanford University

James A. McHugh
New Jersey Institute
of Technology

Marshall Kirk McKusick
Consultant

Clyde R. Metz
College of Charleston

Keith W. Miller
University of Illinois

Subhasish Mitra
Stanford University

Stuart Mort
U.K. Defence and Evaluation
Research Agency

Rajeev Motwani
Stanford University

Klaus Mueller
State University of New York
at Stony Brook

Sape J. Mullender
Lucent Technologies

Brad A. Myers
Carnegie Mellon University

Peter G. Neumann
SRI International

Jakob Nielsen
Nielsen Norman Group

Robert E. Noonan
College of William and Mary

Ahmed K. Noor
Old Dominion University

Vincent Oria
New Jersey Institute
of Technology

Jason S. Overby
College of Charleston

M. Tamer Özsu
University of Waterloo

Victor Y. Pan
Lehman College, CUNY

Judea Pearl
University of California
at Los Angeles

Jih-Kwon Peir
University of Florida

Radia Perlman
Sun Microsystems Laboratories

Patricia Pia
University of Connecticut

Steve Piacsek
Naval Research Laboratory

Roger S. Pressman
R.S. Pressman & Associates,
Inc.

J. Ross Quinlan
University of New South Wales

Balaji Raghavachari
University of Texas at Dallas

Prabhakar Raghavan
Verity, Inc.

© 2004 by Taylor & Francis Group, LLC

Z. Rahman
College of William and Mary

M.R. Rao
Indian Institute of Management

Bala Ravikumar
University of Rhode Island

Kenneth W. Regan
State University of New York at Buffalo

Edward M. Reingold
Illinois Institute of Technology

Alyn P. Rockwood
Colorado School of Mines

Robert S. Roos
Allegheny College

Erik Rosenthal
University of New Haven

Kevin W. Rudd
Intel, Inc.

Betty Salzberg
Northeastern University

Pierangela Samarati
Universitá degli Studi di Milano

Ravi S. Sandhu
George Mason University

David A. Schmidt
Kansas State University

Stephen B. Seidman
New Jersey Institute of Technology

Stephanie Seneff
Massachusetts Institute of Technology

J.S. Shang
Air Force Research

Dennis Shasha
Courant Institute
New York University

William R. Sherman
National Center for Supercomputing Applications

Avi Silberschatz
Yale University

Gurindar S. Sohi
University of Wisconsin at Madison

Ian Sommerville
Lancaster University

Bharat K. Soni
Mississippi State University

William Stallings
Consultant and Writer

John A. Stankovic
University of Virginia

S. Sudarshan
IIT Bombay

Earl E. Swartzlander Jr.
University of Texas at Austin

Roberto Tamassia
Brown University

Patricia J. Teller
University of Texas at ElPaso

Robert J. Thacker
McMaster University

Nadia Magnenat Thalmann
University of Geneva

Daniel Thalmann
Swiss Federal Institute of Technology (EPFL)

Alexander Thomasian
New Jersey Institute of Technology

Allen B. Tucker
Bowdoin College

Jennifer Tucker
Booz Allen Hamilton

Patrick Valduriez
INRIA and IRIN

Jason T.L. Wang
New Jersey Institute of Technology

Colin Ware
University of New Hampshire

Alan Watt
University of Sheffield

Nigel P. Weatherill
University of Wales Swansea

Peter Wegner
Brown University

Jon B. Weissman
University of Minnesota-Twin Cities

Craig E. Wills
Worcester Polytechnic Institute

George Wolberg
City College of New York

Donghui Zhang
Northeastern University

Victor Zue
Massachusetts Institute of Technology

© 2004 by Taylor & Francis Group, LLC

Contents

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

1 Computer Science: The Discipline and its Impact

Allen B. Tucker
Bowdoin College

Peter Wegner
Brown University

1.1 Introduction

The field of computer science has undergone a dramatic evolution in its short 70-year life. As the field has matured, new areas of research and applications have emerged and joined with classical discoveries in a continuous cycle of revitalization and growth.

In the 1930s, fundamental mathematical principles of computing were developed by Turing and Church. Early computers implemented by von Neumann, Wilkes, Eckert, Atanasoff, and others in the 1940s led to the birth of scientific and commercial computing in the 1950s, and to mathematical programming languages like Fortran, commercial languages like COBOL, and artificial-intelligence languages like LISP. In the 1960s the rapid development and consolidation of the subjects of algorithms, data structures, databases, and operating systems formed the core of what we now call traditional computer science; the 1970s saw the emergence of software engineering, structured programming, and object-oriented programming. The emergence of personal computing and networks in the 1980s set the stage for dramatic advances in computer graphics, software technology, and parallelism. The 1990s saw the worldwide emergence of the Internet, both as a medium for academic and scientific exchange and as a vehicle for international commerce and communication.

This Handbook aims to characterize computer science in the new millenium, incorporating the explosive growth of the Internet and the increasing importance of subject areas like human–computer interaction, massively parallel scientific computation, ubiquitous information technology, and other subfields that

© 2004 by Taylor & Francis Group, LLC

would not have appeared in such an encyclopedia even ten years ago. We begin with the following short definition, a variant of the one offered in [Gibbs 1986], which we believe captures the essential nature of "computer science" as we know it today.

> *Computer science* is the study of computational processes and information structures, including their hardware realizations, their linguistic models, and their applications.

The Handbook is organized into eleven sections which correspond to the eleven major subject areas that characterize computer science [ACM/IEEE 2001], and thus provide a useful modern taxonomy for the discipline. The next section presents a brief history of the computing industry and the parallel development of the computer science curriculum. Section 1.3 frames the practice of computer science in terms of four major conceptual paradigms: theory, abstraction, design, and the social context. Section 1.4 identifies the "grand challenges" of computer science research and the subsequent emergence of information technology and cyber-infrastructure that may provide a foundation for addressing these challenges during the next decade and beyond. Section 1.5 summarizes the subject matter in each of the Handbook's eleven sections in some detail.

This Handbook is designed as a professional reference for researchers and practitioners in computer science. Readers interested in exploring specific subject topics may prefer to move directly to the appropriate section of the Handbook — the chapters are organized with minimal interdependence, so that they can be read in any order. To facilitate rapid inquiry, the Handbook contains a Table of Contents and three indexes (Subject, Who's Who, and Key Algorithms and Formulas), providing access to specific topics at various levels of detail.

1.2 Growth of the Discipline and the Profession

The computer industry has experienced tremendous growth and change over the past several decades. The transition that began in the 1980s, from centralized mainframes to a decentralized networked microcomputer–server technology, was accompanied by the rise and decline of major corporations. The old monopolistic, vertically integrated industry epitomized by IBM's comprehensive client services gave way to a highly competitive industry in which the major players changed almost overnight. In 1992 alone, emergent companies like Dell and Microsoft had spectacular profit gains of 77% and 53%. In contrast, traditional companies like IBM and Digital suffered combined record losses of $7.1 billion in the same year [Economist 1993] (although IBM has since recovered significantly). As the 1990s came to an end, this euphoria was replaced by concerns about new monopolistic behaviors, expressed in the form of a massive antitrust lawsuit by the federal government against Microsoft. The rapid decline of the "dot.com" industry at the end of the decade brought what many believe a long-overdue rationality to the technology sector of the economy. However, the exponential decrease in computer cost and increase in power by a factor of two every 18 months, known as Moore's law, shows no signs of abating in the near future, although underlying physical limits will eventually be reached.

Overall, the rapid 18% annual growth rate that the computer industry had enjoyed in earlier decades gave way in the early 1990s to a 6% growth rate, caused in part by a saturation of the personal computer market. Another reason for this slowing of growth is that the performance of computers (speed, storage capacity) has improved at a rate of 30% per year in relation to their cost. Today, it is not unusual for a laptop or hand-held computer to run at hundreds of times the speed and capacity of a typical computer of the early 1990s, and at a fraction of its cost. However, it is not clear whether this slowdown represents a temporary plateau or whether a new round of fundamental technical innovations in areas such as parallel architectures, nanotechnology, or human–computer interaction might generate new spectacular rates of growth in the future.

© 2004 by Taylor & Francis Group, LLC

1.2.1 Curriculum Development

The computer industry's evolution has always been affected by advances in both the theory and the practice of computer science. Changes in theory and practice are simultaneously intertwined with the evolution of the field's undergraduate and graduate curricula, which have served to define the intellectual and methodological framework for the discipline of computer science itself.

The first coherent and widely cited curriculum for computer science was developed in 1968 by the ACM Curriculum Committee on Computer Science [ACM 1968] in response to widespread demand for systematic undergraduate and graduate programs [Rosser 1966]. "Curriculum 68" defined computer science as comprising three main areas: information structures and processes, information processing systems, and methodologies. Curriculum 68 defined computer science as a discipline and provided concrete recommendations and guidance to colleges and universities in developing undergraduate, master's, and doctorate programs to meet the widespread demand for computer scientists in research, education, and industry. Curriculum 68 stood as a robust and exemplary model for degree programs at all levels for the next decade.

In 1978, a new ACM Curriculum Committee on Computer Science developed a revised and updated undergraduate curriculum [ACM 1978]. The "Curriculum 78" report responded to the rapid evolution of the discipline and the practice of computing, and to a demand for a more detailed elaboration of the computer science (as distinguished from the mathematical) elements of the courses that would comprise the core curriculum.

During the next few years, the IEEE Computer Society developed a model curriculum for engineering-oriented undergraduate programs [IEEE-CS 1976], updated and published it in 1983 as a "Model Program in Computer Science and Engineering" [IEEE-CS 1983], and later used it as a foundation for developing a new set of accreditation criteria for undergraduate programs. A simultaneous effort by a different group resulted in the design of a model curriculum for computer science in liberal arts colleges [Gibbs 1986]. This model emphasized science and theory over design and applications, and it was widely adopted by colleges of liberal arts and sciences in the late 1980s and the 1990s.

In 1988, the ACM Task Force on the Core of Computer Science and the IEEE Computer Society [ACM 1988] cooperated in developing a fundamental redefinition of the discipline. Called "Computing as a Discipline," this report aimed to provide a contemporary foundation for undergraduate curriculum design by responding to the changes in computing research, development, and industrial applications in the previous decade. This report also acknowledged some fundamental methodological changes in the field. The notion that "computer science = programming" had become wholly inadequate to encompass the richness of the field. Instead, three different paradigms—called *theory, abstraction,* and *design*—were used to characterize how various groups of computer scientists did their work. These three points of view — those of the theoretical mathematician or scientist (theory), the experimental or applied scientist (abstraction, or modeling), and the engineer (design) — were identified as essential components of research and development across all nine subject areas into which the field was then divided.

"Computing as a Discipline" led to the formation of a joint ACM/IEEE-CS Curriculum Task Force, which developed a more comprehensive model for undergraduate curricula called "Computing Curricula 91" [ACM/IEEE 1991]. Acknowledging that computer science programs had become widely supported in colleges of engineering, arts and sciences, and liberal arts, Curricula 91 proposed a core body of knowledge that undergraduate majors in all of these programs should cover. This core contained sufficient theory, abstraction, and design content that students would become familiar with the three complementary ways of "doing" computer science. It also ensured that students would gain a broad exposure to the nine major subject areas of the discipline, including their social context. A significant laboratory component ensured that students gained significant abstraction and design experience.

In 2001, in response to dramatic changes that had occurred in the discipline during the 1990s, a new ACM/IEEE-CS Task Force developed a revised model curriculum for computer science [ACM/IEEE 2001]. This model updated the list of major subject areas, and we use this updated list to form the organizational basis for this Handbook (see below). This model also acknowledged that the enormous

© 2004 by Taylor & Francis Group, LLC

growth of the computing field had spawned four distinct but overlapping subfields — “computer science,” “computer engineering,” “software engineering,” and “information systems.” While these four subfields share significant knowledge in common, each one also underlies a distinctive academic and professional field. While the computer science dimension is directly addressed by this Handbook, the other three dimensions are addressed to the extent that their subject matter overlaps that of computer science.

1.2.2 Growth of Academic Programs

Fueling the rapid evolution of curricula in computer science during the last three decades was an enormous growth in demand, by industry and academia, for computer science professionals, researchers, and educators at all levels. In response, the number of computer science Ph.D.-granting programs in the U.S. grew from 12 in 1964 to 164 in 2001. During the period 1966 to 2001, the annual number of Bachelor’s degrees awarded in the U.S. grew from 89 to 46,543; Master’s degrees grew from 238 to 19,577; and Ph.D. degrees grew from 19 to 830 [ACM 1968, Bryant 2001].

Figure 1.1 shows the number of bachelor’s and master’s degrees awarded by U.S. colleges and universities in computer science and engineering (CS&E) from 1966 to 2001. The number of Bachelor’s degrees peaked at about 42,000 in 1986, declined to about 24,500 in 1995, and then grew steadily toward its current peak during the past several years. Master’s degree production in computer science has grown steadily without decline throughout this period.

The dramatic growth of BS and MS degrees in the five-year period between 1996 and 2001 parallels the growth and globalization of the economy itself. The more recent falloff in the economy, especially the collapse of the “dot.com” industry, may dampen this growth in the near future. In the long run, future increases in Bachelor’s and Master’s degree production will continue to be linked to expansion of the technology industry, both in the U.S and throughout the world.

Figure 1.2 shows the number of U.S. Ph.D. degrees in computer science during the same 1966 to 2001 period [Bryant 2001]. Production of Ph.D. degrees in computer science grew throughout the early 1990s, fueled by continuing demand from industry for graduate-level talent and from academia to staff growing undergraduate and graduate research programs. However, in recent years, Ph.D. production has fallen off slightly and approached a steady state. Interestingly, this last five years of non-growth at the Ph.D. level is coupled with five years of dramatic growth at the BS and MS levels. This may be partially explained by the unusually high salaries offered in a booming technology sector of the economy, which may have lured some

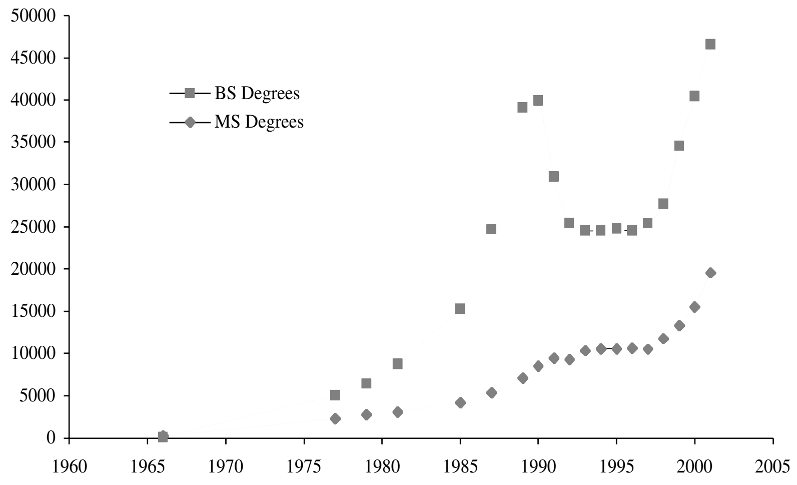


FIGURE 1.1 U.S. bachelor’s and master’s degrees in CS&E.

© 2004 by Taylor & Francis Group, LLC

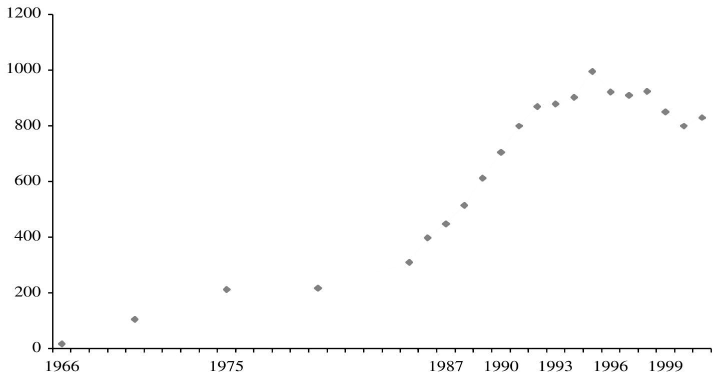


FIGURE 1.2 U.S. Ph.D. degrees in computer science.

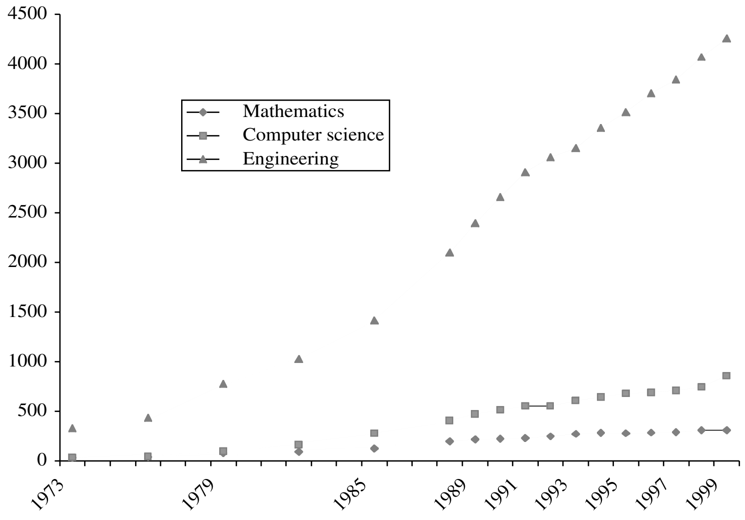


FIGURE 1.3 Academic R&D in computer science and related fields (in millions of dollars).

undergraduates away from immediate pursuit of a Ph.D. The more recent economic slowdown, especially in the technology industry, may help to normalize these trends in the future.

1.2.3 Academic R&D and Industry Growth

University and industrial research and development (R&D) investments in computer science grew rapidly in the period between 1986 and 1999. Figure 1.3 shows that academic research and development in computer science nearly tripled, from \$321 million to \$860 million, during this time period. This growth rate was significantly higher than that of academic R&D in the related fields of engineering and mathematics. During this same period, the overall growth of academic R&D in engineering doubled, while that in mathematics grew by about 50%. About two thirds of the total support for academic R&D comes from federal and state sources, while about 7% comes from industry and the rest comes from the academic institutions themselves [NSF 2002].

Using 1980, 1990, and 2000 U.S. Census data, Figure 1.4 shows recent growth in the number of persons with at least a bachelor's degree who were employed in nonacademic (industry and government) computer

© 2004 by Taylor & Francis Group, LLC

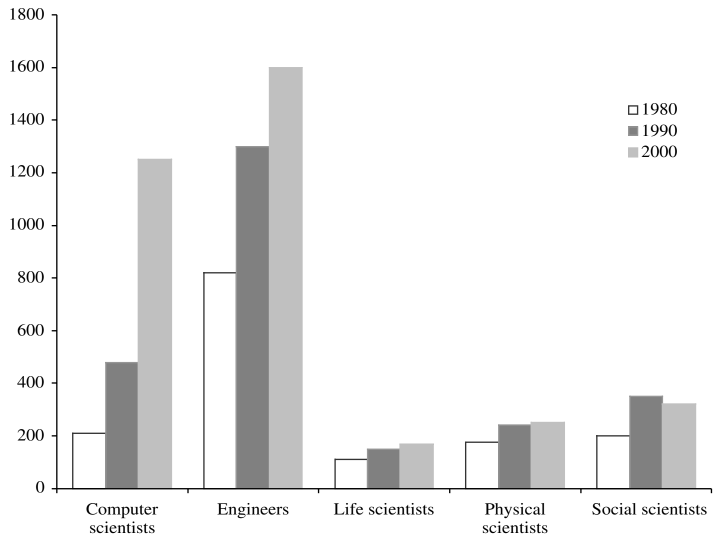


FIGURE 1.4 Nonacademic computer scientists and other professions (thousands).

science positions. Overall, the total number of computer scientists in these positions grew by 600%, from 210,000 in 1980 to 1,250,000 in 2000. Surveys conducted by the Computing Research Association (CRA) suggest that about two thirds of the domestically employed new Ph.D.s accept positions in industry or government, and the remainder accept faculty and postdoctoral research positions in colleges and universities.

CRA surveys also suggest that about one third of the total number of computer science Ph.D.s accept positions abroad [Bryant 2001]. Coupled with this trend is the fact that increasing percentages of U.S. Ph.D.s are earned by non-U.S. citizens. In 2001, about 50% of the total number of Ph.D.s were earned by this group.

Figure 1.4 also provides nonacademic employment data for other science and engineering professions, again considering only persons with bachelor's degrees or higher. Here, we see that all areas grew during this period, with computer science growing at the highest rate. In this group, only engineering had a higher total number of persons in the workforce, at 1.6 million. Overall, the total nonacademic science and engineering workforce grew from 2,136,200 in 1980 to 3,664,000 in 2000, an increase of about 70% [NSF 2001].

1.3 Perspectives in Computer Science

By its very nature, computer science is a multifaceted discipline that can be viewed from at least four different perspectives. Three of the perspectives — *theory, abstraction,* and *design* — underscore the idea that computer scientists in all subject areas can approach their work from different intellectual viewpoints and goals. A fourth perspective — *the social and professional context* — acknowledges that computer science applications directly affect the quality of people's lives, so that computer scientists must understand and confront the social issues that their work uniquely and regularly encounters.

The *theory* of computer science draws from principles of mathematics as well as from the formal methods of the physical, biological, behavioral, and social sciences. It normally includes the use of abstract ideas and methods taken from subfields of mathematics such as logic, algebra, analysis, and statistics. Theory includes the use of various proof and argumentation techniques, like induction and contradiction, to establish properties of formal systems that justify and explain underlying the basic algorithms and data structures used in computational models. Examples include the study of algorithmically unsolvable problems and the study of upper and lower bounds on the complexity of various classes of algorithmic problems. Fields like algorithms and complexity, intelligent systems, computational science, and programming languages have different theoretical models than human–computer interaction or net-centric computing; indeed, all 11 areas covered in this Handbook have underlying theories to a greater or lesser extent.

© 2004 by Taylor & Francis Group, LLC

Abstraction in computer science includes the use of scientific inquiry, modeling, and experimentation to test the validity of hypotheses about computational phenomena. Computer professionals in all 11 areas of the discipline use abstraction as a fundamental tool of inquiry — many would argue that computer science is itself the science of building and examining abstract computational models of reality. Abstraction arises in computer architecture, where the Turing machine serves as an abstract model for complex real computers, and in programming languages, where simple semantic models such as lambda calculus are used as a framework for studying complex languages. Abstraction appears in the design of heuristic and approximation algorithms for problems whose optimal solutions are computationally intractable. It is surely used in graphics and visual computing, where models of three-dimensional objects are constructed mathematically; given properties of lighting, color, and surface texture; and projected in a realistic way on a two-dimensional video screen.

Design is a process that models the essential structure of complex systems as a prelude to their practical implementation. It also encompasses the use of traditional engineering methods, including the classical life-cycle model, to implement efficient and useful computational systems in hardware and software. It includes the use of tools like cost/benefit analysis of alternatives, risk analysis, and fault tolerance that ensure that computing applications are implemented effectively. Design is a central preoccupation of computer architects and software engineers who develop hardware systems and software applications. Design is an especially important activity in computational science, information management, human–computer interaction, operating systems, and net-centric computing.

The *social and professional context* includes many concerns that arise at the computer–human interface, such as liability for hardware and software errors, security and privacy of information in databases and networks (e.g., implications of the Patriot Act), intellectual property issues (e.g., patent and copyright), and equity issues (e.g., universal access to technology and to the profession). All computer scientists must consider the ethical context in which their work occurs and the special responsibilities that attend their work. Chapter 2 discusses these issues, and Appendix B presents the ACM Code of Ethics and Professional Conduct. Several other chapters address topics in which specific social and professional issues come into play. For example, security and privacy issues in databases, operating systems, and networks are discussed in Chapter 60 and Chapter 77. Risks in software are discussed in several chapters of Section XI.

1.4 Broader Horizons: From HPCC to Cyberinfrastructure

In 1989, the Federal Office of Science and Technology announced the "High Performance Computing and Communications Program," or HPCC [OST 1989]. HPCC was designed to encourage universities, research programs, and industry to develop specific capabilities to address the "grand challenges" of the future. To realize these grand challenges would require both fundamental and applied research, including the development of high-performance computing systems with speeds two to three orders of magnitude greater than those of current systems, advanced software technology and algorithms that enable scientists and mathematicians to effectively address these grand challenges, networking to support R&D for a gigabit National Research and Educational Network (NREN), and human resources that expand basic research in all areas relevant to high-performance computing.

The grand challenges themselves were identified in HPCC as those fundamental problems in science and engineering with potentially broad economic, political, or scientific impact that can be advanced by applying high-performance computing technology and that can be solved only by high-level collaboration among computer professionals, scientists, and engineers. A list of grand challenges developed by agencies such as the NSF, DoD, DoE, and NASA in 1989 included:

- Prediction of weather, climate, and global change
- Challenges in materials sciences
- Semiconductor design
- Superconductivity
- Structural biology

© 2004 by Taylor & Francis Group, LLC

- Design of drugs
- Human genome
- Quantum chromodynamics
- Astronomy
- Transportation
- Vehicle dynamics and signature
- Turbulence
- Nuclear fusion
- Combustion systems
- Oil and gas recovery
- Ocean science
- Speech
- Vision
- Undersea surveillance for anti-submarine warfare

The 1992 report entitled "Computing the Future" (CTF) [CSNRCTB 1992], written by a group of leading computer professionals in response to a request by the Computer Science and Technology Board (CSTB), identified the need for computer science to broaden its research agenda and its educational horizons, in part to respond effectively to the grand challenges identified above. The view that the research agenda should be broadened caused concerns among some researchers that this funding and other incentives might overemphasize short-term at the expense of long-term goals. This Handbook reflects the broader view of the discipline in its inclusion of computational science, information management, and human–computer interaction among the major subfields of computer science.

CTF aimed to bridge the gap between suppliers of research in computer science and consumers of research such as industry, the federal government, and funding agencies such as the NSF, DARPA, and DoE. It addressed fundamental challenges to the field and suggested responses that encourage greater interaction between research and computing practice. Its overall recommendations focused on three priorities:

1. To sustain the core effort that creates the theoretical and experimental science base on which applications build
2. To broaden the field to reflect the centrality of computing in science and society
3. To improve education at both the undergraduate and graduate levels

CTF included recommendations to federal policy makers and universities regarding research and education:

- *Recommendations to federal policy makers regarding research:*
  - The High-Performance Computing and Communication (HPCC) program passed by Congress in 1989 [OST 1989] should be fully supported.
  - Application-oriented computer science and engineering research should be strongly encouraged through special funding programs.
- *Recommendations to universities regarding research:*
  - Academic research should broaden its horizons, embracing application-oriented and technology-transfer research as well as core applications.
  - Laboratory research with experimental as well as theoretical content should be supported.
- *Recommendation to federal policy makers regarding education:*
  - Basic and human resources research of HPCC and other areas should be expanded to address educational needs.

© 2004 by Taylor & Francis Group, LLC

- *Recommendations to universities regarding education:*
  - Broaden graduate education to include requirements and incentives to study application areas.
  - Reach out to women and minorities to broaden the talent pool.

Although this report was motivated by the desire to provide a rationale for the HPCC program, its message that computer science must be responsive to the needs of society is much broader. The years since publication of CTF have seen a swing away from pure research toward application-oriented research that is reflected in this edition of the Handbook. However, it remains important to maintain a balance between short-term applications and long-term research in traditional subject areas.

More recently, increased attention has been paid to the emergence of information technology (IT) research as an academic subject area having significant overlap with computer science itself. This development is motivated by several factors, including mainly the emergence of electronic commerce, the shortage of trained IT professionals to fill new jobs in IT, and the continuing need for computing to expand its capability to manage the enormous worldwide growth of electronic information. Several colleges and universities have established new IT degree programs that complement their computer science programs, offering mainly BS and MS degrees in information technology. The National Science Foundation is a strong supporter of IT research, earmarking $190 million in this priority area for FY 2003. This amounts to about 35% of the entire NSF computer science and engineering research budget [NSF 2003a].

The most recent initiative, dubbed "Cyberinfrastructure" [NSF 2003b], provides a comprehensive vision for harnessing the fast-growing technological base to better meet the new challenges and complexities that are shared by a widening community of researchers, professionals, organizations, and citizens who use computers and networks every day. Here are some excerpts from the executive summary for this initiative:

> . . . a new age has dawned in scientific and engineering research, pushed by continuing progress in computing, information, and communication technology, and pulled by the expanding complexity, scope, and scale of today's challenges. The capacity of this technology has crossed thresholds that now make possible a comprehensive "cyberinfrastructure" on which to build new types of scientific and engineering knowledge environments and organizations and to pursue research in new ways and with increased efficacy.
>
> Such environments . . . are required to address national and global priorities, such as understanding global climate change, protecting our natural environment, applying genomics-proteomics to human health, maintaining national security, mastering the world of nanotechnology, and predicting and protecting against natural and human disasters, as well as to address some of our most fundamental intellectual questions such as the formation of the universe and the fundamental character of matter.
>
> This panel's overarching recommendation is that the NSF should establish and lead a large-scale, interagency, and internationally coordinated Advanced Cyberinfrastructure Program (ACP) to create, deploy, and apply cyberinfrastructure in ways that radically empower all scientific and engineering research and allied education. We estimate that sustained new NSF funding of $1 billion per year is needed to achieve critical mass and to leverage the coordinated co-investment from other federal agencies, universities, industry, and international sources necessary to empower a revolution.

It is too early to tell whether the ambitions expressed in this report will provide a new rallying call for science and technology research in the next decade. Achieving them will surely require unprecedented levels of collaboration and funding.

Nevertheless, in response to HPCC and successive initiatives, the two newer subject areas of "computational science" [Stevenson 1994] and "net-centric computing" [ACM/IEEE 2001] have established themselves among the 11 that characterize computer science at this early moment in the 21st century. This Handbook views "computational science" as the application of computational and mathematical models and methods to science, having as a driving force the fundamental interaction between computation and scientific research. For instance, fields like computational astrophysics, computational biology,

© 2004 by Taylor & Francis Group, LLC

and computational chemistry all unify the application of computing in science and engineering with underlying mathematical concepts, algorithms, graphics, and computer architecture. Much of the research and accomplishments of the computational science field is presented in Section III.

Net-centric computing, on the other hand, emphasizes the interactions among people, computers, and the Internet. It affects information technology systems in professional and personal spheres, including the implementation and use of search engines, commercial databases, and digital libraries, along with their risks and human factors. Some of these topics intersect in major ways with those of human–computer interaction, while others fall more directly in the realm of management information systems (MIS). Because MIS is widely viewed as a separate discipline from computer science, this Handbook does not attempt to cover all of MIS. However, it does address many MIS concerns in Section V (human–computer interaction) Section VI (information management), and Section VIII (net-centric computing).

The remaining sections of this Handbook cover relatively traditional areas of computer science — algorithms and complexity, computer architecture, operating systems, programming languages, artificial intelligence, software engineering, and computer graphics. A more careful summary of these sections appears below.

1.5 Organization and Content

In the 1940s, computer science was identified with number crunching, and numerical analysis was considered a central tool. Hardware, logical design, and information theory emerged as important subfields in the early 1950s. Software and programming emerged as important subfields in the mid-1950s and soon dominated hardware as topics of study in computer science. In the 1960s, computer science could be comfortably classified into theory, systems (including hardware and software), and applications. Software engineering emerged as an important subdiscipline in the late 1960s. The 1980 Computer Science and Engineering Research Study (COSERS) [Arden 1980] classified the discipline into nine subfields:

1. Numerical computation
2. Theory of computation
3. Hardware systems
4. Artificial intelligence
5. Programming languages
6. Operating systems
7. Database management systems
8. Software methodology
9. Applications

This Handbook's organization presents computer science in the following 11 sections, which are the subfields defined in [ACM/IEEE 2001].

1. Algorithms and complexity
2. Architecture and organization
3. Computational science
4. Graphics and visual computing
5. Human–computer interaction
6. Information management
7. Intelligent systems
8. Net-centric computing
9. Operating systems
10. Programming languages
11. Software engineering

© 2004 by Taylor & Francis Group, LLC

This overall organization shares much in common with that of the 1980 COSERS study. That is, except for some minor renaming, we can read this list as a broadening of numerical analysis into computational science, and an addition of the new areas of human–computer interaction and graphics. The other areas appear in both classifications with some name changes (theory of computation has become algorithms and complexity, artificial intelligence has become intelligent systems, applications has become net-centric computing, hardware systems has evolved into architecture and networks, and database has evolved into information management). The overall similarity between the two lists suggests that the discipline of computer science has stabilized in the past 25 years.

However, although this high-level classification has remained stable, the content of each area has evolved dramatically. We examine below the scope of each area individually, along with the topics in each area that are emphasized in this Handbook.

1.5.1 Algorithms and Complexity

The subfield of algorithms and complexity is interpreted broadly to include core topics in the theory of computation as well as data structures and practical algorithm techniques. Its chapters provide a comprehensive overview that spans both theoretical and applied topics in the analysis of algorithms. Chapter 3 provides an overview of techniques of algorithm design like divide and conquer, dynamic programming, recurrence relations, and greedy heuristics, while Chapter 4 covers data structures both descriptively and in terms of their space–time complexity.

Chapter 5 examines topics in complexity like P vs. NP and NP-completeness, while Chapter 6 introduces the fundamental concepts of computability and undecidability and formal models such as Turing machines. Graph and network algorithms are treated in Chapter 7, and algebraic algorithms are the subject of Chapter 8.

The wide range of algorithm applications is presented in Chapter 9 through Chapter 15. Chapter 9 covers cryptographic algorithms, which have recently become very important in operating systems and network security applications. Chapter 10 covers algorithms for parallel computer architectures, Chapter 11 discusses algorithms for computational geometry, while Chapter 12 introduces the rich subject of randomized algorithms. Pattern matching and text compression algorithms are examined in Chapter 13, and genetic algorithms and their use in the biological sciences are introduced in Chapter 14. Chapter 15 concludes this section with a treatment of combinatorial optimization.

1.5.2 Architecture

Computer architecture is the design of efficient and effective computer hardware at all levels, from the most fundamental concerns of logic and circuit design to the broadest concerns of parallelism and high-performance computing. The chapters in Section II span these levels, providing a sampling of the principles, accomplishments, and challenges faced by modern computer architects.

Chapter 16 introduces the fundamentals of logic design components, including elementary circuits, Karnaugh maps, programmable array logic, circuit complexity and minimization issues, arithmetic processes, and speedup techniques. Chapter 17 focuses on processor design, including the fetch/execute instruction cycle, stack machines, CISC vs. RISC, and pipelining. The principles of memory design are covered in Chapter 18, while the architecture of buses and other interfaces is addressed in Chapter 19. Chapter 20 discusses the characteristics of input and output devices like the keyboard, display screens, and multimedia audio devices. Chapter 21 focuses on the architecture of secondary storage devices, especially disks.

Chapter 22 concerns the design of effective and efficient computer arithmetic units, while Chapter 23 extends the design horizon by considering various models of parallel architectures that enhance the performance of traditional serial architectures. Chapter 24 focuses on the relationship between computer architecture and networks, while Chapter 25 covers the strategies employed in the design of fault-tolerant and reliable computers.

© 2004 by Taylor & Francis Group, LLC

1.5.3 Computational Science

The area of computational science unites computation, experimentation, and theory as three fundamental modes of scientific discovery. It uses scientific visualization, made possible by simulation and modeling, as a window into the analysis of physical, chemical, and biological phenomena and processes, providing a virtual microscope for inquiry at an unprecedented level of detail.

This section focuses on the challenges and opportunities offered by very high-speed clusters of computers and sophisticated graphical interfaces that aid scientific research and engineering design. Chapter 26 introduces the section by presenting the fundamental subjects of computational geometry and grid generation. The design of graphical models for scientific visualization of complex physical and biological phenomena is the subject of Chapter 27.

Each of the remaining chapters in this section covers the computational challenges and discoveries in a specific scientific or engineering field. Chapter 28 presents the computational aspects of structural mechanics, Chapter 29 summarizes progress in the area of computational electromagnetics, and Chapter 30 addresses computational modeling in the field of fluid dynamics. Chapter 31 addresses the grand challenge of computational ocean modeling. Computational chemistry is the subject of Chapter 32, while Chapter 33 addresses the computational dimensions of astrophysics. Chapter 34 closes this section with a discussion of the dramatic recent progress in computational biology.

1.5.4 Graphics and Visual Computing

Computer graphics is the study and realization of complex processes for representing physical and conceptual objects visually on a computer screen. These processes include the internal modeling of objects, rendering, projection, and motion. An overview of these processes and their interaction is presented in Chapter 35.

Fundamental to all graphics applications are the processes of modeling and rendering. Modeling is the design of an effective and efficient internal representation for geometric objects, which is the subject of Chapter 36 and Chapter 37. Rendering, the process of representing the objects in a three-dimensional scene on a two-dimensional screen, is discussed in Chapter 38. Among its special challenges are the elimination of hidden surfaces and the modeling of color, illumination, and shading.

The reconstruction of scanned and digitally photographed images is another important area of computer graphics. Sampling, filtering, reconstruction, and anti-aliasing are the focus of Chapter 39. The representation and control of motion, or animation, is another complex and important area of computer graphics. Its special challenges are presented in Chapter 40.

Chapter 41 discusses volume datasets, and Chapter 42 looks at the emerging field of virtual reality and its particular challenges for computer graphics. Chapter 43 concludes this section with a discussion of progress in the computer simulation of vision.

1.5.5 Human–Computer Interaction

This area, the study of how humans and computers interact, has the goal of improving the quality of such interaction and the effectiveness of those who use technology in the workplace. This includes the conception, design, implementation, risk analysis, and effects of user interfaces and tools on the people who use them.

Modeling the organizational environments in which technology users work is the subject of Chapter 44. Usability engineering is the focus of Chapter 45, while Chapter 46 covers task analysis and the design of functionality at the user interface. The influence of psychological preferences of users and programmers and the integration of these preferences into the design process is the subject of Chapter 47.

Specific devices, tools, and techniques for effective user-interface design form the basis for the next few chapters in this section. Lower-level concerns for the design of interface software technology are addressed in Chapter 48. The special challenges of integrating multimedia with user interaction are presented in Chapter 49. Computer-supported collaboration is the subject of Chapter 50, and the impact of international standards on the user interface design process is the main concern of Chapter 51.

© 2004 by Taylor & Francis Group, LLC

1.5.6 Information Management

The subject area of information management addresses the general problem of storing large amounts of data in such a way that they are reliable, up-to-date, accessible, and efficiently retrieved. This problem is prominent in a wide range of applications in industry, government, and academic research. Availability of such data on the Internet and in forms other than text (e.g., CD, audio, and video) makes this problem increasingly complex.

At the foundation are the fundamental data models (relational, hierarchical, and object-oriented) discussed in Chapter 52. The conceptual, logical, and physical levels of designing a database for high performance in a particular application domain are discussed in Chapter 53.

A number of basic issues surround the effective design of database models and systems. These include choosing appropriate access methods (Chapter 54), optimizing database queries (Chapter 55), controlling concurrency (Chapter 56), and processing transactions (Chapter 57).

The design of databases for distributed and parallel systems is discussed in Chapter 58, while the design of hypertext and multimedia databases is the subject of Chapter 59. The contemporary issue of database security and privacy protection, in both stand-alone and networked environments, is the subject of Chapter 60.

1.5.7 Intelligent Systems

The field of intelligent systems, often called artificial intelligence (AI), studies systems that simulate human rational behavior in all its forms. Current efforts are aimed at constructing computational mechanisms that process visual data, understand speech and written language, control robot motion, and model physical and cognitive processes. Robotics is a complex field, drawing heavily from AI as well as other areas of science and engineering.

Artificial intelligence research uses a variety of distinct algorithms and models. These include fuzzy, temporal, and other logics, as described in Chapter 61. The related idea of qualitative modeling is discussed in Chapter 62, while the use of complex specialized search techniques that address the combinatorial explosion of alternatives in AI problems is the subject of Chapter 63. Chapter 64 addresses issues related to the mechanical understanding of spoken language.

Intelligent systems also include techniques for automated learning and planning. The use of decision trees and neural networks in learning and other areas is the subject of Chapter 65 and Chapter 66. Chapter 67 presents the rationale and uses of planning and scheduling models, while Chapter 68 contains a discussion of deductive learning. Chapter 69 addresses the challenges of modeling from the viewpoint of cognitive science, while Chapter 70 treats the challenges of decision making under uncertainty.

Chapter 71 concludes this section with a discussion of the principles and major results in the field of robotics: the design of effective devices that simulate mechanical, sensory, and intellectual functions of humans in specific task domains such as navigation and planning.

1.5.8 Net-Centric Computing

Extending system functionality across a networked environment has added an entirely new dimension to the traditional study and practice of computer science. Chapter 72 presents an overview of network organization and topologies, while Chapter 73 describes network routing protocols. Basic issues in network management are addressed in Chapter 74.

The special challenges of information retrieval and data mining from large databases and the Internet are addressed in Chapter 75. The important topic of data compression for internetwork transmission and archiving is covered in Chapter 76.

Modern computer networks, especially the Internet, must ensure system integrity in the event of inappropriate access, unexpected malfunction and breakdown, and violations of data and system security or individual privacy. Chapter 77 addresses the principles surrounding these security and privacy issues. A discussion of some specific malicious software and hacking events appears in Chapter 78. This section concludes with Chapter 79, which discusses protocols for user authentication, access control, and intrusion detection.

© 2004 by Taylor & Francis Group, LLC

1.5.9 Operating Systems

An operating system is the software interface between the computer and its applications. This section covers operating system analysis, design, and performance, along with the special challenges for operating systems in a networked environment. Chapter 80 briefly traces the historical development of operating systems and introduces the fundamental terminology, including process scheduling, memory management, synchronization, I/O management, and distributed systems.

The "process" is a key unit of abstraction in operating system design. Chapter 81 discusses the dynamics of processes and threads. Strategies for process and device scheduling are presented in Chapter 82. The special requirements for operating systems in real-time and embedded system environments are treated in Chapter 83. Algorithms and techniques for process synchronization and interprocess communication are the subject of Chapter 84.

Memory and input/output device management is also a central concern of operating systems. Chapter 85 discusses the concept of virtual memory, from its early incarnations to its uses in present-day systems and networks. The different models and access methods for secondary storage and filesystems are covered in Chapter 86.

The influence of networked environments on the design of distributed operating systems is considered in Chapter 87. Distributed and multiprocessor scheduling are the focus in Chapter 88, while distributed file and memory systems are discussed in Chapter 89.

1.5.10 Programming Languages

This section examines the design of programming languages, including their paradigms, mechanisms for compiling and runtime management, and theoretical models, type systems, and semantics. Overall, this section provides a good balance between considerations of programming paradigms, implementation issues, and theoretical models.

Chapter 90 considers traditional language and implementation questions for imperative programming languages such as Fortran, C, and Ada. Chapter 91 examines object-oriented concepts such as classes, inheritance, encapsulation, and polymorphism, while Chapter 92 presents the view of functional programming, including lazy and eager evaluation. Chapter 93 considers declarative programming in the logic/constraint programming paradigm, while Chapter 94 covers the design and use of special purpose scripting languages. Chapter 95 considers the emergent paradigm of event-driven programming, while Chapter 96 covers issues regarding concurrent, distributed, and parallel programming models.

Type systems are the subject of Chapter 97, while Chapter 98 covers programming language semantics. Compilers and interpreters for sequential languages are considered in Chapter 99, while the issues surrounding runtime environments and memory management for compilers and interpreters are addressed in Chapter 100.

Brief summaries of the main features and applications of several contemporary languages appear in Appendix D, along with links to Web sites for more detailed information on these languages.

1.5.11 Software Engineering

The section on software engineering examines formal specification, design, verification and testing, project management, and other aspects of the software process. Chapter 101 introduces general software qualities such as maintainability, portability, and reuse that are needed for high-quality software systems, while Chapter 109 covers the general topic of software architecture.

Chapter 102 reviews specific models of the software life cycle such as the waterfall and spiral models. Chapter 106 considers a more formal treatment of software models, including formal specification languages.

Chapter 103 deals with the traditional design process, featuring a case study in top-down functional design. Chapter 104 considers the complementary strategy of object-oriented software design. Chapter 105

© 2004 by Taylor & Francis Group, LLC

treats the subject of validation and testing, including risk and reliability issues. Chapter 107 deals with the use of rigorous techniques such as formal verification for quality assurance.

Chapter 108 considers techniques of software project management, including team formation, project scheduling, and evaluation, while Chapter 110 concludes this section with a treatment of specialized system development.

1.6 Conclusion

In 2002, the ACM celebrated its 55th anniversary. These five decades of computer science are characterized by dramatic growth and evolution. While it is safe to reaffirm that the field has attained a certain level of maturity, we surely cannot assume that it will remain unchanged for very long. Already, conferences are calling for new visions that will enable the discipline to continue its rapid evolution in response to the world's continuing demand for new technology and innovation.

This Handbook is designed to convey the modern spirit, accomplishments, and direction of computer science as we see it in 2003. It interweaves theory with practice, highlighting "best practices" in the field as well as emerging research directions. It provides today's answers to computational questions posed by professionals and researchers working in all 11 subject areas. Finally, it identifies key professional and social issues that lie at the intersection of the technical aspects of computer science and the people whose lives are impacted by such technology.

The future holds great promise for the next generations of computer scientists. These people will solve problems that have only recently been conceived, such as those suggested by the HPCC as "grand challenges." To address these problems in a way that benefits the world's citizenry will require substantial energy, commitment, and real investment on the part of institutions and professionals throughout the field. The challenges are great, and the solutions are not likely to be obvious.

References

ACM Curriculum Committee on Computer Science 1968. Curriculum 68: recommendations for the undergraduate program in computer science. *Commun. ACM*, 11(3):151–197, March.

ACM Curriculum Committee on Computer Science 1978. Curriculum 78: recommendations for the undergraduate program in computer science. *Commun. ACM*, 22(3):147–166, March.

ACM Task Force on the Core of Computer Science: Denning, P., Comer, D., Gries, D., Mulder, M., Tucker, A., and Young, P., 1988. *Computing as a Discipline*. Abridged version, *Commun. ACM*, Jan. 1989.

ACM/IEEE-CS Joint Curriculum Task Force. Computing Curricula 1991. ACM Press. Abridged version, *Commun. ACM*, June 1991, and *IEEE Comput.* Nov. 1991.

ACM/IEEE-CS Joint Task Force. Computing Curricula 2001: Computer Science Volume. ACM and IEEE Computer Society, December 2001, (http://www.acm.org/sigcse/cc2001).

Arden, B., Ed., 1980. *What Can be Automated?* Computer Science and Engineering Research (COSERS) Study. MIT Press, Boston, MA.

Bryant, R.E. and M.Y. Vardi, 2001. 2000–2001 Taulbee Survey: Hope for More Balance in Supply and Demand. Computing Research Assoc (http://www.cra.org).

CSNRCTB 1992. Computer Science and National Research Council Telecommunications Board. *Computing the Future: A Broader Agenda for Computer Science and Engineering*. National Academy Press, Washington, D.C.

Economist 1993. The computer industry: reboot system and start again. *Economist*, Feb. 27.

Gibbs, N. and A. Tucker 1986. A Model Curriculum for a Liberal Arts Degree in Computer Science. *Communications of the ACM*, March.

IEEE-CS 1976. Education Committee of the IEEE Computer Society. *A Curriculum in Computer Science and Engineering*. IEEE Pub. EH0119-8, Jan. 1977.

© 2004 by Taylor & Francis Group, LLC

IEEE-CS 1983. Educational Activities Board. *The 1983 Model Program in Computer Science and Engineering.* Tech. Rep. 932. Computer Society of the IEEE, December.

NSF 2002. National Science Foundation. *Science and Engineering Indicators* (Vol. I and II), National Science Board, Arlington, VA.

NSF 2003a. National Science Foundation. *Budget Overview FY 2003* (http://www.nsf.gov/bfa/bud/fy2003/overview.htm).

NSF 2003b. National Science Foundation. *Revolutionizing Science and Engineering through Cyberinfrastructure*, report of the NSF Blue-Ribbon Advisory Panel on Cyberinfrastructure, January.

OST 1989. Office of Science and Technology. *The Federal High Performance Computing and Communication Program.* Executive Office of the President, Washington, D.C.

Rosser, J.B. et al. 1966. *Digital Computer Needs in Universities and Colleges.* Publ. 1233, National Academy of Sciences, National Research Council, Washington, D.C.

Stevenson, D.E. 1994. Science, computational science, and computer science. *Commun. ACM*, December.

© 2004 by Taylor & Francis Group, LLC

2 Ethical Issues for Computer Scientists

Deborah G. Johnson
University of Virginia

Keith W. Miller
University of Illinois

2.1 Introduction: Why a Chapter on Ethical Issues?

Computers have had a powerful impact on our world and are destined to shape our future. This observation, now commonplace, is the starting point for any discussion of professionalism and ethics in computing. The work of computer scientists and engineers is part of the social, political, economic, and cultural world in which we live, and it affects many aspects of that world. Professionals who work with computers have special knowledge. That knowledge, when combined with computers, has significant power to change people's lives — by changing socio-technical systems; social, political and economic institutions; and social relationships.

In this chapter, we provide a perspective on the role of computer and engineering professionals and we examine the relationships and responsibilities that go with having and using computing expertise. In addition to the topic of professional ethics, we briefly discuss several of the social–ethical issues created or exacerbated by the increasing power of computers and information technology: privacy, property, risk and reliability, and globalization.

Computers, digital data, and telecommunications have changed work, travel, education, business, entertainment, government, and manufacturing. For example, work now increasingly involves sitting in front of a computer screen and using a keyboard to make things happen in a manufacturing process or to keep track of records. In the past, these same tasks would have involved physically lifting, pushing, and twisting or using pens, paper, and file cabinets. Changes such as these in the way we do things have, in turn, fundamentally changed who we are as individuals, communities, and nations. Some would argue, for example, that new kinds of communities (e.g., cyberspace on the Internet) are forming, individuals are developing new types of personal identities, and new forms of authority and control are taking hold as a result of this evolving technology.

© 2004 by Taylor & Francis Group, LLC

Computer technology is shaped by social–cultural concepts, laws, the economy, and politics. These same concepts, laws, and institutions have been pressured, challenged, and modified by computer technology. Technological advances can antiquate laws, concepts, and traditions, compelling us to reinterpret and create new laws, concepts, and moral notions. Our attitudes about work and play, our values, and our laws and customs are deeply involved in technological change.

When it comes to the social–ethical issues surrounding computers, some have argued that the issues are not unique. All of the ethical issues raised by computer technology can, it is said, be classified and worked out using traditional *moral* concepts, distinctions, and theories. There is nothing new here in the sense that we can understand the new issues using traditional moral concepts, such as privacy, property, and responsibility, and traditional moral values, such as individual freedom, autonomy, accountability, and community. These concepts and values predate computers; hence, it would seem there is nothing unique about *computer ethics*.

On the other hand, those who argue for the uniqueness of the issues point to the fundamental ways in which computers have changed so many human activities, such as manufacturing, record keeping, banking, international trade, education, and communication. Taken together, these changes are so radical, it is claimed, that traditional moral concepts, distinctions, and theories, if not abandoned, must be significantly reinterpreted and extended. For example, they must be extended to computer-mediated relationships, computer software, computer art, datamining, virtual systems, and so on.

The uniqueness of the ethical issues surrounding computers can be argued in a variety of ways. Computer technology makes possible a scale of activities not possible before. This includes a larger scale of record keeping of personal information, as well as larger-scale calculations which, in turn, allow us to build and do things not possible before, such as undertaking space travel and operating a global communication system. Among other things, the increased scale means finer-grained personal information collection and more precise data matching and datamining. In addition to scale, computer technology has involved the creation of new kinds of entities for which no rules initially existed: entities such as computer files, computer programs, the Internet, Web browsers, cookies, and so on. The uniqueness argument can also be made in terms of the power and pervasiveness of computer technology. Computers and information technology seem to be bringing about a magnitude of change comparable to that which took place during the Industrial Revolution, transforming our social, economic, and political institutions; our understanding of what it means to be human; and the distribution of power in the world. Hence, it would seem that the issues are at least special, if not unique.

In this chapter, we will take an approach that synthesizes these two views of computer ethics by assuming that the analysis of computer ethical issues involves both working on something new and drawing on something old. We will view issues in computer ethics as new species of older ethical problems [Johnson 1994], such that the issues can be understood using traditional moral concepts such as autonomy, privacy, property, and responsibility, while at the same time recognizing that these concepts may have to be extended to what is new and special about computers and the situations they create.

Most ethical issues arising around computers occur in contexts in which there are already social, ethical, and legal norms. In these contexts, often there are implicit, if not formal (legal), rules about how individuals are to behave; there are familiar practices, social meanings, interdependencies, and so on. In this respect, the issues are not new or unique, or at least cannot be resolved without understanding the prevailing context, meanings, and values. At the same time, the situation may have special features because of the involvement of computers — features that have not yet been addressed by prevailing norms. These features can make a moral difference. For example, although property rights and even intellectual property rights had been worked out long before the creation of software, when software first appeared, it raised a new form of property issue. Should the arrangement of icons appearing on the screen of a user interface be ownable? Is there anything intrinsically wrong in copying software? Software has features that make the distinction between idea and expression (a distinction at the core of copyright law) almost incoherent. As well, software has features that make standard intellectual property laws difficult to enforce. Hence, questions about what should be owned when it comes to software and how to evaluate violations of software ownership rights are not new in the sense that they are property rights issues, but they are new

© 2004 by Taylor & Francis Group, LLC

in the sense that nothing with the characteristics of software had been addressed before. We have, then, a new species of traditional property rights.

Similarly, although our understanding of rights and responsibilities in the employer–employee relationship has been evolving for centuries, never before have employers had the capacity to monitor their workers electronically, keeping track of every keystroke, and recording and reviewing all work done by an employee (covertly or with prior consent). When we evaluate this new monitoring capability and ask whether employers should use it, we are working on an issue that has never arisen before, although many other issues involving employer–employee rights have. We must address a new species of the tension between employer–employee rights and interests.

The social–ethical issues posed by computer technology are significant in their own right, but they are of special interest here because computer and engineering professionals bear responsibility for this technology. It is of critical importance that they understand the social change brought about by their work and the difficult social–ethical issues posed. Just as some have argued that the social–ethical issues posed by computer technology are not unique, some have argued that the issues of professional ethics surrounding computers are not unique. We propose, in parallel with our previous genus–species account, that the professional ethics issues arising for computer scientists and engineers are species of generic issues of professional ethics. All professionals have responsibilities to their employers, clients, co-professionals, and the public. Managing these types of responsibilities poses a challenge in all professions. Moreover, all professionals bear some responsibility for the impact of their work. In this sense, the professional ethics issues arising for computer scientists and engineers are generally similar to those in other professions. Nevertheless, it is also true to say that the issues arise in unique ways for computer scientists and engineers because of the special features of computer technology.

In what follows, we discuss ethics in general, professional ethics, and finally, the ethical issues surrounding computer and information technology.

2.2 Ethics in General

Rigorous study of ethics has traditionally been the purview of philosophers and scholars of religious studies. Scholars of ethics have developed a variety of ethical theories with several tasks in mind:

To explain and justify the idea of morality and prevailing moral notions
To critique ordinary moral beliefs
To assist in rational, ethical decision making

Our aim in this chapter is not to propose, defend, or attack any particular ethical theory. Rather, we offer brief descriptions of three major and influential ethical theories to illustrate the nature of ethical analysis. We also include a decision-making method that combines elements of each theory.

Ethical analysis involves giving reasons for moral claims and commitments. It is not just a matter of articulating intuitions. When the reasons given for a claim are developed into a moral theory, the theory can be incorporated into techniques for improved technical decision making. The three ethical theories described in this section represent three traditions in ethical analysis and problem solving. The account we give is not exhaustive, nor is our description of the three theories any more than a brief introduction. The three traditions are utilitarianism, deontology, and social contract theory.

2.2.1 Utilitarianism

Utilitarianism has greatly influenced 20th-century thinking, especially insofar as it influenced the development of cost–benefit analysis. According to utilitarianism, we should make decisions about what to do by focusing on the consequences of actions and policies; we should choose actions and policies that bring about the best consequences. Ethical rules are derived from their usefulness (their utility) in bringing about happiness. In this way, utilitarianism offers a seemingly simple moral principle to determine what to do

© 2004 by Taylor & Francis Group, LLC

in a given situation: everyone ought to act so as to bring about the greatest amount of happiness for the greatest number of people.

According to utilitarianism, happiness is the only value that can serve as a foundational base for ethics. Because happiness is the ultimate good, morality must be based on creating as much of this good as possible. The utilitarian principle provides a decision procedure. When you want to know what to do, the right action is the alternative that produces the most overall net happiness (happiness-producing consequences minus unhappiness-producing consequences). The right action may be one that brings about some unhappiness, but that is justified if the action also brings about enough happiness to counterbalance the unhappiness or if the action brings about the least unhappiness of all possible alternatives.

Utilitarianism should not be confused with egoism. Egoism is a theory claiming that one should act so as to bring about the most good consequences *for oneself*. Utilitarianism does not say that you should maximize your own good. Rather, total happiness in the world is what is at issue; when you evaluate your alternatives, you must ask about their effects on the happiness of everyone. It may turn out to be right for you to do something that will diminish your own happiness because it will bring about an increase in overall happiness.

The emphasis on consequences found in utilitarianism is very much a part of personal and policy decision making in our society, in particular as a framework for law and public policy. Cost–benefit and risk–benefit analysis are, for example, consequentialist in character.

Utilitarians do not all agree on the details of utilitarianism; there are different kinds of utilitarianism. One issue is whether the focus should be on *rules* of behavior or individual *acts*. Utilitarians have recognized that it would be counter to overall happiness if each one of us had to calculate at every moment what the consequences of every one of our actions would be. Sometimes we must act quickly, and often the consequences are difficult or impossible to foresee. Thus, there is a need for general rules to guide our actions in ordinary situations. Hence, *rule-utilitarians* argue that we ought to adopt rules that, if followed by everyone, would, in general and in the long run, maximize happiness. *Act-utilitarians*, on the other hand, put the emphasis on judging individual actions rather than creating rules.

Both rule-utilitarians and act-utilitarians, nevertheless, share an emphasis on consequences; deontological theories do not share this emphasis.

2.2.2 Deontological Theories

Deontological theories can be understood as a response to important criticisms of utilitarian theories. A standard criticism is that utilitarianism seems to lead to conclusions that are incompatible with our most strongly held moral intuitions. Utilitarianism seems, for example, open to the possibility of justifying enormous burdens on some individuals for the sake of others. To be sure, every person counts equally; no one person's happiness or unhappiness is more important than any other person's. However, because utilitarians are concerned with the total amount of happiness, we can imagine situations where great overall happiness would result from sacrificing the happiness of a few. Suppose, for example, that having a small number of slaves would create great happiness for large numbers of people; or suppose we kill one healthy person and use his or her body parts to save ten people in need of transplants.

Critics of utilitarianism say that if utilitarianism justifies such practices, then the theory must be wrong. Utilitarians have a defense, arguing that such practices could not be justified in utilitarianism because of the long-term consequences. Such practices would produce so much fear that the happiness temporarily created would never counterbalance the unhappiness of everyone living in fear that they might be sacrificed for the sake of overall happiness.

We need not debate utilitarianism here. The point is that deontologists find utilitarianism problematic because it puts the emphasis on the consequences of an act rather than on the quality of the act itself. Deontological theories claim that the internal character of the act is what is important. The rightness or wrongness of an action depends on the principles inherent in the action. If an action is done from a sense of duty, and if the principle of the action can be universalized, then the action is right. For example, if I tell the truth because it is convenient for me to do so or because I fear the consequences of getting caught in a

© 2004 by Taylor & Francis Group, LLC

lie, my action is not worthy. A worthy action is an action that is done from duty, which involves respecting other people and recognizing them as ends in themselves, not as means to some good effect.

According to deontologists, utilitarianism is wrong because it treats individuals as means to an end (maximum happiness). For deontologists, what grounds morality is not happiness, but human beings as rational agents. Human beings are capable of reasoning about what they want to do. The laws of nature determine most activities: plants grow toward the sun, water boils at a certain temperature, and objects accelerate at a constant rate in a vacuum. Human action is different in that it is self-determining; humans initiate action after thinking, reasoning, and deciding. The human capacity for rational decisions makes morality possible, and it grounds deontological theory. Because each human being has this capacity, each human being must be treated accordingly — with respect. No one else can make our moral choices for us, and each of us must recognize this capacity in others.

Although deontological theories can be formulated in a number of ways, one formulation is particularly important: Immanuel Kant's categorical imperative [Kant 1785]. There are three versions of it, and the second version goes as follows: *Never treat another human being merely as a means but always as an end*. It is important to note the *merely* in the categorical imperative. Deontologists do not insist that we never use another person; only that we never *merely* use them. For example, if I own a company and hire employees to work in my company, I might be thought of as using those employees as a means to my end (i.e., the success of my business). This, however, is not wrong if the employees agree to work for me and if I pay them a fair wage. I thereby respect their ability to choose for themselves, and I respect the value of their labor. What would be wrong would be to take them as slaves and make them work for me, or to pay them so little that they must borrow from me and remain always in my debt. This would show disregard for the value of each person as a freely choosing, rationally valuing, efficacious person.

2.2.3 Social Contract Theories

A third tradition in ethics thinks of ethics on the model of a social contract. There are many different social contract theories, and some, at least, are based on a deontological principle. Individuals are rational free agents; hence, it is immoral to exert undue power over them, that is, to coerce them. Government and society are problematic insofar as they seem to force individuals to obey rules, apparently treating individuals as means to social good. Social contract theories get around this problem by claiming that morality (and government policy) is, in effect, the outcome of rational agents agreeing to social rules. In agreeing to live by certain rules, we make a contract. Morality and government are not, then, systems imposed on individuals; they do not exactly involve coercion. Rather, they are systems created by freely choosing individuals (or they are institutions that rational individuals would choose if given the opportunity).

Philosophers such as Rousseau, Locke, Hobbes, and more recently Rawls [1971] are generally considered social contract theorists. They differ in how they get to the social contract and what it implies. For our purposes, however, the key idea is that principles and rules guiding behavior may be derived from identifying what it is that rational (even self-interested) individuals would agree to in making a social contract. Such principles and rules are the basis of a shared morality. For example, it would be rational for me to agree to live by rules that forbid killing and lying. Even though such rules constrain me, they also give me some degree of protection: if they are followed, I will not be killed or lied to.

It is important to note, however, that social contract theory cannot be used simply by asking what rules you would agree to *now*. Most theorists recognize that what you would agree to now is influenced by your present position in society. Most individuals would opt for rules that would benefit their particular situation and characteristics. Hence, most social contract theorists insist that the principles or rules of the social contract must be derived by assuming certain things about human nature or the human condition. Rawls, for example, insists that we imagine ourselves behind a *veil of ignorance*. We are not allowed to know important features about ourselves (e.g., what talents we have, what race or gender we are), for if we know these things, we will not agree to just rules, but only to rules that will maximize our self-interest. Justice consists of the rules we would agree to when we do not know who we are, for we would want rules that would give us a fair situation no matter where we ended up in the society.

© 2004 by Taylor & Francis Group, LLC

2.2.4 A Paramedic Method for Computer Ethics

Drawing on elements of the three theories described, Collins and Miller [1992] have proposed a decision-assisting method, called the *paramedic method for computer ethics*. This is *not* an algorithm for solving ethical problems; it is not nearly detailed or objective enough for that designation. It is merely a guideline for an organized approach to ethical problem solving.

Assume that a computer professional is faced with a decision that involves human values in a significant way. There may already be some obvious alternatives, and there also may be creative solutions not yet discovered. The paramedic method is designed to help the professional to analyze alternative actions and to encourage the development of creative solutions. To illustrate the method, suppose you are in a tight spot and do not know exactly what the right thing to do is. The method proceeds as follows:

1. Identify alternative actions; list the few alternatives that seem most promising. If an action requires a long description, summarize it as a title with just a few words. Call the alternative actions A_1, $A_2, \ldots, A_a$. No more than five actions should be analyzed at a time.
2. Identify people, groups of people, or organizations that will be affected by each of the alternative decision-actions. Again, hold down the number of entities to the five or six that are affected most. Label the people $P_1, P_2, \ldots, P_p$.
3. Make a table with the horizontal rows labeled by the identified people and the vertical columns labeled with the identified actions. We call such a table a *P × A table*. Make two copies of the P × A table; label one the *opportunities* table and the other the *vulnerabilities* table. In the opportunities table, list in each interior cell of the table at entry $[x, y]$ the possible good that is likely to happen to person x if action y is taken. Similarly, in the vulnerability table, at position $[x, y]$ list all of the things that are likely to happen badly for x if the action y is taken. These two graphs represent benefit–cost calculations for a consequentialist, utilitarian analysis.
4. Make a new table with the set of persons marking both the columns and the rows (a P × P table). In each cell $[x, y]$ name any responsibilities or duties that x owes y in this situation. (The cells on the diagonal $[x, x]$ are important; they list things one owes oneself.) Now, make copies of this table, labeling one copy for each of the alternative actions being considered. Work through each cell $[x, y]$ of each table and place a $+$ next to a duty if the action for that table is likely to fulfill the duty x owes y; mark the duty with a $-$ if the action is unlikely to fulfill that duty; mark the duty with a $+/-$ if the action partially fulfills it and partially does not; and mark the duty with a ? if the action is irrelevant to the duty or if it is impossible to predict whether or not the duty will be fulfilled. (Few cells generally fall into this last category.)
5. Review the tables from steps 3 and 4. Envision a meeting of all of the parties (or one representative from each of the groups) in which no one knows which role they will take or when they will leave the negotiation. Which alternative do you think such a group would adopt, if any? Do you think such a group could discover a new alternative, perhaps combining the best elements of the previously listed actions? If this thought experiment produces a new alternative, expand the P × A tables from step 3 to include the new alternative action, make a new copy of the P × P table in step 4, and do the $+$ and $-$ marking for the new table.
6. If any one of the alternatives seems to be clearly preferred (i.e., it has high opportunity and low vulnerability for all parties and tends to fulfill all the duties in the P × P table), then that becomes the recommended decision. If no one alternative action stands out, the professionals can examine trade-offs using the charts or can iteratively attempt step 5 (perhaps with outside consultations) until an acceptable alternative is generated.

Using the paramedic method can be time consuming, and it does not eliminate the need for judgment. But it can help organize and focus analysis as an individual or a group works through the details of a situation to arrive at a decision.

© 2004 by Taylor & Francis Group, LLC

2.2.5 Easy and Hard Ethical Decision Making

Sometimes ethical decision making is easy; for example, when it is clear that an action will prevent a serious harm and has no drawbacks, then that action is the right thing to do. Sometimes, however, ethical decision making is more complicated and challenging. Take the following case: your job is to make decisions about which parts to buy for a computer manufacturing company. A person who sells parts to the company offers you tickets to an expensive Broadway show. Should you accept the tickets? In this case, the right thing to do is more complicated because you may be able to accept the tickets and not have this affect your decision about parts. You owe your employer a decision on parts that is in the best interests of the company, but will accepting the tickets influence future decisions?

Other times, you know what the right thing to do is, but doing it will have such great personal costs that you cannot bring yourself to do it. For example, you might be considering blowing the whistle on your employer, who has been extremely kind and generous to you, but who now has asked you to cheat on the testing results on a life-critical software system designed for a client.

To make good decisions, professionals must be aware of potential issues and must have a fairly clear sense of their responsibilities in various kinds of situations. This often requires sorting out complex relationships and obligations, anticipating the effects of various actions, and balancing responsibilities to multiple parties. This activity is part of professional ethics.

2.3 Professional Ethics

Ethics is not just a matter for individuals as individuals. We all occupy a variety of social roles that involve special responsibilities and privileges. As parents, we have special responsibilities for children. As citizens, members of churches, officials in clubs, and so on, we have special rights and duties — and so it is with professional roles. Being a professional is often distinguished from merely having an occupation, because a professional makes a different sort of commitment. Being a professional means more than just having a job. The difference is commitment to doing the right thing because you are a member of a group that has taken on responsibility for a domain of activity. The group is accountable to society for this domain, and for this reason, professionals must behave in ways that are worthy of public trust.

Some theorists explain this commitment in terms of a social contract between a profession and the society in which it functions. Society grants special rights and privileges to the professional group, such as control of admission to the group, access to educational institutions, and confidentiality in professional–client relationships. Society, in turn, may even grant the group a monopoly over a domain of activity (e.g., only licensed engineers can sign off on construction designs, and only doctors can prescribe drugs). In exchange, the professional group promises to self-regulate and practice its profession in ways that are beneficial to society, that is, to promote safety, health, and welfare. The social contract idea is a way of illustrating the importance of the trust that clients and the public put in professionals; it shows the importance of professionals acting so as to be worthy of that trust.

The special responsibilities of professionals have been accounted for in other theoretical frameworks, as well. For example, Davis [1995] argues that members of professions implicitly, if not explicitly, agree among themselves to adhere to certain standards because this elevates the level of activity. If all computer scientists and engineers, for example, agreed never to release software that has not met certain testing standards, this would prevent market pressures from driving down the quality of software being produced. Davis's point is that the special responsibilities of professionals are grounded in what members of a professional group owe to one another: they owe it to one another to live up to agreed-upon rules and standards. Other theorists have tried to ground the special responsibilities of professionals in ordinary morality. Alpern [1991] argues, for example, that the engineer's responsibility for safety derives from the ordinary moral edict *do no harm*. Because engineers are in a position to do greater harm than others, engineers have a special responsibility in their work to take greater care.

In the case of computing professionals, responsibilities are not always well articulated because of several factors. Computing is a relatively new field. There is no single unifying professional association that

© 2004 by Taylor & Francis Group, LLC

controls membership, specifies standards of practice, and defines what it means to be a member of the profession. Moreover, many computer scientists and engineers are employees of companies or government agencies, and their role as computer professional may be somewhat in tension with their role as an employee of the company or agency. This can blur an individual's understanding of his or her professional responsibilities. Being a professional means having the independence to make decisions on the basis of special expertise, but being an employee often means acting in the best interests of the company, i.e., being loyal to the organization. Another difficulty in the role of computing professional is the diversity of the field. Computing professionals are employed in a wide variety of contexts, have a wide variety of kinds of expertise, and come from diverse educational backgrounds. As mentioned before, there is no single unifying organization, no uniform admission standard, and no single identifiable professional role.

To be sure, there are pressures on the field to move more in the direction of professionalization, but this seems to be happening to factions of the group rather than to the field as a whole. An important event moving the field in the direction of professionalization was the decision of the state of Texas to provide a licensing system for software engineers. The system specifies a set of requirements and offers an exam that must be passed in order for a computer professional to receive a software engineering license.

At the moment, Texas is the only state that offers such a license, so the field of computing remains loosely organized. It is not a strongly differentiated profession in the sense that there is no single characteristic (or set of characteristics) possessed by all computer professionals, no characteristic that distinguishes members of the group from anyone who possesses knowledge of computing. At this point, the field of computing is best described as a large group of individuals, all of whom work with computers, many of whom have expertise in subfields; they have diverse educational backgrounds, follow diverse career paths, and engage in a wide variety of job activities.

Despite the lack of unity in the field, there are many professional organizations, several professional codes of conduct, and expectations for professional practice. The codes of conduct, in particular, form the basis of an emerging professional ethic that may, in the future, be refined to the point where there will be a strongly differentiated role for computer professionals.

Professional codes play an important role in articulating a collective sense of both the ideal of the profession and the minimum standards required. Codes of conduct state the consensus views of members while shaping behavior.

A number of professional organizations have codes of ethics that are of interest here. The best known include the following:

The Association for Computing Machinery (ACM) Code of Ethics and Professional Conduct (see Appendix B)
The Institute of Electrical and Electronic Engineers (IEEE) Code of Ethics
The Joint ACM/IEEE Software Engineering Code of Ethics and Professional Practice
The Data Processing Managers Association (DPMA, now the Association of Information Technology Professionals [AITP]) Code of Ethics and Standards of Conduct
The Institute for Certification of Computer Professionals (ICCP) Code of Ethics
The Canadian Information Processing Society Code of Ethics
The British Computer Society Code of Conduct

Each of these codes has different emphases and goals. Each in its own way, however, deals with issues that arise in the context in which computer scientists and engineers typically practice.

The codes are relatively consistent in identifying computer professionals as having responsibilities to be faithful to their employers and clients, and to protect public safety and welfare. The most salient ethical issues that arise in professional practice have to do with balancing these responsibilities with personal (or nonprofessional) responsibilities. Two common areas of tension are worth mentioning here, albeit briefly.

As previously mentioned, computer scientists may find themselves in situations in which their responsibility as professionals to protect the public comes into conflict with loyalty to their employer. Such situations sometimes escalate to the point where the computer professional must decide whether to blow

© 2004 by Taylor & Francis Group, LLC

the whistle. Such a situation might arise, for example, when the computer professional believes that a piece of software has not been tested enough but her employer wants to deliver the software on time and within the allocated budget (which means immediate release and no more resources being spent on the project). Whether to blow the whistle is one of the most difficult decisions computer engineers and scientists may have to face. Whistle blowing has received a good deal of attention in the popular press and in the literature on professional ethics, because this tension seems to be built into the role of engineers and scientists, that is, the combination of being a professional with highly technical knowledge and being an employee of a company or agency.

Of course, much of the literature on whistle blowing emphasizes strategies that avoid the need for it. Whistle blowing can be avoided when companies adopt mechanisms that give employees the opportunity to express their concerns without fear of repercussions, for example, through ombudspersons to whom engineers and scientists can report their concerns anonymously. The need to blow the whistle can also be diminished when professional societies maintain hotlines that professionals can call for advice on how to get their concerns addressed.

Another important professional ethics issue that often arises is directly tied to the importance of being worthy of client (and, indirectly, public) trust. Professionals can find themselves in situations in which they have (or are likely to have) a conflict of interest. A conflict-of-interest situation is one in which the professional is hired to perform work for a client and the professional has some personal or professional interest that may (or may appear to) interfere with his or her judgment on behalf of the client. For example, suppose a computer professional is hired by a company to evaluate its needs and recommend hardware and software that will best suit the company. The computer professional does precisely what is requested, but fails to mention being a silent partner in a company that manufactures the hardware and software that has been recommended. In other words, the professional has a personal interest — financial benefit — in the company's buying certain equipment. If the company were told this upfront, it might expect the computer professional to favor his own company's equipment; however, if the company finds out about the affiliation later on, it might rightly think that it had been deceived. The professional was hired to evaluate the needs of the company and to determine how best to meet those needs, and in so doing to have the best interests of the company fully in mind. Now, the company suspects that the professional's judgment was biased. The professional had an interest that might have interfered with his judgment on behalf of the company.

There are a number of strategies that professions use to avoid these situations. A code of conduct may, for example, specify that professionals reveal all relevant interests to their clients before they accept a job. Or the code might specify that members never work in a situation where there is even the appearance of a conflict of interest.

This brings us to the special character of computer technology and the effects that the work of computer professionals can have on the shape of the world. Some may argue that computer professionals have very little say in what technologies get designed and built. This seems to be mistaken on at least two counts. First, we can distinguish between computer professionals as individuals and computer professionals as a group. Even if individuals have little power in the jobs they hold, they can exert power collectively. Second, individuals can have an effect if they think of themselves as professionals and consider it their responsibility to anticipate the impact of their work.

2.4 Ethical Issues That Arise from Computer Technology

The effects of a new technology on society can draw attention to an old issue and can change our understanding of that issue. The issues listed in this section — privacy, property rights, risk and reliability, and global communication — were of concern, even problematic, before computers were an important technology. But computing and, more generally, electronic telecommunications, have added new twists and new intensity to each of these issues. Although computer professionals cannot be expected to be experts on all of these issues, it is important for them to understand that computer technology is shaping the world. And it is important for them to keep these impacts in mind as they work with computer technology. Those

© 2004 by Taylor & Francis Group, LLC

who are aware of privacy issues, for example, are more likely to take those issues into account when they design database management systems; those who are aware of risk and reliability issues are more likely to articulate these issues to clients and attend to them in design and documentation.

2.4.1 Privacy

Privacy is a central topic in computer ethics. Some have even suggested that privacy is a notion that has been antiquated by technology and that it should be replaced by a new openness. Others think that computers must be harnessed to help restore as much privacy as possible to our society. Although they may not like it, computer professionals are at the center of this controversy. Some are designers of the systems that facilitate information gathering and manipulation; others maintain and protect the information. As the saying goes, *information is power* — but power can be used or abused.

Computer technology creates wide-ranging possibilities for tracking and monitoring of human behavior. Consider just two ways in which personal privacy may be affected by computer technology. First, because of the capacity of computers, massive amounts of information can be gathered by record-keeping organizations such as banks, insurance companies, government agencies, and educational institutions. The information gathered can be kept and used indefinitely, and shared with other organizations rapidly and frequently. A second way in which computers have enhanced the possibilities for monitoring and tracking of individuals is by making possible new kinds of information. When activities are done using a computer, transactional information is created. When individuals use automated bank teller machines, records are created; when certain software is operating, keystrokes on a computer keyboard are recorded; the content and destination of electronic mail can be tracked, and so on. With the assistance of newer technologies, much more of this transactional information is likely to be created. For example, television advertisers may be able to monitor television watchers with scanning devices that record who is sitting in a room facing the television. Highway systems allow drivers to pass through toll booths without stopping as a beam reading a bar code on the automobile charges the toll, simultaneously creating a record of individual travel patterns. All of this information (transactional and otherwise) can be brought together to create a detailed portrait of a person's life, a portrait that the individual may never see, although it is used by others to make decisions about the individual.

This picture suggests that computer technology poses a serious threat to personal privacy. However, one can counter this picture in a number of ways. Is it computer technology *per se* that poses the threat or is it just the way the technology has been used (and is likely to be used in the future)? Computer professionals might argue that they create the technology but are not responsible for how it is used. This argument is, however, problematic for a number of reasons and perhaps foremost because it fails to recognize the potential for solving some of the problems of abuse in the design of the technology. Computer professionals are in the ideal position to think about the potential problems with computers and to design so as to avoid these problems. When, instead of deflecting concerns about privacy as out of their purview, computer professionals set their minds to solve privacy and security problems, the systems they design can improve.

At the same time we think about changing computer technology, we also must ask deeper questions about privacy itself and what it is that individuals need, want, or are entitled to when they express concerns about the loss of privacy. In this sense, computers and privacy issues are ethical issues. They compel us to ask deep questions about what makes for a good and just society. Should individuals have more choice about who has what information about them? What is the proper relationship between citizens and government, between individuals and private corporations? How are we to negotiate the tension between the competing needs for privacy and security? As previously suggested, the questions are not completely new, but some of the possibilities created by computers are new, and these possibilities do not readily fit the concepts and frameworks used in the past. Although we cannot expect computer professionals to be experts on the philosophical and political analysis of privacy, it seems clear that the more they know, the better the computer technology they produce is likely to be.

© 2004 by Taylor & Francis Group, LLC

2.4.2 Property Rights and Computing

The protection of intellectual property rights has become an active legal and ethical debate, involving national and international players. Should software be copyrighted, patented, or free? Is computer software a process, a creative work, a mathematical formalism, an idea, or some combination of these? What is society's stake in protecting software rights? What is society's stake in widely disseminating software? How do corporations and other institutions protect their rights to ideas developed by individuals? And what are the individuals' rights? Such questions must be answered publicly through legislation, through corporate policies, and with the advice of computing professionals. Some of the answers will involve technical details, and all should be informed by ethical analysis and debate.

An issue that has received a great deal of legal and public attention is the ownership of software. In the course of history, software is a relatively new entity. Whereas Western legal systems have developed property laws that encourage invention by granting certain rights to inventors, there are provisions against ownership of things that might interfere with the development of the technological arts and sciences. For this reason, copyrights protect only the expression of ideas, not the ideas themselves, and we do not grant patents on laws of nature, mathematical formulas, and abstract ideas. The problem with computer software is that it has not been clear that we could grant ownership of it without, in effect, granting ownership of numerical sequences or mental steps. Software can be copyrighted, because a copyright gives the holder ownership of the *expression* of the idea (not the idea itself), but this does not give software inventors as much protection as they need to compete *fairly*. Competitors may see the software, grasp the idea, and write a somewhat different program to do the same thing. The competitor can sell the software at less cost because the cost of developing the first software does not have to be paid. Patenting would provide stronger protection, but until quite recently the courts have been reluctant to grant this protection because of the problem previously mentioned: patents on software would appear to give the holder control of the building blocks of the technology, an ownership comparable to owning ideas themselves. In other words, too many patents may interfere with technological development.

Like the questions surrounding privacy, property rights in computer software also lead back to broader ethical and philosophical questions about what constitutes a just society. In computing, as in other areas of technology, we want a system of property rights that promotes invention (creativity, progress), but at the same time, we want a system that is fair in the sense that it rewards those who make significant contributions but does not give anyone so much control that others are prevented from creating. Policies with regard to property rights in computer software cannot be made without an understanding of the technology. This is why it is so important for computer professionals to be involved in public discussion and policy setting on this topic.

2.4.3 Risk, Reliability, and Accountability

As computer technology becomes more important to the way we live, its risks become more worrisome. System errors can lead to physical danger, sometimes catastrophic in scale. There are security risks due to hackers and crackers. Unreliable data and intentional misinformation are risks that are increased because of the technical and economic characteristics of digital data. Furthermore, the use of computer programs is, in a practical sense, inherently unreliable.

Each of these issues (and many more) requires computer professionals to face the linked problems of risk, reliability, and accountability. Professionals must be candid about the risks of a particular application or system. Computing professionals should take the lead in educating customers and the public about what predictions we can and cannot make about software and hardware reliability. Computer professionals should make realistic assessments about costs and benefits, and be willing to take on both for projects in which they are involved.

There are also issues of sharing risks as well as resources. Should liability fall to the individual who buys software or to the corporation that developed it? Should society acknowledge the inherent risks in using

© 2004 by Taylor & Francis Group, LLC

software in life-critical situations and shoulder some of the responsibility when something goes wrong? Or should software providers (both individuals and institutions) be exclusively responsible for software safety? All of these issues require us to look at the interaction of technical decisions, human consequences, rights, and responsibilities. They call not just for technical solutions but for solutions that recognize the kind of society we want to have and the values we want to preserve.

2.4.4 Rapidly Evolving Globally Networked Telecommunications

The system of computers and connections known as the Internet provides the infrastructure for new kinds of communities — electronic communities. Questions of individual accountability and social control, as well as matters of etiquette, arise in electronic communities, as in all societies. It is not just that we have societies forming in a new physical environment; it is also that ongoing electronic communication changes the way individuals understand their identity, their values, and their plans for their lives. The changes that are taking place must be examined and understood, especially the changes affecting fundamental social values such as democracy, community, freedom, and peace.

Of course, speculating about the Internet is now a popular pastime, and it is important to separate the hype from the reality. The reality is generally much more complex and much more subtle. We will not engage in speculation and prediction about the future. Rather, we want to emphasize how much better off the world would be if (instead of watching social impacts of computer technology after the fact) computer engineers and scientists were thinking about the potential effects early in the design process. Of course, this can only happen if computer scientists and engineers are encouraged to see the social–ethical issues as a component of their professional responsibility. This chapter has been written with that end in mind.

2.5 Final Thoughts

Computer technology will, no doubt, continue to evolve and will continue to affect the character of the world we live in. Computer scientists and engineers will play an important role in shaping the technology. The technologies we use shape how we live and who we are. They make every difference in the moral environment in which we live. Hence, it seems of utmost importance that computer scientists and engineers understand just how their work affects humans and human values.

References

Alpern, K.D. 1991. Moral responsibility for engineers. In *Ethical Issues in Engineering*, D.G. Johnson, Ed., pp. 187–195. Prentice Hall, Englewood Cliffs, NJ.

Collins, W.R., and Miller, K. 1992. A paramedic method for computing professionals. *J. Syst. Software*. 17(1): 47–84.

Davis, M. 1995. Thinking like an engineer: the place of a code of ethics in the practice of a profession. In *Computers, Ethics, and Social Values*, D.G. Johnson and H. Nissenbaum, Eds., pp. 586–597. Prentice Hall, Englewood Cliffs, NJ.

Johnson, D.G. 2001. *Computer Ethics*, *3rd edition*. Prentice Hall, Englewood Cliffs, NJ.

Kant, I. 1785. *Foundations of the Metaphysics of Morals*. L. Beck, trans., 1959. Library of Liberal Arts, 1959.

Rawls, J. 1971. *A Theory of Justice*. Harvard Univ. Press, Cambridge, MA.

© 2004 by Taylor & Francis Group, LLC

I

Algorithms and Complexity

This section addresses the challenges of solving hard problems algorithmically and efficiently. These chapters cover basic methodologies (divide and conquer), data structures, complexity theory (space and time measures), parallel algorithms, and strategies for solving hard problems and identifying unsolvable problems. They also cover some exciting contemporary applications of algorithms, including cryptography, genetics, graphs and networks, pattern matching and text compression, and geometric and algebraic algorithms.

© 2004 by Taylor & Francis Group, LLC

© 2004 by Taylor & Francis Group, LLC

3 Basic Techniques for Design and Analysis of Algorithms

Edward M. Reingold
Illinois Institute of Technology

3.1 Introduction

We outline the basic methods of algorithm design and analysis that have found application in the manipulation of discrete objects such as lists, arrays, sets, graphs, and geometric objects such as points, lines, and polygons. We begin by discussing recurrence relations and their use in the analysis of algorithms. Then we discuss some specific examples in algorithm analysis, sorting, and priority queues. In the final three sections, we explore three important techniques of algorithm design: divide-and-conquer, dynamic programming, and greedy heuristics.

3.2 Analyzing Algorithms

It is convenient to classify algorithms based on the relative amount of time they require: how fast does the time required grow as the size of the problem increases? For example, in the case of arrays, the "size of the problem" is ordinarily the number of elements in the array. If the size of the problem is measured by a variable n, we can express the time required as a function of n, $T(n)$. When this function $T(n)$ grows rapidly, the algorithm becomes unusable for large n; conversely, when $T(n)$ grows slowly, the algorithm remains useful even when n becomes large.

We say an algorithm is $\Theta(n^2)$ if the time it takes quadruples when n doubles; an algorithm is $\Theta(n)$ if the time it takes doubles when n doubles; an algorithm is $\Theta(\log n)$ if the time it takes increases by a constant, independent of n, when n doubles; an algorithm is $\Theta(1)$ if its time does not increase at all when n increases. In general, an algorithm is $\Theta(T(n))$ if the time it requires on problems of size n grows proportionally to $T(n)$ as n increases. Table 3.1 summarizes the common growth rates encountered in the analysis of algorithms.

© 2004 by Taylor & Francis Group, LLC

TABLE 3.1 Common Growth Rates of Times of Algorithms

Rate of Growth	Comment	Examples
$\Theta(1)$	Time required is constant, independent of problem size	Expected time for hash searching
$\Theta(\log\log n)$	Very slow growth of time required	Expected time of interpolation search
$\Theta(\log n)$	Logarithmic growth of time required — doubling the problem size increases the time by only a constant amount	Computing x^n; binary search of an array
$\Theta(n)$	Time grows linearly with problem size — doubling the problem size doubles the time required	Adding/subtracting n-digit numbers; linear search of an n-element array
$\Theta(n\log n)$	Time grows worse than linearly, but not much worse — doubling the problem size more than doubles the time required	Merge sort; heapsort; lower bound on comparison-based sorting
$\Theta(n^2)$	Time grows quadratically — doubling the problem size quardruples the time required	Simple-minded sorting algorithms
$\Theta(n^3)$	Time grows cubically — doubling the problem size results in an eight fold increase in the time required	Ordinary matrix multiplication
$\Theta(c^n)$	Time grows exponentially — increasing the problem size by 1 results in a c-fold increase in the time required; doubling the problem size *squares* the time required	Traveling salesman problem

The analysis of an algorithm is often accomplished by finding and solving a recurrence relation that describes the time required by the algorithm. The most commonly occurring families of recurrences in the analysis of algorithms are linear recurrences and divide-and-conquer recurrences. In the following subsection we describe the "method of operators" for solving linear recurrences; in the next subsection we describe how to transform divide-and-conquer recurrences into linear recurrences by substitution to obtain an asymptotic solution.

3.2.1 Linear Recurrences

A *linear recurrence with constant coefficients* has the form

$$c_0a_n + c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k} = f(n), \tag{3.1}$$

for some constant k, where each c_i is constant. To solve such a recurrence for a broad class of functions f (that is, to express a_n in closed form as a function of n) by the *method of operators*, we consider two basic operators on sequences: $\mathcal{S}$, which shifts the sequence left,

$$\mathcal{S}\langle a_0, a_1, a_2, \ldots\rangle = \langle a_1, a_2, a_3, \ldots\rangle,$$

and C, which, for any constant C, multiplies each term of the sequence by C:

$$C\langle a_0, a_1, a_2, \ldots\rangle = \langle Ca_0, Ca_1, Ca_2, \ldots\rangle.$$

Then, given operators A and B, we define the sum and product

$$\begin{aligned}(A+B)\langle a_0, a_1, a_2, \ldots\rangle &= A\langle a_0, a_1, a_2, \ldots\rangle + B\langle a_0, a_1, a_2, \ldots\rangle,\\ (AB)\langle a_0, a_1, a_2, \ldots\rangle &= A(B\langle a_0, a_1, a_2, \ldots\rangle).\end{aligned}$$

Thus, for example,

$$(\mathcal{S}^2 - 4)\langle a_0, a_1, a_2, \ldots\rangle = \langle a_2 - 4a_0, a_3 - 4a_1, a_4 - 4a_2, \ldots\rangle,$$

which we write more briefly as

$$(\mathcal{S}^2 - 4)\langle a_i\rangle = \langle a_{i+2} - 4a_i\rangle.$$

© 2004 by Taylor & Francis Group, LLC

With the operator notation, we can rewrite Equation (3.1) as

$$P(\mathcal{S})\langle a_i \rangle = \langle f(i) \rangle,$$

where

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k$$

is a polynomial in $\mathcal{S}$.

Given a sequence $\langle a_i \rangle$, we say that the operator $P(\mathcal{S})$ *annihilates* $\langle a_i \rangle$ if $P(\mathcal{S})\langle a_i \rangle = \langle 0 \rangle$. For example, $\mathcal{S}^2 - 4$ annihilates any sequence of the form $\langle u2^i + v(-2)^i \rangle$, with constants u and v. In general,

$$\text{The operator } \mathcal{S}^{k+1} - c \text{ annihilates } \langle c^i \times \text{a polynomial in } i \text{ of degree } k \rangle.$$

The *product* of two annihilators annihilates the *sum* of the sequences annihilated by each of the operators, that is, if A annihilates $\langle a_i \rangle$ and B annihilates $\langle b_i \rangle$, then AB annihilates $\langle a_i + b_i \rangle$. Thus, determining the annihilator of a sequence is tantamount to determining the sequence; moreover, it is straightforward to determine the annihilator from a recurrence relation.

For example, consider the Fibonacci recurrence

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{i+2} &= F_{i+1} + F_i. \end{aligned}$$

The last line of this definition can be rewritten as $F_{i+2} - F_{i+1} - F_i = 0$, which tells us that $\langle F_i \rangle$ is annihilated by the operator

$$\mathcal{S}^2 - \mathcal{S} - 1 = (\mathcal{S} - \phi)(\mathcal{S} + 1/\phi),$$

where $\phi = (1 + \sqrt{5})/2$. Thus we conclude that

$$F_i = u\phi^i + v(-\phi)^{-i}$$

for some constants u and v. We can now use the initial conditions $F_0 = 0$ and $F_1 = 1$ to determine u and v: These initial conditions mean that

$$\begin{aligned} u\phi^0 + v(-\phi)^{-0} &= 0 \\ u\phi^1 + v(-\phi)^{-1} &= 1 \end{aligned}$$

and these linear equations have the solution

$$u = v = 1/\sqrt{5},$$

and hence

$$F_i = \phi^i/\sqrt{5} + (-\phi)^{-i}/\sqrt{5}.$$

In the case of the similar recurrence,

$$\begin{aligned} G_0 &= 0 \\ G_1 &= 1 \\ G_{i+2} &= G_{i+1} + G_i + i, \end{aligned}$$

© 2004 by Taylor & Francis Group, LLC

TABLE 3.2 Rate of Growth of the Solution to the Recurrence $T(n) = g(n) + uT(n/v)$: The Divide-and-Conquer Recurrence Relations

$g(n)$	u, v	Growth Rate of $T(n)$
$\Theta(1)$	$u = 1$	$\Theta(\log n)$
	$u \neq 1$	$\Theta(n^{\log_v u})$
$\Theta(\log n)$	$u = 1$	$\Theta[(\log n)^2]$
	$u \neq 1$	$\Theta(n^{\log_v u})$
$\Theta(n)$	$u < v$	$\Theta(n)$
	$u = v$	$\Theta(n \log n)$
	$u > v$	$\Theta(n^{\log_v u})$
$\Theta(n^2)$	$u < v^2$	$\Theta(n^2)$
	$u = v^2$	$\Theta(n^2 \log n)$
	$u > v^2$	$\Theta(n^{\log_v u})$

u and v are positive constants, independent of n, and $v > 1$.

the last equation tells us that

$$(\mathcal{S}^2 - \mathcal{S} - 1)\langle G_i \rangle = \langle i \rangle,$$

so the annihilator for $\langle G_i \rangle$ is $(\mathcal{S}^2 - \mathcal{S} - 1)(\mathcal{S} - 1)^2$ since $(\mathcal{S} - 1)^2$ annihilates $\langle i \rangle$ (a polynomial of degree 1 in i) and hence the solution is

$$G_i = u\phi^i + v(-\phi)^{-i} + \text{(a polynomial of degree 1 in i)};$$

that is,

$$G_i = u\phi^i + v(-\phi)^{-i} + wi + z.$$

Again, we use the initial conditions to determine the constants u, v, w, and x.

In general, then, to solve the recurrence in Equation 3.1, we factor the annihilator

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k,$$

multiply it by the annihilator for $\langle f(i) \rangle$, write the form of the solution from this product (which is the annihilator for the sequence $\langle a_i \rangle$), and the use the initial conditions for the recurrence to determine the coefficients in the solution.

3.2.2 Divide-and-Conquer Recurrences

The divide-and-conquer paradigm of algorithm construction that we discuss in Section 4 leads naturally to divide-and-conquer recurrences of the type

$$T(n) = g(n) + uT(n/v),$$

for constants u and v, $v > 1$, and sufficient initial values to define the sequence $\langle T(0), T(1), T(2), \ldots \rangle$. The growth rates of $T(n)$ for various values of u and v are given in Table 3.2. The growth rates in this table are derived by transforming the divide-and-conquer recurrence into a linear recurrence for a subsequence of $\langle T(0), T(1), T(2), \ldots \rangle$.

To illustrate this method, we derive the penultimate line in Table 3.2. We want to solve

$$T(n) = n^2 + v^2 T(n/v).$$

© 2004 by Taylor & Francis Group, LLC

So, we want to find a subsequence of $\langle T(0), T(1), T(2), \ldots \rangle$ that will be easy to handle. Let $n_k = v^k$; then,

$$T(n_k) = n_k^2 + v^2 T(n_k/v),$$

or

$$T(v^k) = v^{2k} + v^2 T(v^{k-1}).$$

Defining $t_k = T(v^k)$,

$$t_k = v^{2k} + v^2 t_{k-1}.$$

The annihilator for t_k is then $(\mathcal{S} - v^2)^2$ and thus

$$t_k = v^{2k}(ak + b),$$

for constants a and b. Expressing this in terms of $T(n)$,

$$T(n) \approx t_{\log_v n} = v^{2\log_v n}(a\log_v n + b) = an^2 \log_v n + bn^2,$$

or,

$$T(n) = \Theta(n^2 \log n).$$

3.3 Some Examples of the Analysis of Algorithms

In this section we introduce the basic ideas of analyzing algorithms by looking at some data structure problems that commonly occur in practice, problems relating to maintaining a collection of n objects and retrieving objects based on their relative size. For example, how can we determine the smallest of the elements? Or, more generally, how can we determine the kth largest of the elements? What is the running time of such algorithms in the worst case? Or, on average, if all $n!$ permutations of the input are equally likely? What if the set of items is dynamic — that is, the set changes through insertions and deletions — how efficiently can we keep track of, say, the largest element?

3.3.1 Sorting

The most demanding request that we can make of an array of n values `x[1]`, `x[2]`, ..., `x[n]` is that they be kept in perfect order so that `x[1]` $\leq$ `x[2]` $\leq \cdots \leq$ `x[n]`. The simplest way to put the values in order is to mimic what we might do by hand: take item after item and insert each one into the proper place among those items already inserted:

```
void insert (float x[], int i, float a) {
 // Insert a into x[1] ... x[i]
 // x[1] ... x[i-1] are sorted;  x[i] is unoccupied
 if (i == 1 || x[i-1] <= a)
  x[i] = a;
 else {
  x[i] = x[i-1];
  insert(x, i-1, a);
 }
}

void insertionSort (int n, float x[]) {
 // Sort  x[1] ... x[n]
```

© 2004 by Taylor & Francis Group, LLC

```
  if (n > 1) {
    insertionSort(n-1, x);
    insert(x, n, x[n]);
  }
}
```

To determine the time required in the worst case to sort n elements with `insertionSort`, we let t_n be the time to sort n elements and derive and solve a recurrence relation for t_n. We have,

$$t_n \begin{cases} \Theta(1) & \text{if } n = 1, \\ t_{n-1} + s_{n-1} + \Theta(1) & \text{otherwise,} \end{cases}$$

where s_m is the time required to insert an element in place among m elements using `insert`. The value of s_m is also given by a recurrence relation:

$$s_m \begin{cases} \Theta(1) & \text{if } m = 1, \\ s_{m-1} + \Theta(1) & \text{otherwise.} \end{cases}$$

The annihilator for $\langle s_i \rangle$ is $(\mathcal{S} - 1)^2$, so $s_m = \Theta(m)$. Thus, the annihilator for $\langle t_i \rangle$ is $(\mathcal{S} - 1)^3$, so $t_n = \Theta(n^2)$. The analysis of the average behavior is nearly identical; only the constants hidden in the Θ-notation change.

We can design better sorting methods using the divide-and-conquer idea of the next section. These algorithms avoid $\Theta(n^2)$ worst-case behavior, working in time $\Theta(n \log n)$. We can also achieve time $\Theta(n \log n)$ using a clever way of viewing the array of elements to be sorted as a tree: consider `x[1]` as the root of the tree and, in general, `x[2*i]` is the root of the left subtree of `x[i]` and `x[2*i+1]` is the root of the right subtree of `x[i]`. If we further insist that parents be greater than or equal to children, we have a *heap*; Figure 3.1 shows a small example.

A heap can be used for sorting by observing that the largest element is at the root, that is, `x[1]`; thus, to put the largest element in place, we swap `x[1]` and `x[n]`. To continue, we must restore the heap property, which may now be violated at the root. Such restoration is accomplished by swapping `x[1]` with its larger child, if that child is larger than `x[1]`, and the continuing to swap it downward until either it reaches the bottom or a spot where it is greater or equal to its children. Because the tree-cum-array has height $\Theta(\log n)$, this restoration process takes time $\Theta(\log n)$. Now, with the heap in `x[1]` to `x[n-1]` and `x[n]` the largest value in the array, we can put the second largest element in place by swapping `x[1]` and `x[n-1]`; then we restore the heap property in `x[1]` to `x[n-2]` by propagating `x[1]` downward; this takes time $\Theta(\log(n - 1))$. Continuing in this fashion, we find we can sort the entire array in time

$$\Theta(\log n + \log(n - 1) + \cdots + \log 1) = \Theta(n \log n).$$

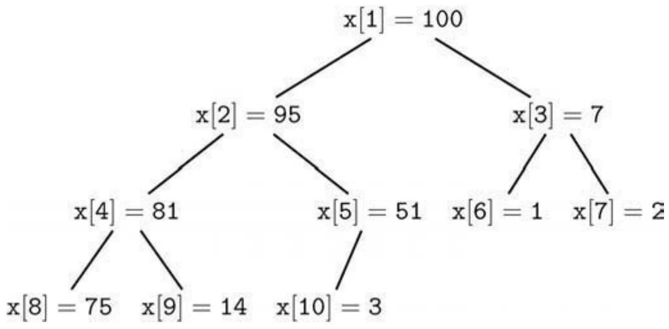


FIGURE 3.1 A heap — that is, an array, interpreted as a binary tree.

© 2004 by Taylor & Francis Group, LLC

The initial creation of the heap from an unordered array is done by applying the restoration process successively to `x[n/2]`, `x[n/2-1]`, ..., `x[1]`, which takes time $\Theta(n)$.

Hence, we have the following $\Theta(n \log n)$ sorting algorithm:

```
void heapify (int n, float x[], int i) {
  // Repair heap property below x[i] in x[1] ... x[n]
  int largest = i;  // largest of x[i], x[2*i], x[2*i+1]
  if (2*i <= n && x[2*i] > x[i])
    largest = 2*i;
  if (2*i+1 <= n && x[2*i+1] > x[largest])
    largest = 2*i+1;
  if (largest != i) {
    // swap x[i] with larger child and repair heap below
    float t = x[largest]; x[largest] = x[i]; x[i] = t;
    heapify(n, x, largest);
  }
}

void makeheap (int n, float x[]) {
  // Make x[1] ... x[n] into a heap
  for (int i=n/2; i>0; i--)
    heapify(n, x, i);
}

void heapsort (int n, float x[]) {
  // Sort  x[1] ... x[n]
  float t;
  makeheap(n, x);
  for (int i=n; i>1; i--) {
    // put x[1] in place and repair heap
    t = x[1]; x[1] = x[i]; x[i] = t;
    heapify(i-1, x, 1);
  }
}
```

Can we find sorting algorithms that take less time than $\Theta(n \log n)$? The answer is no if we are restricted to sorting algorithms that derive their information from comparisons between the values of elements. The flow of control in such sorting algorithms can be viewed as binary trees in which there are $n!$ leaves, one for every possible sorted output arrangement. Because a binary tree with height h can have at most 2^h leaves, it follows that the height of a tree with $n!$ leaves must be at least $\log_2 n! = \Theta(n \log n)$. Because the height of this tree corresponds to the longest sequence of element comparisons possible in the flow of control, any such sorting algorithm must, in its worst case, use time proportional to $n \log n$.

3.3.2 Priority Queues

Aside from its application to sorting, the heap is an interesting data structure in its own right. In particular, heaps provide a simple way to implement a *priority queue*; a priority queue is an abstract data structure that keeps track of a dynamically changing set of values allowing the operations

`create`: Create an empty priority queue.
`insert`: Insert a new element into a priority queue.
`decrease`: Decrease an element in a priority queue.
`minimum`: Report the minimum element in a priority queue.

© 2004 by Taylor & Francis Group, LLC

`deleteMinimum`: Delete the minimum element in a priority queue.
`delete`: Delete an element in a priority queue.
`merge`: Merge two priority queues.

A heap can implement a priority queue by altering the heap property to insist that parents are less than or equal to their children, so that that smallest value in the heap is at the root, that is, in the first array position. Creation of an empty heap requires just the allocation of an array, an $\Theta(1)$ operation; we assume that once created, the array containing the heap can be extended arbitrarily at the right end. Inserting a new element means putting that element in the $(n+1)$st location and "bubbling it up" by swapping it with its parent until it reaches either the root or a parent with a smaller value. Because a heap has logarithmic height, insertion to a heap of n elements thus requires worst-case time $O(\log n)$. Decreasing a value in a heap requires only a similar $O(\log n)$ "bubbling up." The minimum element of such a heap is always at the root, so reporting it takes $\Theta(1)$ time. Deleting the minimum is done by swapping the first and last array positions, bubbling the new root value downward until it reaches its proper location, and truncating the array to eliminate the last position. `Delete` is handled by decreasing the value so that it is the least in the heap and then applying the `deleteMinimum` operation; this takes a total of $O(\log n)$ time.

The `merge` operation, unfortunately, is not so economically accomplished; there is little choice but to create a new heap out of the two heaps in a manner similar to the `makeheap` function in heapsort. If there are a total of n elements in the two heaps to be merged, this re-creation will require time $O(n)$.

There are better data structures than a heap for implementing priority queues, however. In particular, the *Fibonacci heap* provides an implementation of priority queues in which the `delete` and `deleteMinimum` operations take $O(\log n)$ time and the remaining operations take $\Theta(1)$ time, *provided we consider the times required for a sequence of priority queue operations, rather than individual times.* That is, we must consider the cost of the individual operations *amortized over the sequence of operations*: Given a sequence of n priority queue operations, we will compute the total time $T(n)$ for all n operations. In doing this computation, however, we do not simply add the costs of the individual operations; rather, we subdivide the cost of each operation into two parts: the *immediate cost* of doing the operation and the *long-term savings* that result from doing the operation. The long-term savings represent costs *not* incurred by later operations as a result of the present operation. The immediate cost minus the long-term savings give the amortized cost of the operation.

It is easy to calculate the immediate cost (time required) of an operation, but how can we measure the long-term savings that result? We imagine that the data structure has associated with it a bank account; at any given moment, the bank account must have a non-negative balance. When we do an operation that will save future effort, we are making a deposit to the savings account; and when, later on, we derive the benefits of that earlier operation, we are making a withdrawal from the savings account. Let $\mathcal{B}(i)$ denote the balance in the account after the ith operation, $\mathcal{B}(0) = 0$. We define the amortized cost of the ith operation to be

$$\begin{aligned}\text{Amortized cost of } i\text{th operation} &= (\text{Immediate cost of } i\text{th operation}) + (\text{Change in bank account})\\ &= (\text{Immediate cost of } i\text{th operation}) + (\mathcal{B}(i) - \mathcal{B}(i-1)).\end{aligned}$$

Because the bank account $\mathcal{B}$ can go up or down as a result of the ith operation, the amortized cost may be less than or more than the immediate cost. By summing the previous equation, we get

$$\begin{aligned}\sum_{i=1}^{n}(\text{Amortized cost of } i\text{th operation}) &= \sum_{i=1}^{n}(\text{Immediate cost of } i\text{th operation}) + (\mathcal{B}(n) - \mathcal{B}(0))\\ &= (\text{Total cost of all } n \text{ operations}) + \mathcal{B}(n)\\ &\geq \text{Total cost of all } n \text{ operations}\\ &= T(n)\end{aligned}$$

© 2004 by Taylor & Francis Group, LLC

because $\mathcal{B}(i)$ is non-negative. Thus defined, the sum of the amortized costs of the operations gives us an upper bound on the total time $T(n)$ for all n operations.

It is important to note that the function $\mathcal{B}(i)$ is not part of the data structure, but is just our way to measure how much time is used by the sequence of operations. As such, we can choose *any rules* for $\mathcal{B}$, provided $\mathcal{B}(0) = 0$ and $\mathcal{B}(i) \geq 0$ for $i \geq 1$. Then the sum of the amortized costs defined by

$$\text{Amortized cost of } i\text{th operation} = (\text{Immediate cost of } i\text{th operation}) + (\mathcal{B}(i) - \mathcal{B}(i-1))$$

bounds the overall cost of the operation of the data structure.

Now to apply this method to priority queues. A *Fibonacci heap* is a list of heap-ordered trees (not necessarily binary); because the trees are heap ordered, the minimum element must be one of the roots and we keep track of which root is the overall minimum. Some of the tree nodes are *marked*. We define

$$\begin{aligned}\mathcal{B}(i) = {} & (\text{Number of trees after the } i\text{th operation}) \\ & + 2 \times (\text{Number of marked nodes after the } i\text{th operation}).\end{aligned}$$

The clever rules by which nodes are marked and unmarked, and the intricate algorithms that manipulate the set of trees, are too complex to present here in their complete form, so we just briefly describe the simpler operations and show the calculation of their amortized costs:

`Create`: To create an empty Fibonacci heap we create an empty list of heap-ordered trees. The immediate cost is $\Theta(1)$; because the numbers of trees and marked nodes are zero before and after this operation, $\mathcal{B}(i) - \mathcal{B}(i-1)$ is zero and the amortized time is $\Theta(1)$.

`Insert`: To insert a new element into a Fibonacci heap we add a new one-element tree to the list of trees constituting the heap and update the record of what root is the overall minimum. The immediate cost is $\Theta(1)$. $\mathcal{B}(i) - \mathcal{B}(i-1)$ is also 1 because the number of trees has increased by 1, while the number of marked nodes is unchanged. The amortized time is thus $\Theta(1)$.

`Decrease`: Decreasing an element in a Fibonacci heap is done by cutting the link to its parent, if any, adding the item as a root in the list of trees, and decreasing its value. Furthermore, the marked parent of a cut element is itself cut, propagating upward in the tree. Cut nodes become unmarked, and the unmarked parent of a cut element becomes marked. The immediate cost of this operation is $\Theta(c)$, where c is the number of cut nodes. If there were t trees and m marked elements before this operation, the value of $\mathcal{B}$ before the operation was $t + 2m$. After the operation, the value of $\mathcal{B}$ is $(t+c)+2(m-c+2)$, so $\mathcal{B}(i)-\mathcal{B}(i-1) = 4-c$. The amortized time is thus $\Theta(c)+4-c = \Theta(1)$ *by changing the definition of $\mathcal{B}$ by a multiplicative constant large enough to dominate the constant hidden in $\Theta(c)$.*

`Minimum`: Reporting the minimum element in a Fibonacci heap takes time $\Theta(1)$ and does not change the numbers of trees and marked nodes; the amortized time is thus $\Theta(1)$.

`DeleteMinimum`: Deleting the minimum element in a Fibonacci heap is done by deleting that tree root, making its children roots in the list of trees. Then, the list of tree roots is "consolidated" in a complicated $O(\log n)$ operation that we do not describe. The result takes amortized time $O(\log n)$.

`Delete`: Deleting an element in a Fibonacci heap is done by decreasing its value to $-\infty$ and then doing a `deleteMinimum`. The amortized cost is the sum of the amortized cost of the two operations, $O(\log n)$.

`Merge`: Merging two Fibonacci heaps is done by concatenating their lists of trees and updating the record of which root is the minimum. The amortized time is thus $\Theta(1)$.

Notice that the amortized cost of each operation is $\Theta(1)$ except `deleteMinimum` and `delete`, both of which are $O(\log n)$.

© 2004 by Taylor & Francis Group, LLC

3.4 Divide-and-Conquer Algorithms

One approach to the design of algorithms is to decompose a problem into subproblems that resemble the original problem, but on a reduced scale. Suppose, for example, that we want to compute x^n. We reason that the value we want can be computed from $x^{\lfloor n/2 \rfloor}$ because

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ (x^{\lfloor n/2 \rfloor})2 & \text{if } n \text{ is even,} \\ x \times (x^{\lfloor n/2 \rfloor})2 & \text{if } n \text{ is odd.} \end{cases}$$

This recursive definition can be translated directly into

```
 1  int power (float x, int n) {
 2   // Compute the n-th power of x
 3    if (n == 0)
 4      return 1;
 5    else {
 6      int t = power(x, floor(n/2));
 7      if ((n % 2) == 0)
 8        return t*t;
 9      else
10        return x*t*t;
11    }
12  }
```

To analyze the time required by this algorithm, we notice that the time will be proportional to the number of multiplication operations performed in lines 8 and 10, so the divide-and-conquer recurrence

$$T(n) = 2 + T(\lfloor n/2 \rfloor),$$

with $T(0) = 0$, describes the rate of growth of the time required by this algorithm. By considering the subsequence $n_k = 2^k$, we find, using the methods of the previous section, that $T(n) = \Theta(\log n)$. Thus, the above algorithm is considerably more efficient than the more obvious

```
int power (int k, int n) {
 // Compute the n-th power of k
  int product = 1;
  for (int i = 1; i <= n; i++)
    // at this point power is k*k*k*...*k (i times)
    product = product * k;
  return product;
}
```

which requires time $\Theta(n)$.

An extremely well-known instance of divide-and-conquer algorithms is *binary search* of an ordered array of n elements for a given element; we "probe" the middle element of the array, continuing in either the lower or upper segment of the array, depending on the outcome of the probe:

```
int binarySearch (int x, int w[], int low, int high) {
  // Search for x among sorted array w[low..high]. The integer returned
  // is either the location of x in w, or the location where x belongs.
  if (low > high) // Not found
    return low;
```

© 2004 by Taylor & Francis Group, LLC

```
  else {
    int middle := (low+high)/2;
    if (w[middle] < x)
      return binarySearch(x, w, middle+1, high);
    else if (w[middle] == x)
      return middle;
    else
      return binarySearch(x, w, low, middle-1);
  }
}
```

The analysis of binary search in an array of n elements is based on counting the number of probes used in the search, because all remaining work is proportional to the number of probes. But, the number of probes needed is described by the divide-and-conquer recurrence

$$T(n) = 1 + T(n/2),$$

with $T(0) = 0$, $T(1) = 1$. We find from Table 3.2 (the top line) that $T(n) = \Theta(\log n)$. Hence, binary search is much more efficient than a simple linear scan of the array.

To multiply two very large integers x and y, assume that x has exactly $l \geq 2$ digits and y has at most l digits. Let $x_0, x_1, x_2, \ldots, x_{l-1}$ be the digits of x and let $y_0, y_1, \ldots, y_{l-1}$ be the digits of y (some of the significant digits at the end of y may be zeros, if y is shorter than x), so that

$$x = x_0 + 10x_1 + 10^2 x_2 + \cdots + 10^{l-1} x_{l-1},$$

and

$$y = y_0 + 10y_1 + 10^2 y_2 + \cdots + 10^{l-1} y_{l-1},$$

We apply the divide-and-conquer idea to multiplication by chopping x into two pieces — the leftmost n digits and the remaining digits:

$$x = x_{\text{left}} + 10^n x_{\text{right}},$$

where $n = l/2$. Similarly, chop y into two corresponding pieces:

$$y = y_{\text{left}} + 10^n y_{\text{right}},$$

because y has at most the number of digits that x does, y_{right} might be 0. The product $x \times y$ can be now written

$$\begin{aligned} x \times y &= (x_{\text{left}} + 10^n x_{\text{right}}) \times (y_{\text{left}} + 10^n y_{\text{right}}), \\ &= x_{\text{left}} \times y_{\text{left}} \\ &\quad + 10^n (x_{\text{right}} \times y_{\text{left}} + x_{\text{left}} \times y_{\text{right}}) \\ &\quad + 10^{2n} x_{\text{right}} \times y_{\text{right}}. \end{aligned}$$

If $T(n)$ is the time to multiply two n-digit numbers with this method, then

$$T(n) = kn + 4T(n/2);$$

the kn part is the time to chop up x and y and to do the needed additions and shifts; each of these tasks involves n-digit numbers and hence $\Theta(n)$ time. The $4T(n/2)$ part is the time to form the four needed subproducts, each of which is a product of about $n/2$ digits.

© 2004 by Taylor & Francis Group, LLC

The line for $g(n) = \Theta(n)$, $u = 4 > v = 2$ in Table 3.2 tells us that $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$, so the divide-and-conquer algorithm is no more efficient than the elementary-school method of multiplication. However, we can be more economical in our formation of subproducts:

$$\begin{aligned} x \times y &= \left(x_{\text{left}} + 10^n x_{\text{right}}\right) \times \left(y_{\text{left}} + 10^n y_{\text{right}}\right), \\ &= B + 10^n C + 10^{2n} A, \end{aligned}$$

where

$$\begin{aligned} A &= x_{\text{right}} \times y_{\text{right}} \\ B &= x_{\text{left}} \times y_{\text{left}} \\ C &= (x_{\text{left}} + x_{\text{right}}) \times (y_{\text{left}} + y_{\text{right}}) - A - B. \end{aligned}$$

The recurrence for the time required changes to

$$T(n) = kn + 3T(n/2).$$

The kn part is the time to do the two additions that form $x \times y$ from A, B, and C and the two additions and the two subtractions in the formula for C; each of these six additions/subtractions involves n-digit numbers. The $3T(n/2)$ part is the time to (recursively) form the three needed products, each of which is a product of about $n/2$ digits. The line for $g(n) = \Theta(n)$, $u = 3 > v = 2$ in Table 3.2 now tells us that

$$T(n) = \Theta\left(n^{\log_2 3}\right).$$

Now,

$$\log_2 3 = \frac{\log_{10} 3}{\log_{10} 2} \approx 1.5849625\cdots,$$

which means that this divide-and-conquer multiplication technique will be faster than the straightforward $\Theta(n^2)$ method for large numbers of digits.

Sorting a sequence of n values efficiently can be done using the divide-and-conquer idea. Split the n values arbitrarily into two piles of $n/2$ values each, sort each of the piles separately, and then merge the two piles into a single sorted pile. This sorting technique, pictured in Figure 3.2, is called *merge sort*. Let $T(n)$ be the time required by merge sort for sorting n values. The time needed to do the merging is proportional to the number of elements being merged, so that

$$T(n) = cn + 2T(n/2),$$

because we must sort the two halves (time $T(n/2)$ each) and then merge (time proportional to n). We see by Table 3.2 that the growth rate of $T(n)$ is $\Theta(n \log n)$, since $u = v = 2$ and $g(n) = \Theta(n)$.

3.5 Dynamic Programming

In the design of algorithms to solve optimization problems, we need to make the optimal (lowest cost, highest value, shortest distance, etc.) choice from among a large number of alternative solutions. *Dynamic programming* is an organized way to find an optimal solution by systematically exploring all possibilities without unnecessary repetition. Often, dynamic programming leads to efficient, polynomial-time algorithms for problems that appear to require searching through exponentially many possibilities.

Like the divide-and-conquer method, dynamic programming is based on the observation that many optimization problems can be solved by solving similar subproblems and the composing the solutions of those subproblems into a solution for the original problem. In addition, the problem is viewed as

© 2004 by Taylor & Francis Group, LLC

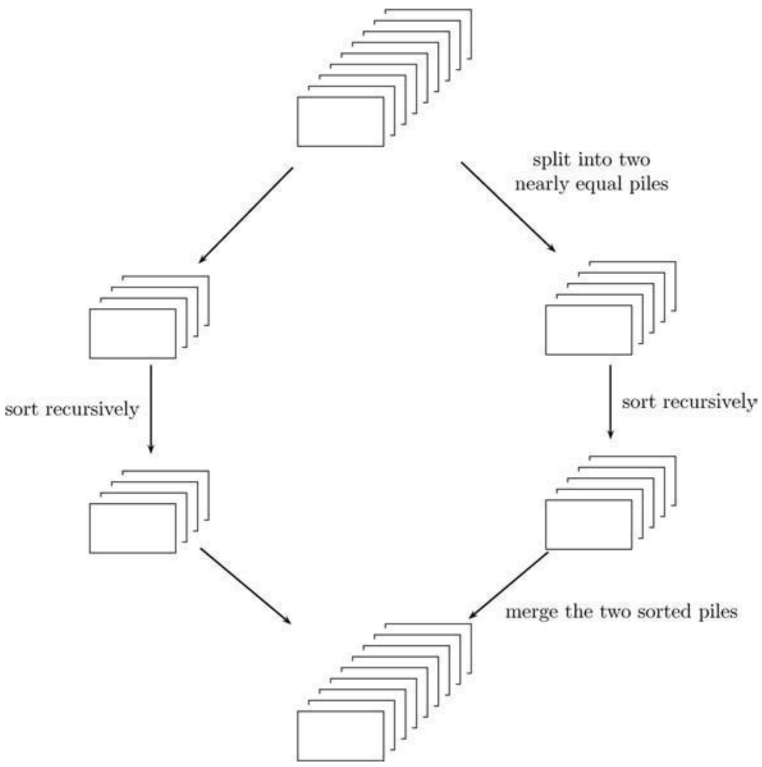


FIGURE 3.2 Schematic description of merge sort.

a sequence of decisions, each decision leading to different subproblems; if a wrong decision is made, a suboptimal solution results, so all possible decisions need to be accounted for.

As an example of dynamic programming, consider the problem of constructing an optimal search pattern for probing an ordered sequence of elements. The problem is similar to searching an array. In the previous section we described binary search, in which an interval in an array is repeatedly bisected until the search ends. Now, however, suppose we know the frequencies with which the search will seek various elements (both in the sequence and missing from it). For example, if we know that the last few elements in the sequence are frequently sought — binary search does not make use of this information — it might be more efficient to begin the search at the right end of the array, not in the middle. Specifically, we are given an ordered sequence $x_1 < x_2 < \cdots < x_n$ and associated frequencies of access $\beta_1, \beta_2, \ldots, \beta_n$, respectively; furthermore, we are given $\alpha_0, \alpha_1, \ldots, \alpha_n$ where α_i is the frequency with which the search will fail because the object sought, z, was missing from the sequence, $x_i < z < x_{i+1}$ (with the obvious meaning when $i = 0$ or $i = n$). What is the optimal order to search for an unknown element z? In fact, how should we describe the optimal search order?

We express a search order as a *binary search tree*, a diagram showing the sequence of probes made in every possible search. We place at the root of the tree the sequence element at which the first probe is made, for example, x_i; the left subtree of x_i is constructed recursively for the probes made when $z < x_i$, and the right subtree of x_i is constructed recursively for the probes made when $z > x_i$. We label each item in the tree with the frequency that the search ends at that item. Figure 3.3 shows a simple example. The search of sequence $x_1 < x_2 < x_3 < x_4 < x_5$ according the tree of Figure 3.3 is done by comparing the unknown element z with x_4 (the root); if $z = x_4$, the search ends. If $z < x_2$, z is compared with x_2 (the root of the left subtree); if $z = x_2$, the search ends. Otherwise, if $z < x_2$, z is compared with x_1 (the root of the left subtree of x_2); if $z = x_1$, the search ends. Otherwise, if $z < x_1$, the search ends unsuccessfully at the leaf labeled α_0. Other results of comparisons lead along other paths in the tree from the root downward. By its

© 2004 by Taylor & Francis Group, LLC

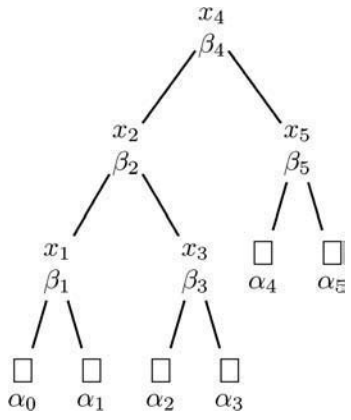


FIGURE 3.3 A binary search tree.

nature, a binary search tree is *lexicographic* in that for all nodes in the tree, the elements in the left subtree of the node are smaller and the elements in the right subtree of the node are larger than the node.

Because we are to find an optimal search pattern (tree), we want the cost of searching to be minimized. The cost of searching is measured by the *weighted path length* of the tree:

$$\sum_{i=1}^{n} \beta_i \times [1 + \text{level}(\beta_i)] + \sum_{i=0}^{n} \alpha_i \times \text{level}(\alpha_i),$$

defined formally as

$$W(\square) = 0,$$
$$W\left(T = \overset{\bullet}{\overset{\wedge}{T_l \; T_r}} \right) = W(T_l) + W(T_r) + \sum \alpha_i + \sum \beta_i,$$

where the summations $\sum \alpha_i$ and $\sum \beta_i$ are over all α_i and β_i in T. Because there are exponentially many possible binary trees, finding the one with minimum weighted path length could, if done naïvely, take exponentially long.

The key observation we make is that a *principle of optimality* holds for the cost of binary search trees: subtrees of an optimal search tree must themselves be optimal. This observation means, for example, that if the tree shown in Figure 3.3 is optimal, then its left subtree must be the optimal tree for the problem of searching the sequence $x_1 < x_2 < x_3$ with frequencies $\beta_1, \beta_2, \beta_3$ and $\alpha_0, \alpha_1, \alpha_2, \alpha_3$. (If a subtree in Figure 3.3 were *not* optimal, we could replace it with a better one, reducing the weighted path length of the entire tree because of the recursive definition of weighted path length.) In general terms, the principle of optimality states that subsolutions of an optimal solution must themselves be optimal.

The optimality principle, together with the recursive definition of weighted path length, means that we can express the construction of an optimal tree recursively. Let $C_{i,j}$, $0 \leq i \leq j \leq n$, be the cost of an optimal tree over $x_{i+1} < x_{i+2} < \cdots < x_j$ with the associated frequencies $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_j$ and $\alpha_i, \alpha_{i+1}, \ldots, \alpha_j$. Then,

$$C_{i,i} = 0,$$
$$C_{i,j} = \min_{i<k\leq j}(C_{i,k-1} + C_{k,j}) + W_{i,j},$$

where

$$W_{i,i} = \alpha_i,$$
$$W_{i,j} = W_{i,j-1} + \beta_j + \alpha_j.$$

© 2004 by Taylor & Francis Group, LLC

These two recurrence relations can be implemented directly as recursive functions to compute $C_{0,n}$, the cost of the optimal tree, leading to the following two functions:

```
 1  int W (int i, int j) {
 2    if (i == j)
 3      return alpha[j];
 4    else
 5      return W(i,j-1) + beta[j] + alpha[j];
 6  }
 7
 8  int C (int i, int j) {
 9    if (i == j)
10      return 0;
11    else {
12      int minCost = MAXINT;
13      int cost;
14      for (int k = i+1; k <= j; k++) {
15        cost = C(i,k-1) + C(k,j) + W(i,j);
16        if (cost < minCost)
17          minCost = cost;
18      }
19      return minCost;
20    }
21  }
```

These two functions correctly compute the cost of an optimal tree; the tree itself can be obtained by storing the values of `k` when `cost < minCost` in line 16.

However, the above functions are unnecessarily time consuming (requiring exponential time) because the same subproblems are solved repeatedly. For example, each call `W(i,j)` uses time $\Theta($`j` $-$ `i`$)$ and such calls are made repeatedly for the same values of `i` and `j`. We can make the process more efficient by caching the values of `W(i,j)` in an array as they are computed and using the cached values when possible:

```
int W[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    W[i][j] = MAXINT;

int W (int i, int j) {
  if (W[i][j] = MAXINT)
    if (i == j)
       W[i][j] = alpha[j];
    else
      W[i][j] =  W(i,j-1) + beta[j] + alpha[j];
  return W[i][j];
}
```

In the same way, we should cache the values of `C(i,j)` in an array as they are computed:

```
int C[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    C[i][j] = MAXINT;

```

© 2004 by Taylor & Francis Group, LLC

```
int C (int i, int j) {
  if (C[i][j] == MAXINT)
    if (i == j)
       C[i][j] = 0;
    else {
      int minCost = MAXINT;
      int cost;
      for (int k = i+1; k <= j; k++) {
        cost = C(i,k-1) + C(k,j) + W(i,j);
        if (cost < minCost)
          minCost = cost;
      }
      C[i][j] = minCost;
    }
  return C[i][j];
}
```

The idea of caching the solutions to subproblems is crucial to making the algorithm efficient. In this case, the resulting computation requires time $\Theta(n^3)$; this is surprisingly efficient, considering that an optimal tree is being found from among exponentially many possible trees.

By studying the pattern in which the arrays `C` and `W` are filled in, we see that the main diagonal `C[i][i]` is filled in first, then the first upper super-diagonal `C[i][i+1]`, then the second upper super-diagonal `C[i][i+2]`, and so on until the upper-right corner of the array is reached. Rewriting the code to do this directly, and adding an array `R[][]` to keep track of the roots of subtrees, we obtain:

```
int W[n][n];
int R[n][n];
int C[n][n];

// Fill in main diagonal
for (int i = 0; i < n; i++) {
  W[i][i] = alpha[i];
  R[i][i] = 0;
  C[i][i] = 0;
}

int minCost, cost;
for (int d = 1; d < n; d++)
  // Fill in d-th upper super-diagonal
  for (i = 0; i < n-d; i++) {
    W[i][i+d] = W[i][i+d-1] + beta[i+d] + alpha[i+d];
    R[i][i+d] = i+1;
    C[i][i+d] = C[i][i] + C[i+1][i+d] + W[i][i+d];
    for (int k = i+2; k <= i+d; k++) {
      cost = C[i][k-1] + C[k][i+d] + W[i][i+d];
      if (cost < C[i][i+d]) {
        R[i][i+d] = k;
        C[i][i+d] = cost;
      }
    }
  }
```

which more clearly shows the $\Theta(n^3)$ behavior.

© 2004 by Taylor & Francis Group, LLC

As a second example of dynamic programming, consider the *traveling salesman problem* in which a salesman must visit n cities, returning to his starting point, and is required to minimize the cost of the trip. The cost of going from city i to city j is $C_{i,j}$. To use dynamic programming we must specify an optimal tour in a recursive framework, with subproblems resembling the overall problem. Thus we define

$$T(i; j_1, j_2, \ldots, j_k) = \begin{cases} \text{cost of an optimal tour from city } i \text{ to city} \\ 1 \text{ that goes through each of the cities } j_1, \\ j_2, \ldots, j_k \text{ exactly once, in any order, and} \\ \text{through no other cities.} \end{cases}$$

The principle of optimality tells us that

$$T(i; j_1, j_2, \ldots, j_k) = \min_{1 \le m \le k} \{C_{i,j_m} + T(j_m; j_1, j_2, \ldots, j_{m-1}, j_{m+1}, \ldots, j_k)\},$$

where, by definition,

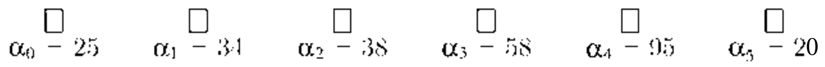
$$T(i; j) = C_{i,j} + C_{j,1}.$$

We can write a function `T` that directly implements the above recursive definition, but as in the optimal search tree problem, many subproblems would be solved repeatedly, leading to an algorithm requiring time $\Theta(n!)$. By caching the values $T(i; j_1, j_2, \ldots, j_k)$, we reduce the time required to $\Theta(n^2 2^n)$, still exponential, but considerably less than without caching.

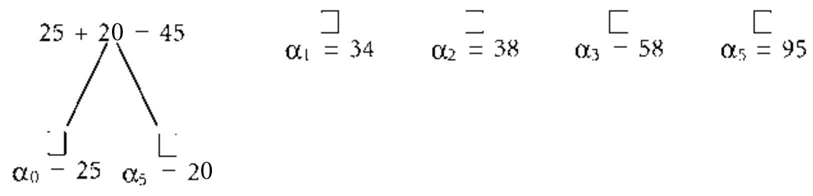
3.6 Greedy Heuristics

Optimization problems always have an objective function to be minimized or maximized, but it is not often clear what steps to take to reach the optimum value. For example, in the optimum binary search tree problem of the previous section, we used dynamic programming to systematically examine all possible trees. But perhaps there is a simple rule that leads directly to the best tree; say, by choosing the largest β_i to be the root and then continuing recursively. Such an approach would be less time-consuming than the $\Theta(n^3)$ algorithm we gave, but it does not necessarily give an optimum tree (if we follow the rule of choosing the largest β_i to be the root, we get trees that are no better, on the average, than a randomly chosen trees). The problem with such an approach is that it makes decisions that are *locally optimum*, although perhaps not *globally optimum*. But such a "greedy" sequence of locally optimum choices does lead to a globally optimum solution in some circumstances.

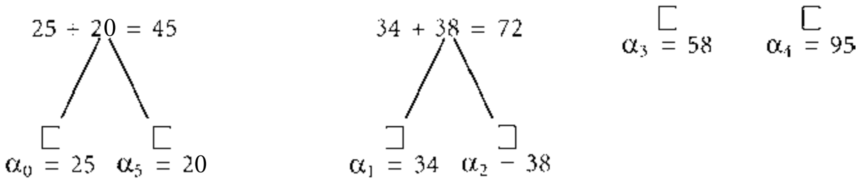
Suppose, for example, $\beta_i = 0$ for $1 \le i \le n$, and we remove the lexicographic requirement of the tree; the resulting problem is the determination of an optimal prefix code for $n + 1$ letters with frequencies $\alpha_0, \alpha_1, \ldots, \alpha_n$. Because we have removed the lexicographic restriction, the dynamic programming solution of the previous section no longer works, but the following simple greedy strategy yields an optimum tree: repeatedly combine the two lowest-frequency items as the left and right subtrees of a newly created item whose frequency is the sum of the two frequencies combined. Here is an example of this construction; we start with five leaves with weights



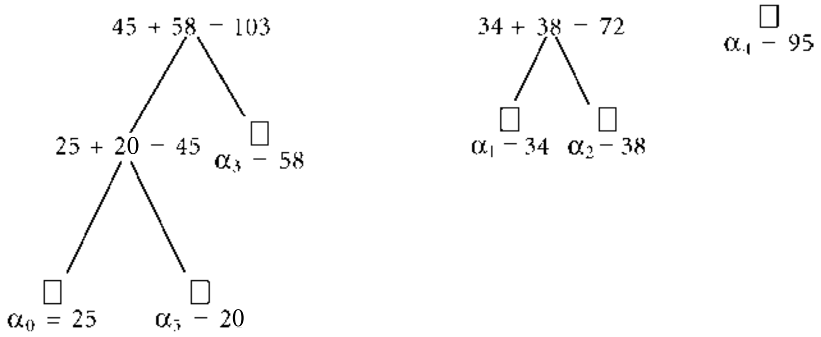
First, combine leaves $\alpha_0 = 25$ and $\alpha_5 = 20$ into a subtree of frequency $25 + 20 = 45$:



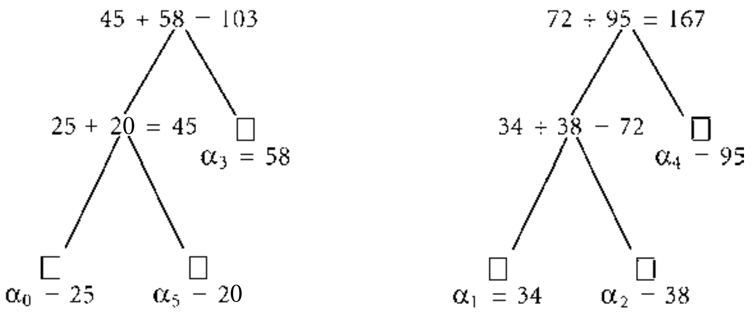
Then combine leaves $\alpha_1 = 34$ and $\alpha_2 = 38$ into a subtree of frequency $34 + 38 = 72$:



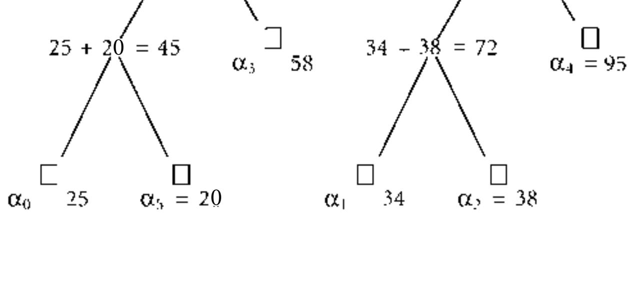
Next, combine the subtree of frequency $\alpha_0 + \alpha_5 = 45$ with $\alpha_3 = 58$:



Then combine the subtree of frequency $\alpha_1 + \alpha_2 = 72$ with $\alpha_4 = 95$:

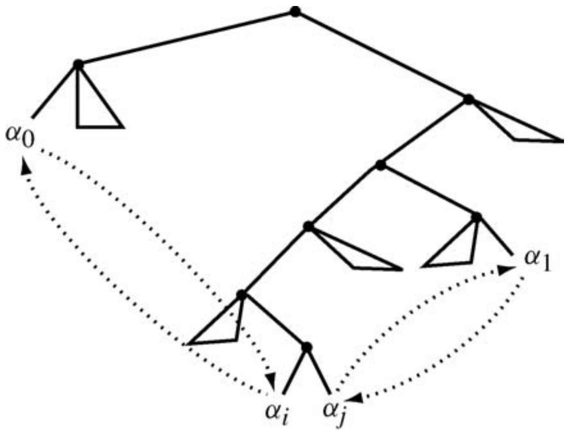


Finally, combine the only two remaining subtrees:



© 2004 by Taylor & Francis Group, LLC

How do we know that the above-outlined process leads to an optimum tree? The key to proving that the tree is optimum is to assume, by way of contradiction, that it is not optimum. In this case, the greedy strategy must have erred in one of its choices, so let's look at the *first* error this strategy made. Because all previous greedy choices were not errors, and hence lead to an optimum tree, we can assume that we have a sequence of frequencies $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that the first greedy choice is erroneous — without loss of generality assume that α_0 and α_1 are two smallest frequencies, those combined erroneously by the greedy strategy. For this combination to be erroneous, there must be no optimum tree in which these two leaves are siblings, so consider an optimum tree, the locations of α_0 and α_1, and the location of the two deepest leaves in the tree, α_i and α_j:



By interchanging the positions of α_0 and α_i and α_1 and α_j (as shown), we obtain a tree in which α_0 and α_1 are siblings. Because α_0 and α_1 are the two lowest frequencies (because they were the greedy algorithm's choice) $\alpha_0 \le \alpha_i$ and $\alpha_1 \le \alpha_j$, the weighted path length of the modified tree is no larger than before the modification since $\text{level}(\alpha_0) \ge \text{level}(\alpha_i)$, $\text{level}(\alpha_1) \ge \text{level}(\alpha_j)$ and, hence,

$$\text{level}(\alpha_i) \times \alpha_0 + \text{level}(\alpha_j) \times \alpha_1 \le \text{level}(\alpha_0) \times \alpha_0 + \text{level}(\alpha_1) \times \alpha_1.$$

In other words, the first so-called mistake of the greedy algorithm was in fact not a mistake because there is an optimum tree in which α_0 and α_1 are siblings. Thus we conclude that the greedy algorithm never makes a first mistake — that is, it never makes a mistake at all!

The greedy algorithm above is called *Huffman's algorithm.* If the subtrees are kept on a priority queue by cumulative frequency, the algorithm needs to insert the $n + 1$ leaf frequencies onto the queue, and then repeatedly remove the two least elements on the queue, unite those to elements into a single subtree, and put that subtree back on the queue. This process continues until the queue contains a single item, the optimum tree. Reasonable implementations of priority queues will yield $O(n \log n)$ implementations of Huffman's greedy algorithm.

The idea of making greedy choices, facilitated with a priority queue, works to find optimum solutions to other problems too. For example, a spanning tree of a weighted, connected, undirected graph $G = (V, E)$ is a subset of $|V| - 1$ edges from E connecting all the vertices in G; a spanning tree is minimum if the sum of the weights of its edges is as small as possible. *Prim's algorithm* uses a sequence of greedy choices to determine a minimum spanning tree: start with an arbitrary vertex $v \in V$ as the spanning-tree-to-be. Then, repeatedly add the cheapest edge connecting the spanning-tree-to-be to a vertex not yet in it. If the vertices not yet in the tree are stored in a priority queue implemented by a Fibonacci heap, the total time required by Prim's algorithm will be $O(|E| + |V| \log |V|)$. But why does the sequence of greedy choices lead to a minimum spanning tree?

© 2004 by Taylor & Francis Group, LLC

Suppose Prim's algorithm does *not* result in a minimum spanning tree. As we did with Huffman's algorithm, we ask what the state of affairs must be when Prim's algorithm makes its first mistake; we will see that the assumption of a first mistake leads to a contradiction, thus proving the correctness of Prim's algorithm. Let the edges added to the spanning tree be, in the order added, $e_1, e_2, e_3, \ldots$, and let e_i be the first mistake. In other words, there is a minimum spanning tree $T_{\min}$ containing $e_1, e_2, \ldots, e_{i-1}$, but no minimum spanning tree contains $e_1, e_2, \ldots, e_i$. Imagine what happens if we add the edge e_i to $T_{\min}$: because $T_{\min}$ is a spanning tree, the addition of e_i causes a cycle containing e_i. Let $e_{\max}$ be the highest-cost edge on that cycle. Because Prim's algorithm makes a greedy choice — that is, chooses the lowest cost available edge — the cost of $e_{\max}$ is at least that of e_i, so the cost of the spanning $T_{\min} - \{e_{\max}\} \cup \{e_i\}$ is at most that of $T_{\min}$; in other words, $T_{\min} - \{e_{\max}\} \cup \{e_i\}$ is also a minimum spanning tree, contradicting our assumption that the choice of e_i is the first mistake. Therefore, the spanning tree constructed by Prim's algorithm must be a minimum spanning tree.

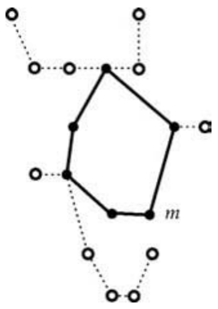
We can apply the greedy heuristic to many optimization problems, and even if the results are not optimal, they are often quite good. For example, in the n-city traveling salesman problem, we can get near-optimal tours in time $O(n^2)$ when the intercity costs are symmetric ($C_{i,j} = C_{j,i}$ for all i and j) and satisfy the triangle inequality ($C_{i,j} \leq C_{i,k} + C_{k,j}$ for all i, j, and k). The *closest insertion algorithm* starts with a "tour" consisting of a single, arbitrarily chosen city, and successively inserts the remaining cities to the tour, making a greedy choice about which city to insert next and where to insert it: the city chosen for insertion is the city not on the tour but closest to a city on the tour; the chosen city is inserted adjacent to the city on the tour to which it is closest.

Given an $n \times n$ symmetric distance matrix C that satisfies the triangle inequality, let I_n be the tour of length $|I_n|$ produced by the closest insertion heuristic and let O_n be an optimal tour of length $|O_n|$. Then,

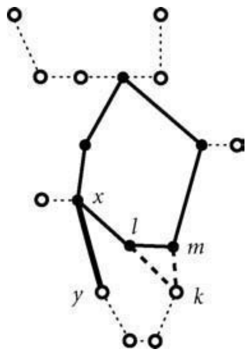
$$\frac{|I_n|}{|O_n|} < 2.$$

This bound is proved by an incremental form of the optimality proofs for greedy heuristics we saw seen above: we ask not where the first error is, but by how much we are in error at each greedy insertion to the tour; we establish a correspondence between edges of the optimal tour and cities inserted on the closest insertion tour. We show that at each insertion of a new city to the closest insertion tour, the cost of that insertion is at most twice the cost of corresponding edge of the optimal tour.

To establish the correspondence, imagine the closest insertion algorithm keeping track not only of the current tour, but also of a spider-like configuration including the edges of the current tour (the body of the spider) and pieces of the optimal tour (the legs of the spider). We show the current tour in solid lines and the pieces of optimal tour as dotted lines:



Initially, the spider consists of the arbitrarily chosen city with which the closest insertion tour begins and the legs of the spider consist of all the edges of the optimal tour *except* for one edge eliminated arbitrarily. As each city is inserted into the closest insertion tour, the algorithm will delete from the spider-like configuration one of the dotted edges from the optimal tour. When city k is inserted between cities l and m, the edge deleted is the one attaching spider to the leg containing the city inserted (from city x to city y), shown here in bold:



Now,

$$C_{k,m} \leq C_{x,y}$$

because of the greedy choice to add city k to the tour and not city y. By the triangle inequality,

$$C_{l,k} \leq C_{l,m} + C_{m,k},$$

and by symmetry, we can combine these two inequalities to get

$$C_{l,k} \leq C_{l,m} + C_{x,y}.$$

Adding this last inequality to the first one above,

$$C_{l,k} + C_{k,m} \leq C_{l,m} + 2C_{x,y},$$

that is,

$$C_{l,k} + C_{k,m} - C_{l,m} \leq 2C_{x,y}.$$

Thus, adding city k between cities l and m adds no more to I_n than $2C_{x,y}$. Summing these incremental, amounts over the cost of the entire algorithm tells us that

$$I_n \leq 2O_n,$$

as we claimed.

References

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, McGraw-Hill, New York, 2nd ed., 2001.

Greene, D. H. and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, 3rd ed., Birkhäuser, Boston, 1990.

Knuth, D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* Addison-Wesley, Reading, MA, 3rd ed., 1997.

Knuth, D. E., *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, MA, 2nd ed., 1998.

Lueker, G. S., "Some techniques for solving recurrences," *Computing Surveys*, **12**, 419–436, 1980.

Reingold, E. M. and W. J. Hansen, *Data Structures in Pascal*, Little, Brown and Company, Boston, 1986.

Reingold, E. M., J. Nievergelt, and N. Deo, *Combinatorial Algorithms*: *Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1977.

© 2004 by Taylor & Francis Group, LLC

4 Data Structures

Roberto Tamassia
Brown University

Bryan M. Cantrill
Sun Microsystems, Inc.

4.1 Introduction

The study of data structures — that is, methods for organizing data that are suitable for computer processing — is one of the classic topics of computer science. At the hardware level, a computer views storage devices such as internal memory and disk as holders of elementary data units (bytes), each accessible through its address (an integer). When writing programs, instead of manipulating the data at the byte level, it is convenient to organize them into higher-level entities called *data structures*.

4.1.1 Containers, Elements, and Positions or Locators

Most data structures can be viewed as **containers** that store a collection of objects of a given type, called the *elements* of the container. Often, a total order is defined among the elements (e.g., alphabetically ordered names, points in the plane ordered by x-coordinate). Following the approach of Goodrich and Tamassia [2001], we assume that the elements of a container can be accessed by means of variables called **positions** or **locators**. When an object is inserted into the container, a position or locator is returned, which can be later used to access or delete the object. A position represents a "place" where an element is stored, Examples of positions are array cells and list nodes. A locator "tracks" the position of an element in the data structure as it changes over time. A locator is typically implemented with an object that stores a pointer to a position.

A data structure has an associated repertory of operations, classified into *queries*, which retrieve information on the data structure (e.g., return the number of elements, or test the presence of a given element), and *updates*, which modify the data structure (e.g., insertion and deletion of elements). The performance

© 2004 by Taylor & Francis Group, LLC

of a data structure is characterized by the space requirement and the time complexity of the operations in its repertory. The *amortized* time complexity of an operation is the average time over a suitably defined sequence of operations.

However, efficiency is not the only quality measure of a data structure. Simplicity and ease of implementation should be taken into account when choosing a data structure for solving a practical problem.

4.1.2 Abstract Data Types

Data structures are concrete implementations of **abstract data types** (ADTs). A *data type* is a collection of objects. A data type can be mathematically specified (e.g., real number, directed graph) or concretely specified within a programming language (e.g., **int** in C, **set** in Pascal). An ADT is a mathematically specified data type equipped with operations that can be performed on the objects. Object-oriented programming languages, such as C++, provide support for expressing ADTs by means of *classes*. ADTs specify the data stored and the operations to be performed on them.

4.1.3 Main Issues in the Study of Data Structures

The following issues are of foremost importance in the study of data structures.

4.1.3.1 Static vs. Dynamic

A *static* data structure supports only queries, whereas a *dynamic* data structure also supports updates. A *dynamic* data structure is often more complicated than its static counterpart supporting the same repertory of queries. A *persistent* data structure (see, e.g., Driscoll et al. [1989]) is a dynamic data structure that supports operations on past versions. There are many problems for which no efficient dynamic data structures are known.

4.1.3.2 Implicit vs. Explicit

Two fundamental data organization mechanisms are used in data structures. In an *explicit* data structure, pointers (i.e., memory addresses) are used to link the elements and access them (e.g., a singly linked list, where each element has a pointer to the next one). In an *implicit* data structure (see, e.g., [Munro and Suwanda 1980]), mathematical relationships support the retrieval of elements (e.g., array representation of a heap, see Section 4.3). Explicit data structures must use additional space to store pointers. However, they are more flexible for complex problems. Most programming languages support pointers and basic implicit data structures, such as arrays.

4.1.3.3 Internal vs. External Memory

In a typical computer, there are two levels of memory: internal memory (also called random access memory, i.e., RAM) and external memory (disk). The internal memory is much faster than external memory but has much smaller capacity. Data structures designed to work for data that fit into internal memory may not perform well for large amounts of data that need to be stored in external memory. For large-scale problems, data structures need to be designed that take into account the two levels of memory [Aggarwal and Vitter 1988]. For example, two-level indices such as B-trees [Comer 1979] have been designed to efficiently search in large databases.

4.1.3.4 Space vs. Time

Data structures often exhibit a trade-off between space and time complexity. For example, suppose we want to represent a set of integers in the range $[0, N]$ (e.g., for a set of social security numbers $N = 10^{10} - 1$) such that we can efficiently query whether a given element is in the set, insert an element, or delete an element. Two possible data structures for this problem are an N-element bit array (where the bit in position i indicates the presence of integer i in the set), and a balanced search tree (such as a 2–3 tree or a red–black tree). The bit array has optimal time complexity because it supports queries, insertions, and

© 2004 by Taylor & Francis Group, LLC

deletions in constant time. However, it uses space proportional to the size N of the range, irrespective of the number of elements actually stored. The balanced **search tree** supports queries, insertions, and deletions in logarithmic time but uses optimal space proportional to the current number of elements stored.

4.1.3.5 Theory vs. Practice

A large and ever-growing body of theoretical research on data structures is available, where the performance is measured in asymptotic terms (big-Oh notation). Although asymptotic complexity analysis is an important mathematical subject, it does not completely capture the notion of efficiency of data structures in practical scenarios, where constant factors cannot be disregarded and the difficulty of implementation substantially affects design and maintenance costs. Experimental studies comparing the practical efficiency of data structures for specific classes of problems should be encouraged to bridge the gap between the theory and practice of data structures.

4.1.4 Fundamental Data Structures

The following data structures are ubiquitously used in the description of discrete algorithms, and serve as basic building blocks for realizing more complex data structures. They are covered in detail in the textbooks listed in the "Further Information" section and in the additional references provided.

4.1.4.1 Sequence

A **sequence** is a container that stores elements in a certain linear order, which is imposed by the operations performed. The basic operations supported are retrieving, inserting, and removing an element given its position. Special types of sequences include stacks and queues, where insertions and deletions can be done only at the head or tail of the sequence. The basic realization of sequences are by means of arrays and linked lists. Concatenable queues (see, e.g., Hoffman et al. [1986]) support additional operations such as splitting and splicing, and determining the sequence containing a given element. In external memory, a sequence is typically associated with a file.

4.1.4.2 Priority Queue

A **priority queue** is a container of elements from a totally ordered universe that supports the basic operations of inserting an element and retrieving/removing the largest element. A key application of priority queues is sorting algorithms. A **heap** is an efficient realization of a priority queue that embeds the elements into the ancestor/descendant partial order of a **binary tree**. A heap also admits an implicit realization where the nodes of the tree are mapped into the elements of an array (see Section 4.3). Sophisticated variations of priority queues include min–max heaps, pagodas, deaps, binomial heaps, and Fibonacci heaps. The buffer tree is an efficient external-memory realization of a priority queue.

4.1.4.3 Dictionary

A **dictionary** is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. **Hash tables** provide an efficient implicit realization of a dictionary. Efficient explicit implementations include skip lists [Pugh 1990], tries, and balanced search trees (e.g., **AVL-trees**, red–black trees, 2–3 trees, 2–3–4 trees, weight-balanced trees, biased search trees, splay trees). The technique of fractional cascading [Chazelle and Guibas 1986] speeds up searching for the same element in a collection of dictionaries. In external memory, dictionaries are typically implemented as B-trees and their variations.

The above data structures are widely used in the following application domains:

1. *Graphs and networks:* adjacency matrix, adjacency lists, link-cut tree [Sleator and Tarjan 1983], dynamic expression tree [Cohen and Tamassia 1995], topology tree [Frederickson 1997], SPQR-tree [Di Battista and Tamassia 1996], sparsification tree [Eppstein et al. 1997]. See also, for example, Di Battista et al. [1999], Even [1979], Mehlhorn [1984], and Tarjan [1983].

© 2004 by Taylor & Francis Group, LLC

2. *Text processing:* string, suffix tree, Patricia tree. See, for example, Gonnet and Baeza-Yates [1991].
3. *Geometry and graphics:* binary space partition tree, chain tree, trapezoid tree, range tree, segment tree, interval tree, priority search tree, hull tree, quad tree, R-tree, grid file, metablock tree. For example, see Chiang and Tamassia [1992], Edelsbrunner [1987], Foley et al. [1990], Mehlhorn [1984], Nievergelt and Hinrichs [1993], O'Rourke [1994], and Preparata and Shamos [1985].

4.1.5 Organization of the Chapter

The remainder of this chapter focuses on three fundamental abstract data types: sequences, priority queues, and dictionaries. Examples of efficient data structures and algorithms for implementing them are presented in detail in Section 4.2 through Section 4.4, respectively. Namely, we cover arrays, singly and doubly linked lists, heaps, search trees, (a, b)-trees, AVL-trees, bucket arrays, and hash tables.

4.2 Sequence

4.2.1 Introduction

A *sequence* is a container that stores elements in linear order, which is imposed by the operations performed. The basic operations supported are:

- INSERTRANK : insert an element in a given position.
- REMOVE: remove an element.

Sequences are a basic form of data organization, and are typically used to realize and implement other data types and data structures.

4.2.2 Operations

Using positions (see Section 4.1.1), we can define a more complete repertory of operations for a sequence S:

SIZE(N): return the number of elements N of S.
HEAD(p): assign to p the position of the first element of S; if S is empty, then p is set to null.
TAIL(p): assign to p the position of the last element of S; if S is empty, then p is set to null.
POSITIONRANK (r, p): assign to p the position of the rth element of S; if $r < 1$ or $r > N$, where N is the size of S, then p is set to null.
PREV(p', p''): assign to p'' the position of the element of S preceding the element with position p'; if p' is the position of the first element of S, then p'' is set to null.
NEXT(p', p''): assign to p'' the position of the element of S following the element with position p'; if p' is the position of the last element of S, then p'' is set to null.
INSERTAFTER(e, p', p''): insert element e into S after the element with position p', and return the position p'' of e.
INSERTBEFORE(e, p', p''): insert element e into S before the element with position p', and return the position p'' of e.
INSERTHEAD(e, p): insert element e at the beginning of S, and return the position p of e.
INSERTTAIL(e, p): insert element e at the end of S, and return the position p of e.
INSERTRANK (e, r, p): insert element e in the rth position of S; if $r < 1$ or $r > N + 1$, where N is the current size of S, then p is set to null.
REMOVE(p, e): remove from S and return element e with position p.
MODIFY(p, e): replace with e the element with position p.

Some of the preceding operations can be easily expressed by means of other operations of the repertory. For example, operations HEAD and TAIL can be easily expressed by means of POSITIONRANK and SIZE.

© 2004 by Taylor & Francis Group, LLC

TABLE 4.1 Performance of a Sequence Implemented with an Array

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
POSITIONRANK	$O(1)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(N)$
INSERTBEFORE	$O(N)$
INSERTHEAD	$O(N)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(1)$

TABLE 4.2 Performance of a Sequence Implemented with a Singly Linked List

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
POSITIONRANK	$O(N)$
PREV	$O(N)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(N)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(1)$

4.2.3 Implementation with an Array

The simplest way to implement a sequence is to use a (one-dimensional) array, where the ith element of the array stores the ith element of the list, and to keep a variable that stores the size N of the sequence. With this implementation, accessing elements takes $O(1)$ time, whereas insertions and deletions take $O(N)$ time. Table 4.1 shows the time complexity of the implementation of a sequence by means of an array.

4.2.4 Implementation with a Singly Linked List

A sequence can also be implemented with a singly linked list, where each position has a pointer to the next one. We also store the size of the sequence and pointers to the first and last position of the sequence.

With this implementation, accessing elements by rank takes $O(N)$ time because we need to traverse the list, whereas some insertions and deletions take $O(1)$ time. Table 4.2 shows the time complexity of the implementation of a sequence by means of a singly linked list.

4.2.5 Implementation with a Doubly Linked List

Better performance can be achieved, at the expense of using additional space, by implementing a sequence with a doubly linked list, where each position has pointers to the next and previous positions. We also

© 2004 by Taylor & Francis Group, LLC

TABLE 4.3 Performance of a Sequence Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
POSITIONRANK	$O(N)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(1)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(1)$

store the size of the sequence and pointers to the first and last positions of the sequence. Table 4.3 shows the time complexity of the implementation of sequence by means of a doubly linked list.

4.3 Priority Queue

4.3.1 Introduction

A priority queue is a container of elements from a totally ordered universe that supports the following two basic operations:

1. INSERT: insert an element into the priority queue.
2. REMOVEMAX: remove the largest element from the priority queue.

Here are some simple applications of a priority queue:

- *Scheduling.* A scheduling system can store the tasks to be performed into a priority queue, and select the task with highest priority to be executed next.
- *Sorting.* To sort a set of N elements, we can insert them one at a time into a priority queue by means of N INSERT operations, and then retrieve them in decreasing order by means of N REMOVEMAX operations. This two-phase method is the paradigm of several popular sorting algorithms, including *selection sort*, *insertion sort*, and *heap-sort*.

4.3.2 Operations

Using locators, we can define a more complete repertory of operations for a priority queue Q:

SIZE(N): return the current number of elements N in Q.
MAX(c): return a locator c to the maximum element of Q.
INSERT(e, c): insert element e into Q and return a locator c to e.
REMOVE(c, e): remove from Q and return element e with locator c.
REMOVEMAX(e): remove from Q and return the maximum element e from Q.
MODIFY(c, e): replace with e the element with locator c.

Note that operation REMOVEMAX(e) is equivalent to MAX(c) followed by REMOVE(c, e).

© 2004 by Taylor & Francis Group, LLC

TABLE 4.4 Performance of a Priority Queue Realized by an Unsorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
REMOVEMAX	$O(N)$
MODIFY	$O(1)$

TABLE 4.5 Performance of a Priority Queue Realized by a Sorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
REMOVEMAX	$O(1)$
MODIFY	$O(N)$

4.3.3 Realization with a Sequence

We can realize a priority queue by reusing and extending the sequence abstract data type (see Section 4.2). Operations SIZE, MODIFY, and REMOVE correspond to the homonymous sequence operations.

4.3.3.1 Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation MAX can be performed by scanning the sequence with an iteration of NEXT operations, keeping track of the maximum element encountered. Finally, as observed earlier, operation REMOVEMAX is a combination of MAX and REMOVE. Table 4.4 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

4.3.3.2 Sorted Sequence

An alternative implementation uses a sequence that is kept sorted. In this case, operation MAX corresponds to simply accessing the last element of the sequence. However, operation INSERT now requires scanning the sequence to find the appropriate position to insert the new element. Table 4.5 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

Realizing a priority queue with a sequence, sorted or unsorted, has the drawback that some operations require linear time in the worst case. Hence, this realization is not suitable in many applications where fast running times are sought for all the priority queue operations.

4.3.3.3 Sorting

For example, consider the sorting application (see the first introduction to this section). We have a collection of N elements from a totally ordered universe, and we want to sort them using a priority queue Q. We

© 2004 by Taylor & Francis Group, LLC

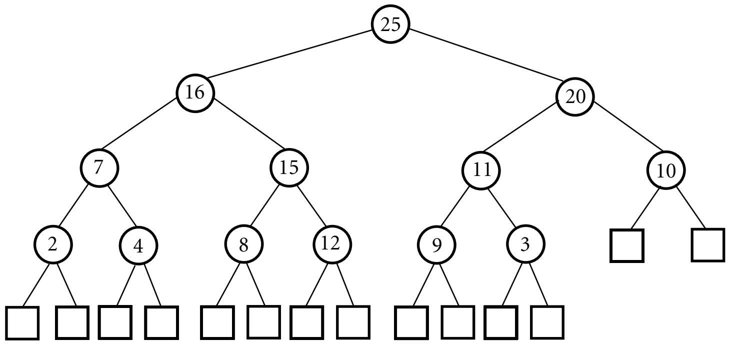


FIGURE 4.1 Example of a heap storing 13 elements.

assume that each element uses $O(1)$ space, and any two elements can be compared in $O(1)$ time. If we realize Q with an unsorted sequence, then the first phase (inserting the N elements into Q) takes $O(N)$ time. However, the second phase (removing N times the maximum element) takes time

$$O\left(\sum_{i=1}^{N} i\right) = O(N^2)$$

Hence, the overall time complexity is $O(N^2)$. This sorting method is known as *selection sort.*

However, if we realize the priority queue with a sorted sequence, then the first phase takes time

$$O\left(\sum_{i=1}^{N} i\right) = O(N^2)$$

while the second phase takes time $O(N)$. Again, the overall time complexity is $O(N^2)$. This sorting method is known as *insertion sort.*

4.3.4 Realization with a Heap

A more sophisticated realization of a priority queue uses a data structure called a *heap*. A heap is a binary tree T whose internal nodes each store one element from a totally ordered universe, with the following properties (see Figure 4.1):

Level property. All of the levels of T are full, except possibly for the bottommost level, which is left filled.

Partial order property. Let μ be a node of T distinct from the root, and let ν be the parent of μ; then the element stored at μ is less than or equal to the element stored at ν.

The leaves of a heap do not store data and serve only as placeholders. The level property implies that heap T is a minimum-height binary tree. More precisely, if T stores N elements and has height h, then each level i with $0 \leq i \leq h - 2$ stores exactly 2^i elements, whereas level $h - 1$ stores between 1 and 2^{h-1} elements. Note that level h contains only leaves. We have

$$2^{h-1} = 1 + \sum_{i=0}^{h-2} 2^i \leq N \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

from which we obtain:

$$\log_2(N + 1) \leq h \leq 1 + \log_2 N$$

© 2004 by Taylor & Francis Group, LLC

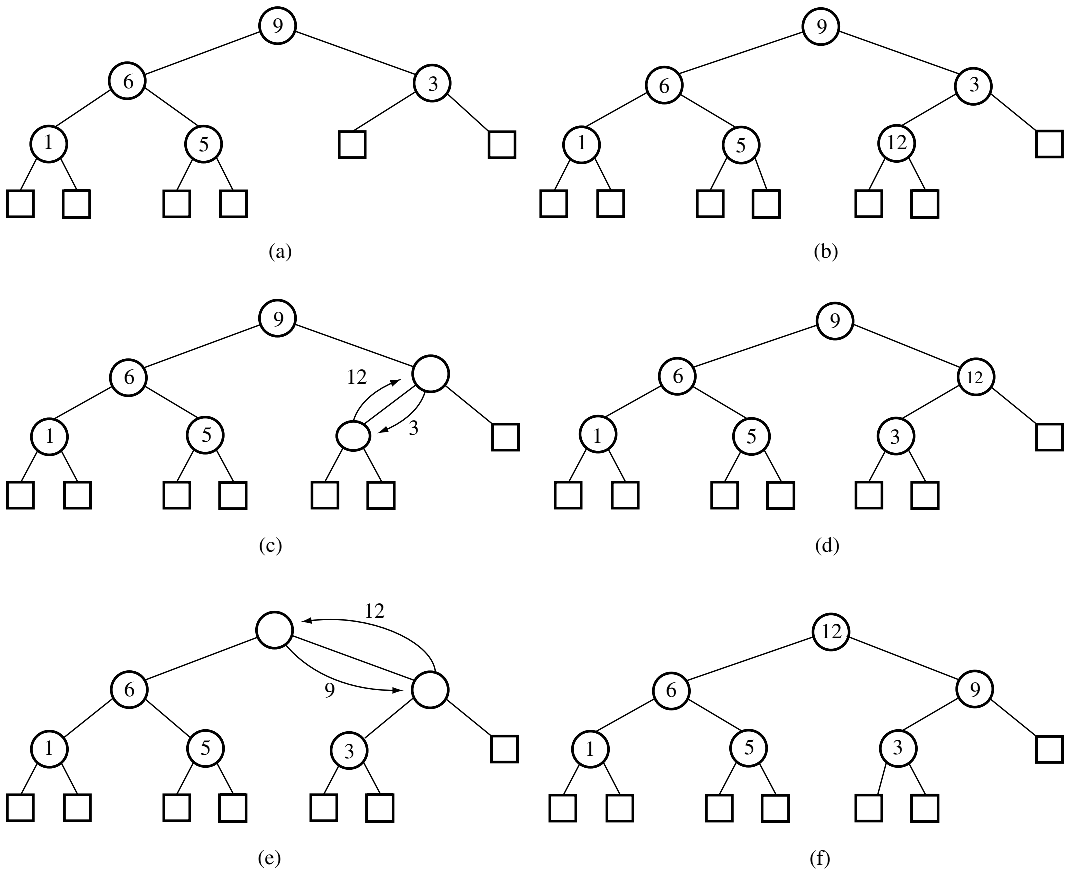


FIGURE 4.2 Operation INSERT in a heap.

Now we show how to perform the various priority queue operations by means of a heap T. We denote with $x(\mu)$ the element stored at an internal node μ of T. We denote with ρ the root of T. We call the *last node* of T the rightmost internal node of the bottommost internal level of T.

By storing a counter that keeps track of the current number of elements, SIZE consists of simply returning the value of the counter. By the partial order property, the maximum element is stored at the root and, hence, operation MAX can be performed by accessing node ρ.

4.3.4.1 Operation INSERT

To insert an element e into T, we add a new internal node μ to T such that μ becomes the new last node of T, and set $x(\mu) = e$. This action ensures that the level property is satisfied, but may violate the partial order property. Hence, if $\mu \neq \rho$, we compare $x(\mu)$ with $x(\nu)$, where ν is the parent of μ. If $x(\mu) > x(\nu)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at μ and ν. This causes the new element e to move up one level. Again, the partial order property may be violated, and we may have to continue moving up the new element e until no violation occurs. In the worst case, the new element e moves up to the root ρ of T by means of $O(\log N)$ exchanges. The upward movement of element e by means of exchanges is conventionally called *upheap*.

An example of a sequence of insertions into a heap is shown in Figure 4.2.

© 2004 by Taylor & Francis Group, LLC

4.3.4.2 Operation REMOVEMAX

To remove the maximum element, we cannot simply delete the root of T, because this would disrupt the binary tree structure. Instead, we access the last node λ of T, copy its element e to the root by setting $x(\rho) = x(\lambda)$, and delete λ. We have preserved the level property, but we may have violated the partial order property. Hence, if ρ has at least one nonleaf child, we compare $x(\rho)$ with the maximum element $x(\sigma)$ stored at a child of ρ. If $x(\rho) < x(\sigma)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at ρ and σ. Again, the partial order property may be violated, and we continue moving down element e until no violation occurs. In the worst case, element e moves down to the bottom internal level of T by means of $O(\log N)$ exchanges. The downward movement of element e by means of exchanges is conventionally called *downheap*.

An example of operation REMOVEMAX in a heap is shown in Figure 4.3.

4.3.4.3 Operation REMOVE

To remove an arbitrary element of heap T, we cannot simply delete its node μ, because this would disrupt the binary tree structure. Instead, we proceed as before and delete the last node of T after copying to μ its element e. We have preserved the level property, but we may have violated the partial order property, which can be restored by performing either upheap or downheap.

Finally, after modifying an element of heap T, if the partial order property is violated, we just need to perform either upheap or downheap.

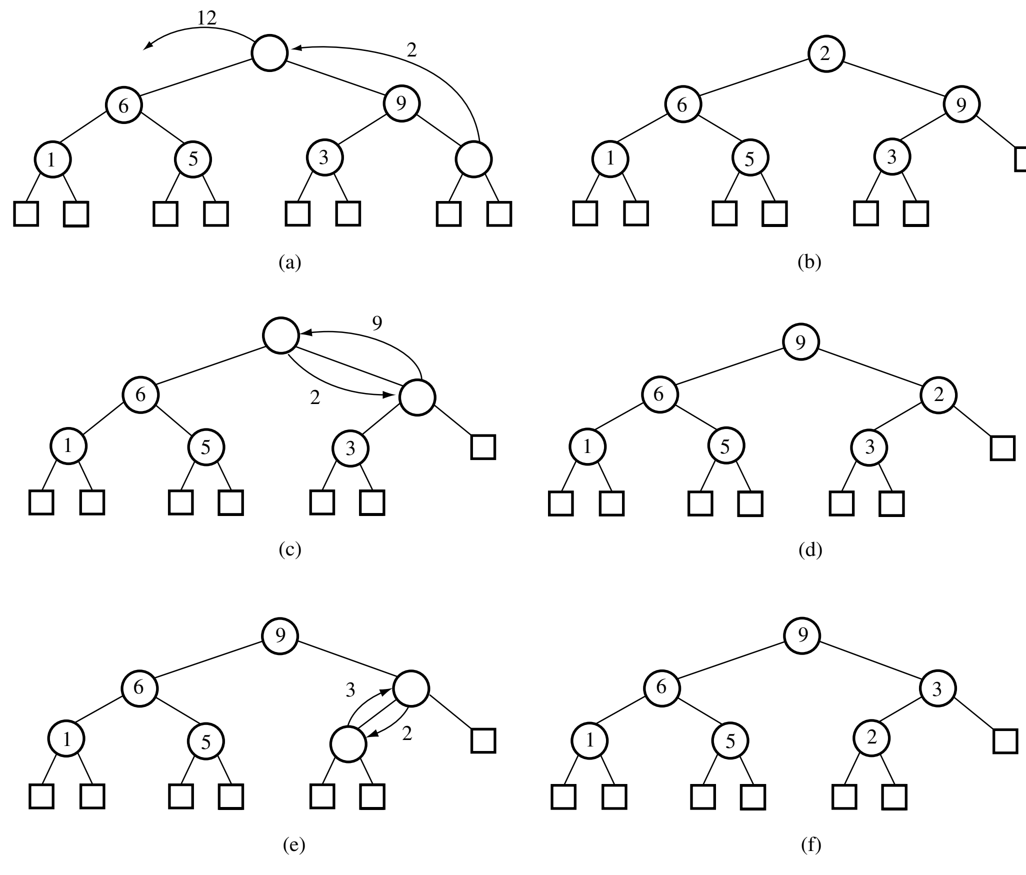


FIGURE 4.3 Operation REMOVEMAX in a heap.

© 2004 by Taylor & Francis Group, LLC

TABLE 4.6 Performance of a Priority Queue Realized by a Heap, Implemented with a Suitable Binary Tree Data Structure

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
REMOVEMAX	$O(\log N)$
MODIFY	$O(\log N)$

4.3.4.4 Time Complexity

Table 4.6 shows the time complexity of the realization of a priority queue by means of a heap. In the table we denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$. We assume that the heap is itself realized by a data structure for binary trees that supports $O(1)$-time access to the children and parent of a node. For instance, we can implement the heap explicitly with a linked structure (with pointers from a node to its parents and children), or implicitly with an array (where node i has children $2i$ and $2i + 1$). Let N be the number of elements in a priority queue Q realized with a heap T at the time an operation is performed. The time bounds of Table 4.6 are based on the following facts:

- In the worst case, the time complexity of upheap and downheap is proportional to the height of T.
- If we keep a pointer to the last node of T, we can update this pointer in time proportional to the height of T in operations INSERT, REMOVE, and REMOVEMAX, as illustrated in Figure 4.4.
- The height of heap T is $O(\log N)$.

The $O(N)$ space complexity bound for the heap is based on the following facts:

- The heap has $2N + 1$ nodes (N internal nodes and $N + 1$ leaves).
- Every node uses $O(1)$ space.
- In the array implementation, because of the level property, the array elements used to store heap nodes are in the contiguous locations 1 through $2N - 1$.

Note that we can reduce the space requirement by a constant factor implementing the leaves of the heap with null objects, such that only the internal nodes have space associated with them.

4.3.4.5 Sorting

Realizing a priority queue with a heap has the advantage that all of the operations take $O(\log N)$ time, where N is the number of elements in the priority queue at the time the operation is performed. For example, in the sorting application (see Section 4.3.1), both the first phase (inserting the N elements) and the second phase (removing N times the maximum element) take time

$$O\left(\sum_{i=1}^{N} \log i\right) = O(N \log N)$$

Hence, sorting with a priority queue realized with a heap takes $O(N \log N)$ time. This sorting method is known as *heap sort*, and its performance is considerably better than that of selection sort and insertion sort (see Section 4.3.3.3), where the priority queue is realized as a sequence.

© 2004 by Taylor & Francis Group, LLC

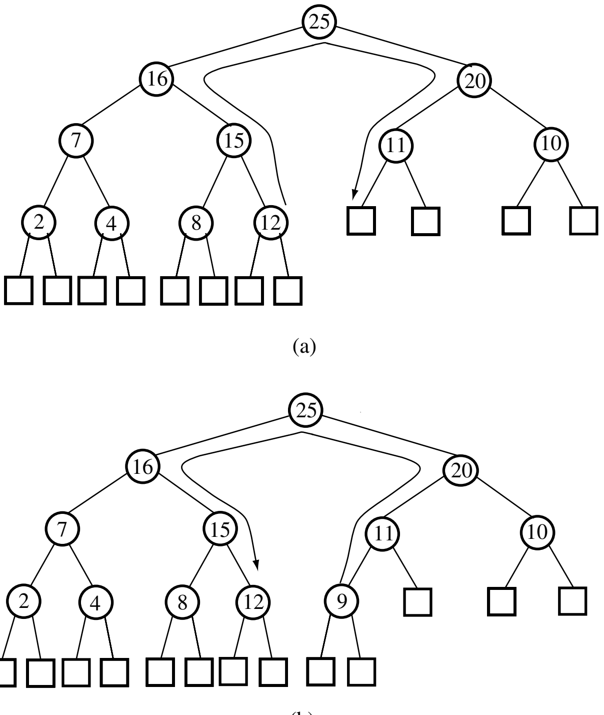


FIGURE 4.4 Update of the pointer to the last node: (a) INSERT and (b) REMOVE or REMOVEMAX.

4.3.5 Realization with a Dictionary

A priority queue can be easily realized with a dictionary (see Section 4.4). Indeed, all of the operations in the priority queue repertory are supported by a dictionary. To achieve $O(1)$ time for operation MAX, we can store the locator of the maximum element in a variable, and recompute it after an update operation. This realization of a priority queue with a dictionary has the same asymptotic complexity bounds as the realization with a heap, provided the dictionary is suitably implemented, for example, with an (a, b)-tree (see section "Realization with an (a, b)-tree") or an AVL-tree (see section "Realization with an AVL-tree"). However, a heap is simpler to program than an (a, b)-tree or an AVL-tree.

4.4 Dictionary

A dictionary is a container of elements from a totally ordered universe that supports the following basic operations:

- FIND: search for an element.
- INSERT: insert an element.
- REMOVE: delete an element.

A major application of dictionaries is database systems.

4.4.1 Operations

In the most general setting, the elements stored in a dictionary are pairs (x, y), where x is the *key* giving the ordering of the elements and y is the auxiliary information. For example, in a database storing student

© 2004 by Taylor & Francis Group, LLC

records, the key could be the student's last name, and the auxiliary information the student's transcript. It is convenient to augment the ordered universe of keys with two *special keys* ($+\infty$ and $-\infty$) and assume that each dictionary has, in addition to its *regular elements*, two *special elements*, with keys $+\infty$ and $-\infty$, respectively. For simplicity, we will also assume that no two elements of a dictionary have the same key. An insertion of an element with the same key as that of an existing element will be rejected by returning a null locator.

Using locators (see Section 4.1), we can define a more complete repertory of operations for a dictionary D:

SIZE(N): return the number of regular elements N of D.

FIND(x, c): if D contains an element with key x, assign to c a locator to such as an element; otherwise; set c equal to a null locator.

LOCATEPREV(x, c): assign to c a locator to the element of D with the largest key less than or equal to x; if x is smaller than all of the keys of the regular elements, then c is a locator to the special element with key $-\infty$; if $x = -\infty$, then c is a null locator.

LOCATENEXT(x, c): assign to c a locator to the element of D with the smallest key greater than or equal to x; if x is larger than all of the keys of the regular elements, then c is a locator to the special element with key $+\infty$; then, if $x = +\infty$, c is a null locator.

PREV(c', c''): assign to c'' a locator to the element of D with the largest key less than that of the element with locator c'; if the key of the element with locator c' is smaller than all of the keys of the regular elements, then this operation returns a locator to the special element with key $-\infty$.

NEXT(c', c''): assign to c'' a locator to the element of D with the smallest key larger than that of the element with locator c'; if the key of the element with locator c' is larger than all of the keys of the regular elements, then this operation returns a locator to the special element with key $+\infty$.

MIN(c): assign to c a locator to the regular element of D with minimum key; if D has no regular elements, then c is a null locator.

MAX(c): assign to c a locator to the regular element of D with maximum key; if D has no regular elements, then c is a null locator.

INSERT(e, c): insert element e into D, and return a locator c to e; if there is already an element with the same key as e, then this operation returns a null locator.

REMOVE(c, e): remove from D and return element e with locator c.

MODIFY(c, e): replace with e the element with locator c.

Some of these operations can be easily expressed by means of other operations of the repertory. For example, operation FIND is a simple variation of LOCATEPREV or LOCATENEXT.

4.4.2 Realization with a Sequence

We can realize a dictionary by reusing and extending the sequence abstract data type (see Section 4.2). Operations SIZE, INSERT, and REMOVE correspond to the homonymous sequence operations.

4.4.2.1 Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation FIND(x, c) can be performed by scanning the sequence with an iteration of NEXT operations, until we either find an element with key x, or we reach the end of the sequence. Table 4.7 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.2.2 Sorted Sequence

We can also use a sorted sequence to realize a dictionary. Operation INSERT now requires scanning the sequence to find the appropriate position to insert the new element. However, in a FIND operation, we can stop scanning the sequence as soon as we find an element with a key larger than the search key.

© 2004 by Taylor & Francis Group, LLC

TABLE 4.7 Performance of a Dictionary Realized by an Unsorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
NEXT	$O(N)$
PREV	$O(N)$
MIN	$O(N)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

TABLE 4.8 Performance of a Dictionary Realized by a Sorted Sequence, Implemented with a Doubly Linked List

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(N)$

Table 4.8 shows the time complexity of this realization by a sorted sequence, assuming that the sequence is implemented with a doubly linked list. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.2.3 Sorted Array

We can obtain a different performance trade-off by implementing the sorted sequence by means of an array, which allows constant-time access to any element of the sequence given its position. Indeed, with this realization we can speed up operation FIND(x, c) using the *binary search* strategy, as follows. If the dictionary is empty, we are done. Otherwise, let N be the current number of elements in the dictionary. We compare the search key k with the key x_m of the middle element of the sequence, that is, the element at position $\lfloor N/2 \rfloor$. If $x = x_m$, we have found the element. Else, we recursively search in the subsequence of the elements preceding the middle element if $x < x_m$, or following the middle element if $x > x_m$. At each recursive call, the number of elements of the subsequence being searched halves. Hence, the number of sequence elements accessed and the number of comparisons performed by binary search is $O(\log N)$. While searching takes $O(\log N)$ time, inserting or deleting elements now takes $O(N)$ time.

© 2004 by Taylor & Francis Group, LLC

TABLE 4.9 Performance of a Dictionary Realized by a Sorted Sequence, Implemented with an Array

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(N)$

Table 4.9 shows the performance of a dictionary realized with a sorted sequence, implemented with an array. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.3 Realization with a Search Tree

A *search tree* for elements of the type (x, y), where x is a key from a totally ordered universe, is a rooted ordered tree T such that:

- Each internal node of T has at least two children and stores a nonempty set of elements.
- A node μ of T with d children $\mu_1, \ldots, \mu_d$ stores $d - 1$ elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$, where $x_1 \leq \cdots \leq x_{d-1}$.
- For each element (x, y) stored at a node in the subtree of T rooted at μ_i, we have $x_{i-1} \leq x \leq x_i$, where $x_0 = -\infty$ and $x_d = +\infty$.

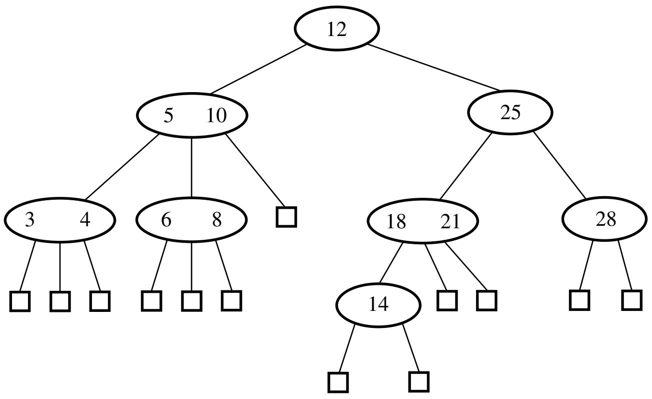
In a search tree, each internal node stores a nonempty collection of keys, whereas the leaves do not store any key and serve only as placeholders. An example search tree is shown in Figure 4.5a. A special type of search tree is a *binary search tree*, where each internal node stores one key and has two children.

We will recursively describe the realization of a dictionary D by means of a search tree T because we will use dictionaries to implement the nodes of T. Namely, an internal node μ of T with children $\mu_1, \ldots, \mu_d$ and elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$ is equipped with a dictionary $D(\mu)$ whose regular elements are the pairs $(x_i, (y_i, \mu_i))$, $i = 1, \ldots, d - 1$ and whose special element with key $+\infty$ is $(+\infty, (\cdot, \mu_d))$. A regular element (x, y) stored in D is associated with a regular element $(x, (y, \nu))$ stored in a dictionary $D(\mu)$, for some node μ of T. See the example in Figure 4.5b.

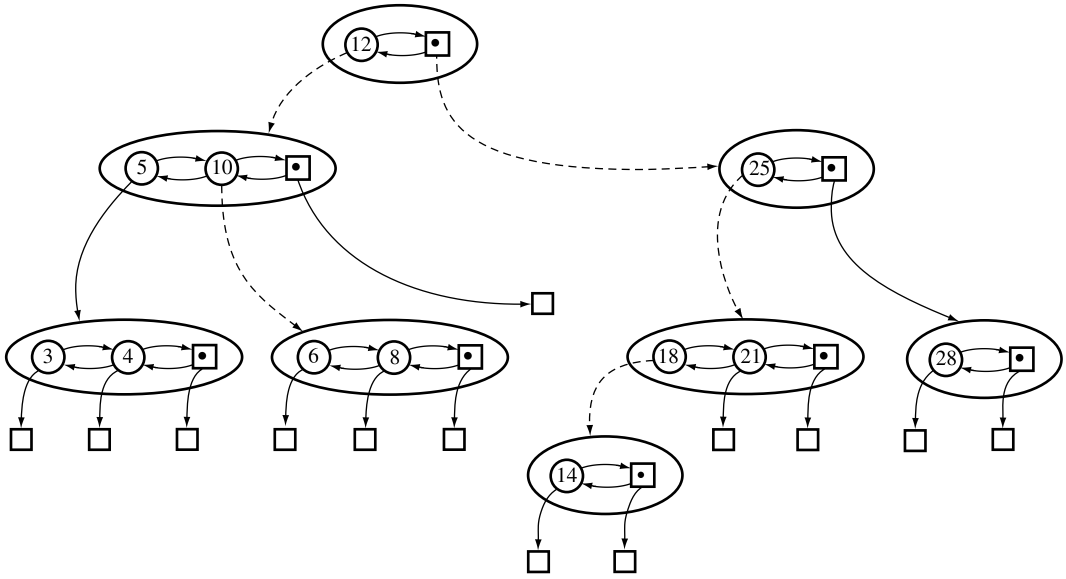
4.4.3.1 Operation FIND

Operation FIND(x, c) on dictionary D is performed by means of the following recursive method for a node μ of T, where μ is initially the root of T [see Figure 4.5b]. We execute LOCATENEXT(x, c') on dictionary $D(\mu)$ and let $(x', (y', \nu))$ be the element pointed by the returned locator c'. We have three cases:

1. Case $x = x'$: we have found x and return locator c to (x', y').
2. Case $x \neq x'$ and ν is a leaf: we have determined that x is not in D and return a null locator c.
3. Case $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.



(a)



(b)

FIGURE 4.5 Realization of a dictionary by means of a search tree: (a) a search tree T, (b) realization of the dictionaries at the nodes of T by means of sorted sequences. The search paths for elements 9 (unsuccessful search) and 14 (successful search) are shown with dashed lines.

4.4.3.2 Operation Insert

Operations LocatePrev, LocateNext, and Insert can be performed with small variations of the previously described method. For example, to perform operation Insert(e, c), where $e = (x, y)$, we modify the previous cases as follows (see Figure 4.6):

1. Case $x = x'$: an element with key x already exists, and we return a null locator.
2. Case $x \neq x'$ and ν is a leaf: we create a new leaf node λ, insert a new element $(x, (y, \lambda))$ into $D(\mu)$, and return a locator c to (x, y).
3. Case $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

Note that new elements are inserted at the bottom of the search tree.

© 2004 by Taylor & Francis Group, LLC

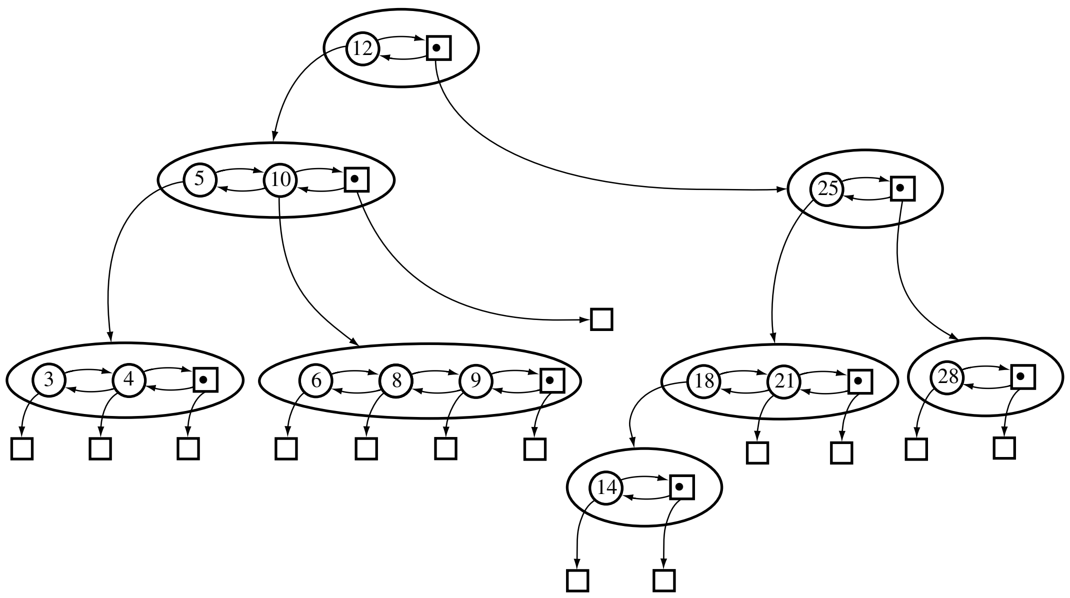


FIGURE 4.6 Insertion of element 9 into the search tree of Figure 4.5.

4.4.3.3 Operation Remove

Operation Remove(e, c) is more complex (see Figure 4.7). Let the associated element of $e = (x, y)$ in T be $(x, (y, \nu))$, stored in dictionary $D(\mu)$ of node μ:

- If node ν is a leaf, we simply delete element $(x, (y, \nu))$ from $D(\mu)$.
- Else (ν is an internal node), we find the successor element $(x', (y', \nu'))$ of $(x, (y, \nu))$ in $D(\mu)$ with a Next operation in $D(\mu)$. (1) If ν' is a leaf, we replace ν' with ν, that is, change element $(x', (y', \nu'))$ to $(x', (y', \nu))$, and delete element $(x, (y, \nu))$ from $D(\mu)$. (2) Else (ν' is an internal node), while the leftmost child ν'' of ν' is not a leaf, we set $\nu' = \nu''$. Let $(x'', (y'', \nu''))$ be the first element of $D(\nu')$ (node ν'' is a leaf). We replace $(x, (y, \nu))$ with $(x'', (y'', \nu))$ in $D(\mu)$ and delete $(x'', (y'', \nu''))$ from $D(\nu')$.

The listed actions may cause dictionary $D(\mu)$ or $D(\nu')$ to become empty. If this happens, say for $D(\mu)$ and μ is not the root of T, we need to remove node μ. Let $(+\infty, (\cdot, \kappa))$ be the special element of $D(\mu)$ with key $+\infty$, and let $(z, (w, \mu))$ be the element pointing to μ in the parent node π of μ. We delete node μ and replace $(z, (w, \mu))$ with $(z, (w, \kappa))$ in $D(\pi)$.

Note that if we start with an initially empty dictionary, a sequence of insertions and deletions performed with the described methods yields a search tree with a single node. In the next sections, we show how to avoid this behavior by imposing additional conditions on the structure of a search tree.

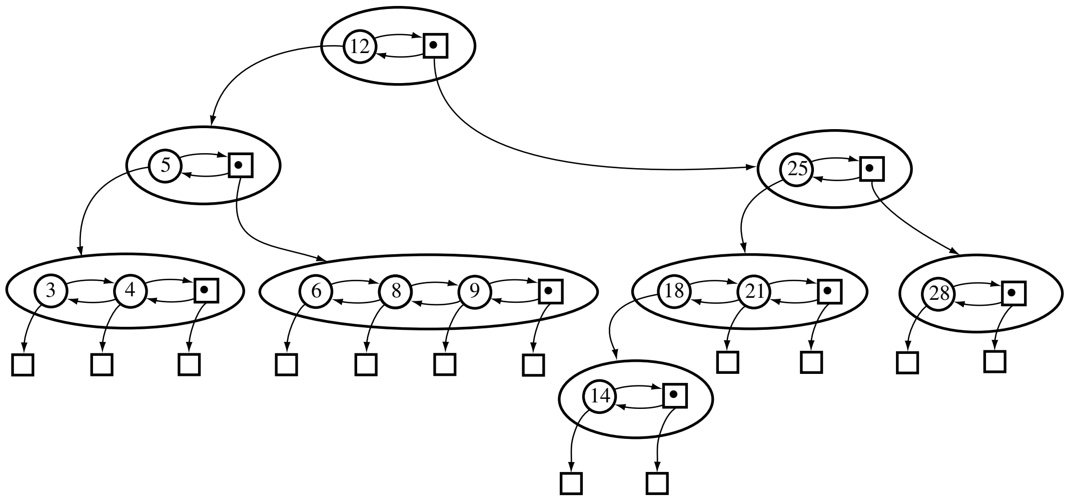
4.4.4 Realization with an (a, b)-Tree

An (a, b)-*tree*, where a and b are integer constants such that $2 \leq a \leq (b + 1)/2$, is a a search tree T with the following additional restrictions:

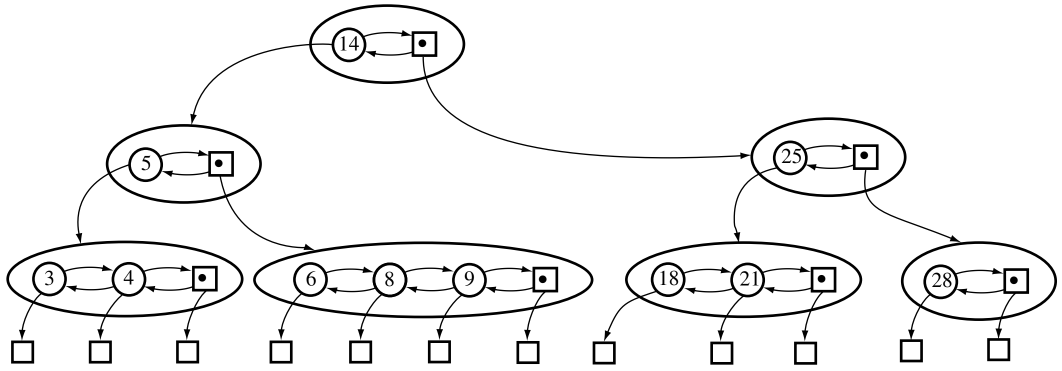
Level property. All of the levels of T are full, that is, all of the leaves are at the same depth.

Size property. Let μ be an internal node of T, and d be the number of children of μ; if μ is the root of T, then $d \geq 2$, else $a \leq d \leq b$.

© 2004 by Taylor & Francis Group, LLC



(a)



(b)

FIGURE 4.7 (a) Deletion of element 10 from the search tree of Figure 4.6. (b) Deletion of element 12 from the search tree of part a.

The height of an (a, b)-tree storing N elements is $O(\log_a N) = O(\log N)$. Indeed, in the worst case, the root has two children and all of the other internal nodes have a children.

The realization of a dictionary with an (a, b)-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE need to be modified in order to preserve the level and size properties. Also, we maintain the current size of the dictionary, and pointers to the minimum and maximum regular elements of the dictionary.

4.4.4.1 Insertion

The implementation of operation INSERT for search trees given earlier in this section adds a new element to the dictionary $D(\mu)$ of an existing node μ of T. Because the structure of the tree is not changed, the level property is satisfied. However, if $D(\mu)$ had the maximum allowed size $b - 1$ before insertion (recall that the size of $D(\mu)$ is one less than the number of children of μ), then the size property is violated at μ because $D(\mu)$ has now size b. To remedy this *overflow* situation, we perform the following *node split* (see Figure 4.8):

© 2004 by Taylor & Francis Group, LLC

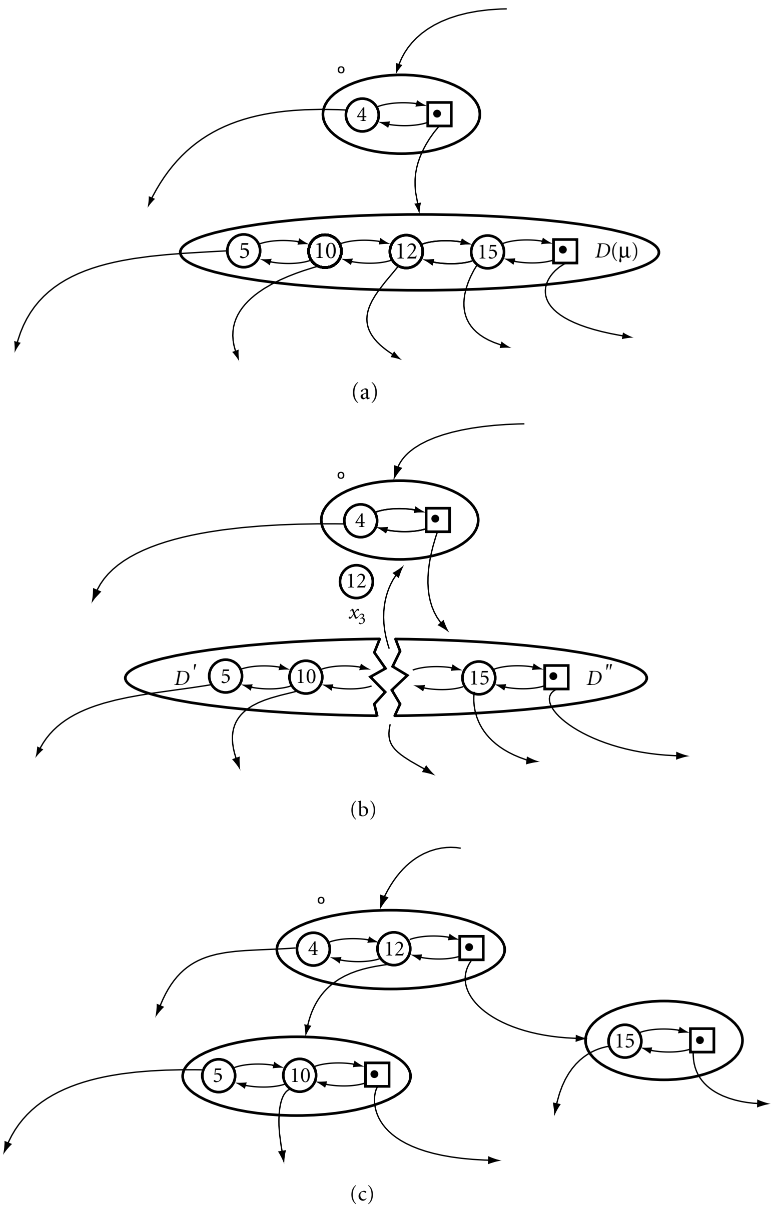


FIGURE 4.8 Example of node split in a 2–4 tree: (a) initial configuration with an overflow at node μ, (b) split of the node μ into μ' and μ'' and insertion of the median element into the parent node π, and (c) final configuration.

- Let the special element of $D(\mu)$ be $(+\infty, (\cdot, \mu_{b+1}))$. Find the median element of $D(\mu)$, that is, the element $e_i = (x_i, (y_i, \mu_i))$ such that $i = \lceil (b+1)/2 \rceil$).
- Split $D(\mu)$ into: (1) dictionary D' containing the $\lceil (b-1)/2 \rceil$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = 1 \cdots i-1$ and the special element $(+\infty, (\cdot, \mu_i))$; (2) element e; and (3) dictionary D'', containing the $\lfloor (b-1)/2 \rfloor$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = i+1 \cdots b$ and the special element $(+\infty, (\cdot, \mu_{b+1}))$.
- Create a new tree node κ, and set $D(\kappa) = D'$. Hence, node κ has children $\mu_1 \cdots \mu_i$.

© 2004 by Taylor & Francis Group, LLC

- Set $D(\mu) = D''$. Hence, node μ has children $\mu_{i+1} \cdots \mu_{b+1}$.
- If μ is the root of T, create a new node π with an empty dictionary $D(\pi)$. Else, let π be the parent of μ.
- Insert element $(x_i, (y_i, \kappa))$ into dictionary $D(\pi)$.

After a node split, the level property is still verified. Also, the size property is verified for all of the nodes of T, except possibly for node π. If π has $b + 1$ children, we repeat the node split for $\mu = \pi$. Each time we perform a node split, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the INSERT operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

4.4.4.2 Deletion

The implementation of operation REMOVE for search trees given earlier in this section removes an element from the dictionary $D(\mu)$ of an existing node μ of T. Because the structure of the tree is not changed, the level property is satisfied. However, if μ is not the root, and $D(\mu)$ had the minimum allowed size $a - 1$ before deletion (recall that the size of the dictionary is one less than the number of children of the node), then the size property is violated at μ because $D(\mu)$ has now size $a-2$. To remedy this *underflow* situation, we perform the following *node merge* (see Figure 4.9 and Figure 4.10):

- If μ has a right sibling, then let μ'' be the right sibling of μ and $\mu' = \mu$; else, let μ' be the left sibling of μ and $\mu'' = \mu$. Let $(+\infty, (\cdot, \nu))$ be the special element of $D(\mu')$.
- Let π be the parent of μ' and μ''. Remove from $D(\pi)$ the regular element $(x, (y, \mu'))$ associated with μ'.
- Create a new dictionary D containing the regular elements of $D(\mu')$ and $D(\mu'')$, regular element $(x, (y, \nu))$, and the special element of $D(\mu'')$.
- Set $D(\mu'') = D$, and destroy node μ'.
- If μ'' has more than b children, perform a node split at μ''.

After a node merge, the level property is still verified. Also, the size property is verified for all the nodes of T, except possibly for node π. If π is the root and has one child (and thus an empty dictionary), we remove node π. If π is not the root and has fewer than $a - 1$ children, we repeat the node merge for $\mu = \pi$. Each time we perform a node merge, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the REMOVE operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

4.4.4.3 Complexity

Let T be an (a, b)-tree storing N elements. The height of T is $O(\log_a N) = O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. We assume that the dictionaries at the nodes of T are realized with sequences. Hence, processing a node takes $O(b) = O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 4.10 shows the performance of a dictionary realized with an (a, b)-tree. In the table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.5 Realization with an AVL-Tree

An *AVL-tree* is a search tree T with the following additional restrictions:

Binary property. T is a binary tree, that is, every internal node has two children (left and right child), and stores one key.

Balance property. For every internal node μ, the heights of the subtrees rooted at the children of μ differ at most by one.

© 2004 by Taylor & Francis Group, LLC

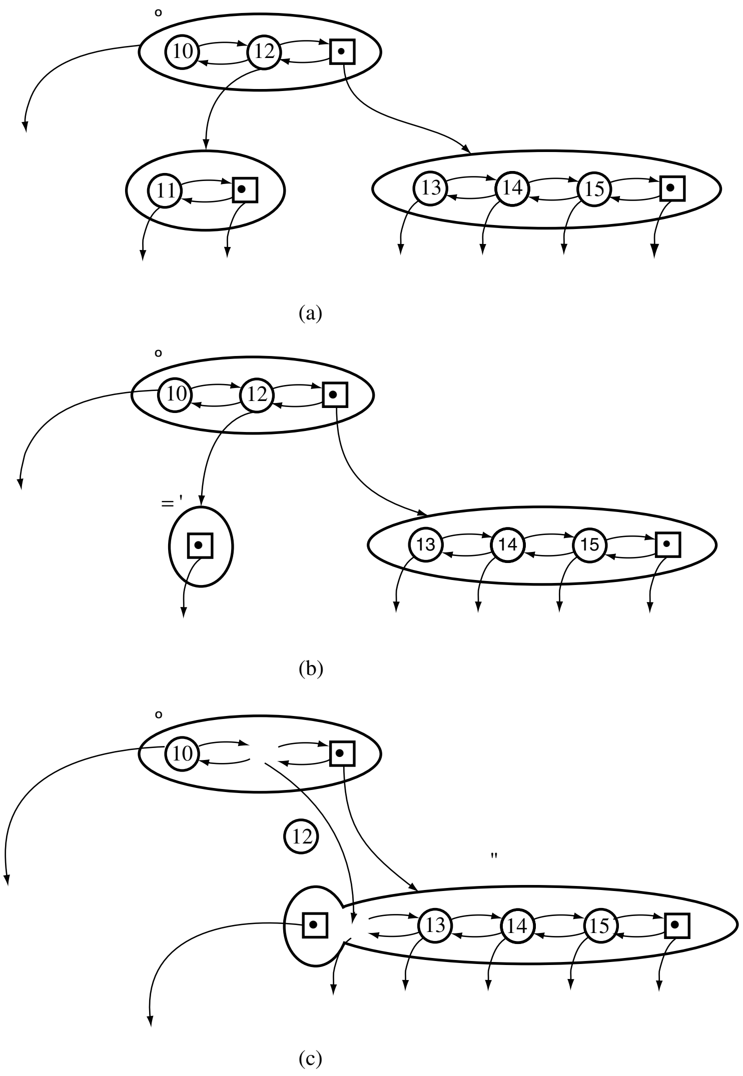


FIGURE 4.9 Example of node merge in a 2–4 tree: (a) initial configuration, (b) the removal of an element from dictionary $D(\mu)$ causes an underflow at node μ, and (c) merging node $\mu = \mu'$ into its sibling μ''.

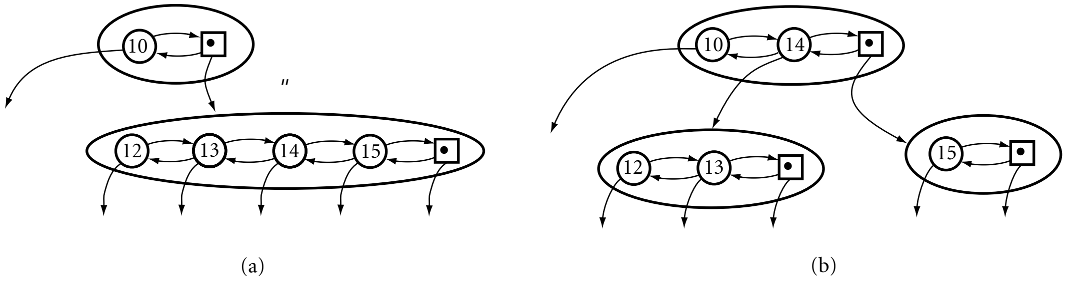


FIGURE 4.10 Example of subsequent node merge in a 2–4 tree: (a) overflow at node μ'' and (b) final configuration after splitting node μ''.

© 2004 by Taylor & Francis Group, LLC

TABLE 4.10 Performance of a Dictionary Realized by an (a, b)-Tree

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

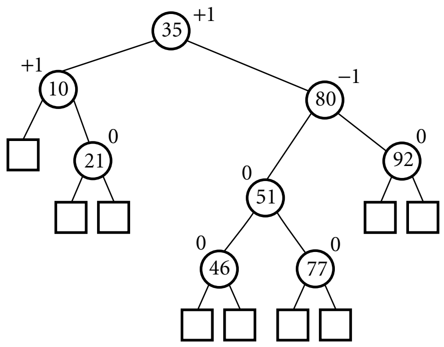


FIGURE 4.11 Example of AVL-tree storing nine elements. The keys are shown inside the nodes, and the balance factors (see subsequent section on rebalancing) are shown next to the nodes.

An example of AVL-tree is shown in Figure 4.11. The height of an AVL-tree storing N elements is $O(\log N)$. This can be shown as follows. Let N_h be the minimum number of elements stored in an AVL-tree of height h. We have $N_0 = 0$, $N_1 = 1$, and

$$N_h = 1 + N_{h-1} + N_{h-2}, \quad \text{for } h \geq 2$$

The preceding recurrence relation defines the well-known Fibonacci numbers. Hence, $N_h = \Omega(\phi^N)$, where $\phi = (1 + \sqrt{5})/2 = 1.6180 \cdots$ is the golden ratio.

The realization of a dictionary with an AVL-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE must be modified to preserve the binary and balance properties after an insertion or deletion.

4.4.5.1 Insertion

The implementation of INSERT for search trees given earlier in this section adds the new element to an existing node. This violates the binary property, and hence cannot be done in an AVL-tree. Hence, we modify the three cases of the INSERT algorithm for search trees as follows:

- Case $x = x'$: an element with key x already exists, and we return a null locator c.
- Case $x \neq x'$ and ν is a leaf: we replace ν with a new internal node κ with two leaf children, store element (x, y) in κ, and return a locator c to (x, y).
- Case $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

© 2004 by Taylor & Francis Group, LLC

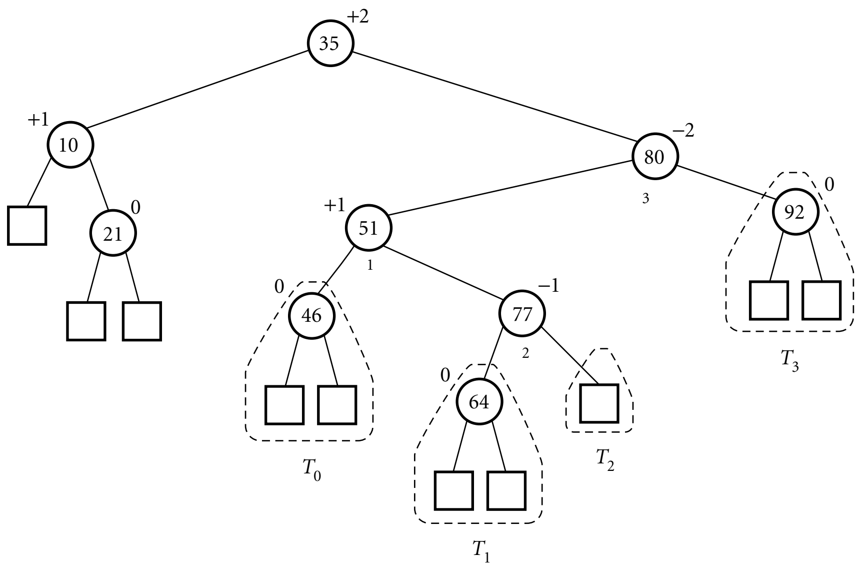


FIGURE 4.12 Insertion of an element with key 64 into the AVL-tree of Figure 4.11. Note that two nodes (with balance factors +2 and −2) have become unbalanced. The dashed lines identify the subtrees that participate in the rebalancing, as illustrated in Figure 4.14.

We have preserved the binary property. However, we may have violated the balance property because the heights of some subtrees of T have increased by one. We say that a node is balanced if the difference between the heights of its subtrees is −1, 0, or 1, and is unbalanced otherwise. The unbalanced nodes form a (possibly empty) subpath of the path from the new internal node κ to the root of T. See the example of Figure 4.12.

4.4.5.2 Rebalancing

To restore the balance property, we *rebalance* the lowest node μ that is unbalanced, as follows:

- Let μ' be the child of μ whose subtree has maximum height, and μ'' be the child of μ' whose subtree has maximum height.
- Let (μ_1, μ_2, μ_3) be the left-to-right ordering of nodes $\{\mu, \mu', \mu''\}$, and (T_0, T_1, T_2, T_3) be the left-to-right ordering of the four subtrees of $\{\mu, \mu', \mu''\}$ not rooted at a node in $\{\mu, \mu', \mu''\}$.
- Replace the subtree rooted at μ with a new subtree rooted at μ_2, where μ_1 is the left child of μ_2 and has subtrees T_0 and T_1, and μ_3 is the right child of μ_2 and has subtrees T_2 and T_3.

Two examples of rebalancing are schematically shown in Figure 4.14. Other symmetric configurations are possible. In Figure 4.13 we show the rebalancing for the tree of Figure 4.12.

Note that the rebalancing causes all the nodes in the subtree of μ_2 to become balanced. Also, the subtree rooted at μ_2 now has the same height as the subtree rooted at node μ before insertion. This causes all of the previously unbalanced nodes to become balanced. To keep track of the nodes that become unbalanced, we can store at each node a *balance factor*, which is the difference of the heights of the left and right subtrees. A node becomes unbalanced when its balance factor becomes +2 or −2. It is easy to modify the algorithm for operation INSERT such that it maintains the balance factors of the nodes.

4.4.5.3 Deletion

The implementation of REMOVE for search trees given earlier in this section preserves the binary property, but may cause the balance property to be violated. After deleting a node, there can be only one unbalanced node, on the path from the deleted node to the root of T.

© 2004 by Taylor & Francis Group, LLC

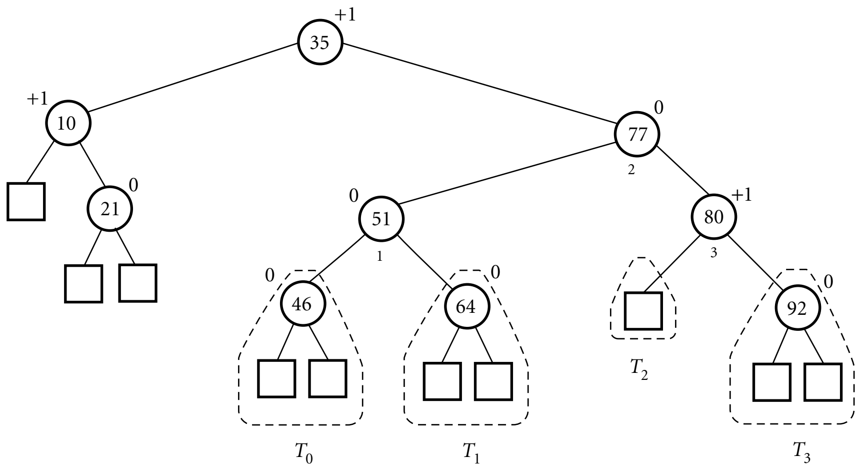


FIGURE 4.13 AVL-tree obtained by rebalancing the lowest unbalanced node in the tree of Figure 4.11. Note that all of the nodes are now balanced. The dashed lines identify the subtrees that participate in the rebalancing, as illustrated in Figure 4.14.

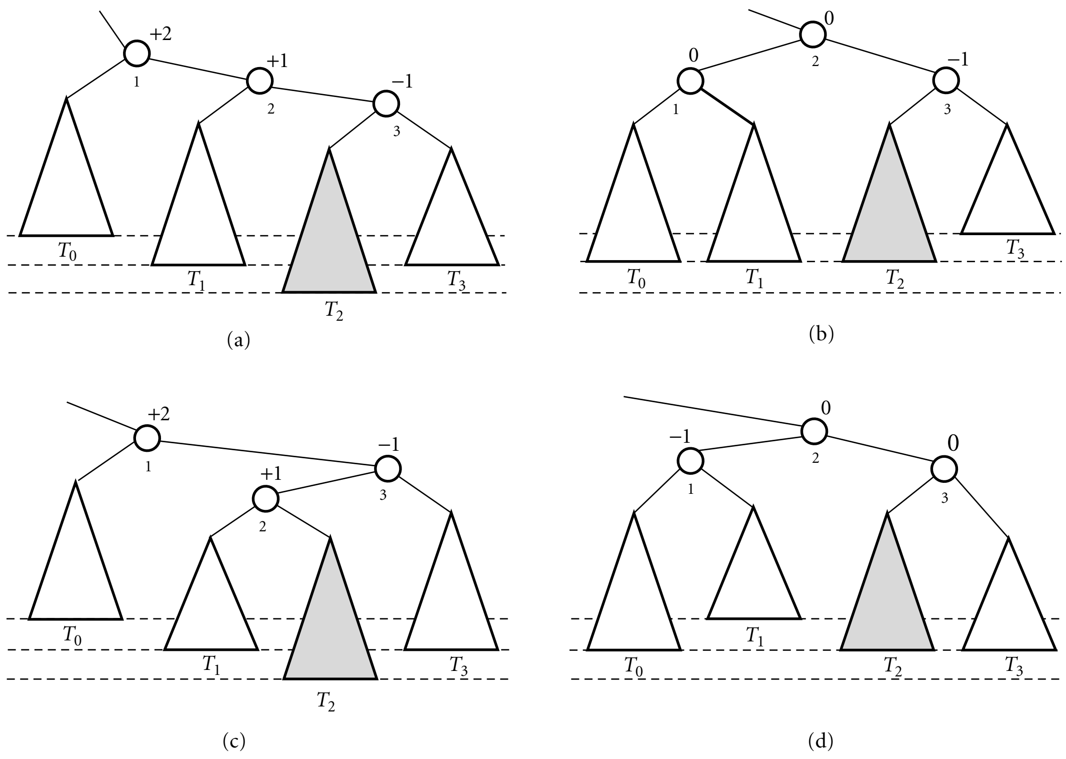


FIGURE 4.14 Schematic illustration of rebalancing a node in the INSERT algorithm for AVL-trees. The shaded subtree is the one where the new element was inserted. (a) and (b) Rebalancing by means of a single rotation. (c) and (d) Rebalancing by means of a double rotation.

© 2004 by Taylor & Francis Group, LLC

TABLE 4.11 Performance of a Dictionary Realized by an AVL-Tree

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

To restore the balance property, we *rebalance* the unbalanced node using the previous algorithm, with minor modifications. If the subtrees of μ' have the same height, the height of the subtree rooted at μ_2 is the same as the height of the subtree rooted at μ before rebalancing, and we are done. If, instead, the subtrees of μ' do not have the same height, then the height of the subtree rooted at μ_2 is one less than the height of the subtree rooted at μ before rebalancing. This may cause an ancestor of μ_2 to become unbalanced, and we repeat the above computation. Balance factors are used to keep track of the nodes that become unbalanced, and can be easily maintained by the REMOVE algorithm.

4.4.5.4 Complexity

Let T be an AVL-tree storing N elements. The height of T is $O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. Rebalancing a node takes $O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 4.11 shows the performance of a dictionary realized with an AVL-tree. In this table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.4.6 Realization with a Hash Table

The previous realizations of a dictionary make no assumptions on the structure of the keys and use comparisons between keys to guide the execution of the various operations.

4.4.6.1 Bucket Array

If the keys of a dictionary D are integers in the range $[1, M]$, we can implement D with a *bucket array* B. An element (x, y) of D is represented by setting $B[x] = y$. If an integer x is not in D, the location $B[x]$ stores a null value. In this implementation, we allocate a bucket for every possible element of D.

Table 4.12 shows the performance of a dictionary realized with a bucket array. In this table the keys in the dictionary are integers in the range $[1, M]$. The space complexity is $O(M)$.

The bucket array method can be extended to keys that are easily mapped to integers. For example, three-letter airport codes can be mapped to the integers in the range $[1, 26^3]$.

4.4.6.2 Hashing

The bucket array method works well when the range of keys is small. However, it is inefficient when the range of keys is large. To overcome this problem, we can use a *hash function* h that maps the keys of the original dictionary D into integers in the range $[1, M]$, where M is a parameter of the hash function. Now, we can apply the bucket array method using the *hashed value* $h(x)$ of the keys. In general, a *collision* may

© 2004 by Taylor & Francis Group, LLC

TABLE 4.12 Performance of a Dictionary Realized by Bucket Array

Operation	Time
SIZE	$O(1)$
FIND	$O(1)$
LOCATEPREV	$O(M)$
LOCATENEXT	$O(M)$
NEXT	$O(M)$
PREV	$O(M)$
MIN	$O(M)$
MAX	$O(M)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

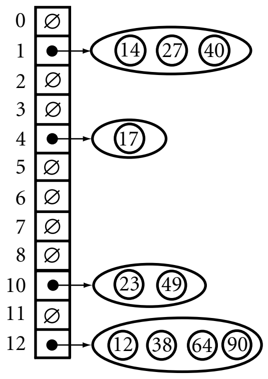


FIGURE 4.15 Example of a hash table of size 13 storing 10 elements. The hash function is $h(x) = x \bmod 13$.

happen, where two distinct keys x_1 and x_2 have the same hashed value, that is, $x_1 \neq x_2$ and $h(x_1) = h(x_2)$. Hence, each bucket must be able to accommodate a collection of elements.

A hash table of size M for a function $h(x)$ is a bucket array B of size M (primary structure) whose entries are dictionaries (secondary structures), such that element (x, y) is stored in the dictionary $B[h(x)]$. For simplicity of programming, the dictionaries used as secondary structures are typically realized with sequences. An example of a hash table is shown in Figure 4.15.

If all of the elements in the dictionary D collide, they are all stored in the same dictionary of the bucket array, and the performance of the hash table is the same as that of the kind of dictionary used for the secondary structures. At the other end of the spectrum, if no two elements of the dictionary D collide, they are stored in distinct one-element dictionaries of the bucket array, and the performance of the hash table is the same as that of a bucket array.

A typical hash function for integer keys is $h(x) = x \bmod M$ (here, the range is $[0, M - 1]$). The size M of the hash table is usually chosen as a prime number. An example of a hash table is shown in Figure 4.15. It is interesting to analyze the performance of a hash table from a probabilistic viewpoint. If we assume that the hashed values of the keys are uniformly distributed in the range $[0, M - 1]$, then each bucket holds on average N/M keys, where N is the size of the dictionary. Hence, when $N = O(M)$, the average size of the secondary data structures is $O(1)$.

Table 4.13 shows the performance of a dictionary realized with a hash table. Both the worst-case and average time complexity in the preceding probabilistic model are indicated. In this table we denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity

© 2004 by Taylor & Francis Group, LLC

TABLE 4.13 Performance of a Dictionary Realized by a Hash Table of Size M

	Time	
Operation	Worst Case	Average
SIZE	$O(1)$	$O(1)$
FIND	$O(N)$	$O(N/M)$
LOCATEPREV	$O(N+M)$	$O(N+M)$
LOCATENEXT	$O(N+M)$	$O(N+M)$
NEXT	$O(N+M)$	$O(N+M)$
PREV	$O(N+M)$	$O(N+M)$
MIN	$O(N+M)$	$O(N+M)$
MAX	$O(N+M)$	$O(N+M)$
INSERT	$O(1)$	$O(1)$
REMOVE	$O(1)$	$O(1)$
MODIFY	$O(1)$	$O(1)$

is $O(N+M)$. The average time complexity refers to a probabilistic model where the hashed values of the keys are uniformly distributed in the range $[1, M]$.

Acknowledgments

Work supported in part by the National Science Foundation under grant DUE–0231202. Bryan Cantrill contributed to this work while at Brown University.

Defining Terms

(a, b)-Tree:	Search tree with additional properties (each node has between a and b children, and all the levels are full).
Abstract data type:	Mathematically specified data type equipped with operations that can be performed on the objects.
AVL-tree:	Binary search tree such that the subtrees of each node have heights that differ by at most one.
Binary search tree:	Search tree such that each internal node has two children.
Bucket array:	Implementation of a dictionary by means of an array indexed by the keys of the dictionary elements.
Container:	Abstract data type storing a collection of objects (elements).
Dictionary:	Container storing elements from a sorted universe supporting searches, insertions, and deletions.
Hash table:	Implementation of a dictionary by means of a bucket array storing secondary dictionaries.
Heap:	Binary tree with additional properties storing the elements of a priority queue.
Position:	Object representing the place of an element stored in a container.
Locator:	Mechanism for tracking an element stored in a container.
Priority queue:	Container storing elements from a sorted universe that supports finding the maximum element, insertions, and deletions.
Search tree:	Rooted ordered tree with additional properties storing the elements of a dictionary.
Sequence:	Container storing objects in a linear order, supporting insertions (in a given position) and deletions.

References

Aggarwal, A. and Vitter, J.S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127.

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. 1983. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA.

© 2004 by Taylor & Francis Group, LLC

Chazelle, B. and Guibas, L.J. 1986. Fractional cascading. I. A data structuring technique. *Algorithmica*, 1:133–162.

Chiang, Y.-J. and Tamassia, R. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434.

Cohen, R.F. and Tamassia, R. 1995. Dynamic expression trees. *Algorithmica*, 13:245–265.

Comer, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. 2001. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

Di Battista, G. and Tamassia, R. 1996. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318.

Di Battista, G., Eades, P., Tamassia, R., and Tollis, I.G. 1999. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ.

Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E. 1989. Making data structures persistent. *J. Comput. Syst. Sci.* 38:86–124.

Edelsbrunner, H. 1987. *Algorithms in Combinatorial Geometry,* Vol. 10, *EATCS Monographs on Theoretical Computer Science.* Springer–Verlag, Heidelberg, Germany.

Eppstein, D., Galil, Z., Italiano, G.F., and Nissenzweig, A. 1997. Sparsification: a technique for speeding up dynamic graph algorithms. *J. ACM*, 44:669–696.

Even, S. 1979. *Graph Algorithms*. Computer Science Press, Potomac, MD.

Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. 1990. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA.

Frederickson, G.N. 1997. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24:37–65.

Galil, Z. and Italiano, G.F. 1991. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344.

Gonnet, G.H. and Baeza-Yates, R. 1991. *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA.

Goodrich, M.T. and Tamassia, R. 2001. *Data Structures and Algorithms in Java*. Wiley, New York.

Hoffmann, K., Mehlhorn, K., Rosenstiehl, P., and Tarjan, R.E. 1986. Sorting Jordan sequences in linear time using level-linked search trees. *Inf. Control*, 68:170–184.

Horowitz, E., Sahni, S., and Metha, D. 1995. *Fundamentals of Data Structures in C++*. Computer Science Press, Potomac, MD.

Knuth, D.E. 1968. *Fundamental Algorithms.* Vol. I. In *The Art of Computer Programming*. Addison-Wesley, Reading, MA.

Knuth, D.E. 1973. *Sorting and Searching,* Vol. 3. In *The Art of Computer Programming.* Addison-Wesley, Reading, MA.

Mehlhorn, K. 1984. *Data Structures and Algorithms.* Vol. 1–3. Springer–Verlag.

Mehlhorn, K. and Näher, S. 1999. *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press.

Mehlhorn, K. and Tsakalidis, A. 1990. Data structures. In *Algorithms and Complexity*. J. van Leeuwen, Ed., Vol. A, *Handbook of Theoretical Computer Science.* Elsevier, Amsterdam.

Munro, J.I. and Suwanda, H. 1980. Implicit Data Structures for Fast Search and Update. *J. Comput. Syst. Sci.*, 21:236–250.

Nievergelt, J. and Hinrichs, K.H. 1993. *Algorithms and Data Structures: With Applications to Graphics and Geometry.* Prentice Hall, Englewood Cliffs, NJ.

O'Rourke, J. 1994. *Computational Geometry in C.* Cambridge University Press,

Overmars, M.H. 1983. *The Design of Dynamic Data Structures,* Vol. 156, *Lecture Notes in Computer Science.* Springer-Verlag.

Preparata, F.P. and Shamos, M.I. 1985. *Computational Geometry: An Introduction.* Springer-Verlag, New York.

Pugh, W. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 35:668–676.

© 2004 by Taylor & Francis Group, LLC

Sedgewick, R. 1992. *Algorithms in C++*. Addison-Wesley, Reading, MA.

Sleator, D.D. and Tarjan, R.E. 1993. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381.

Tamassia, R., Goodrich, M.T., Vismara, L., Handy, M., Shubina, G., Cohen R., Hudson, B., Baker, R.S., Gelfand, N., and Brandes, U. 2001. JDSL: the data structures library in Java. *Dr. Dobb's Journal*, 323:21–31.

Tarjan, R.E. 1983. *Data Structures and Network Algorithms, Vol. 44, CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial Applied Mathematics.

Vitter, J.S. and Flajolet, P. 1990. Average-case analysis of algorithms and data structures. In *Algorithms and Complexity,* J. van Leeuwen, Ed., Vol. A, *Handbook of Theoretical Computer Science,* pp. 431–524. Elsevier, Amsterdam.

Wood, D. 1993. *Data Structures, Algorithms, and Performance.* Addison-Wesley, Reading, MA.

Further Information

Many textbooks and monographs have been written on data structures, for example, Aho et al. [1983], Cormen et al. [2001], Gonnet and Baeza-Yates [1990], Goodrich and Tamassia [2001], Horowitz et al. [1995], Knuth [1968, 1973], Mehlhorn [1984], Nievergelt and Hinrichs [1993], Overmars [1983], Preparata and Shamos [1995], Sedgewick [1992], Tarjan [1983], and Wood [1993].

Papers surveying the state-of-the art in data structures include Chiang and Tamassia [1992], Galil and Italiano [1991], Mehlhorn and Tsakalidis [1990], and Vitter and Flajolet [1990].

JDSL is a library of fundamental data structures in Java [Tamassia et al. 2000]. LEDA is a library of advanced data structures in C++ [Mehlhorn and Näher 1999].

© 2004 by Taylor & Francis Group, LLC

5 Complexity Theory

Eric W. Allender
Rutgers University

Michael C. Loui
University of Illinois at Urbana-Champaign

Kenneth W. Regan
State University of New York at Buffalo

5.1 Introduction

Computational complexity is the study of the difficulty of solving computational problems, in terms of the required computational resources, such as time and space (memory). Whereas the analysis of algorithms focuses on the time or space of an *individual* algorithm for a *specific* problem (such as sorting), complexity theory focuses on the **complexity class** of problems solvable in the same amount of time or space. Most common computational problems fall into a small number of complexity classes. Two important complexity classes are P, the set of problems that can be solved in polynomial time, and NP, the set of problems whose solutions can be verified in polynomial time.

By quantifying the resources required to solve a problem, complexity theory has profoundly affected our thinking about computation. Computability theory establishes the existence of undecidable problems, which cannot be solved in principle regardless of the amount of time invested. However, computability theory fails to find meaningful distinctions among decidable problems. In contrast, complexity theory establishes the existence of decidable problems that, although solvable in principle, cannot be solved in

© 2004 by Taylor & Francis Group, LLC

practice because the time and space required would be larger than the age and size of the known universe [Stockmeyer and Chandra, 1979]. Thus, complexity theory characterizes the computationally feasible problems.

The quest for the boundaries of the set of feasible problems has led to the most important unsolved question in all of computer science: is P different from NP? Hundreds of fundamental problems, including many ubiquitous optimization problems of operations research, are **NP-complete**; they are the hardest problems in NP. If someone could find a polynomial-time algorithm for any one NP-complete problem, then there would be polynomial-time algorithms for all of them. Despite the concerted efforts of many scientists over several decades, no polynomial-time algorithm has been found for any NP-complete problem. Although we do not yet know whether P is different from NP, showing that a problem is NP-complete provides strong evidence that the problem is computationally infeasible and justifies the use of heuristics for solving the problem.

In this chapter, we define P, NP, and related complexity classes. We illustrate the use of **diagonalization** and **padding** techniques to prove relationships between classes. Next, we define NP-completeness, and we show how to prove that a problem is NP-complete. Finally, we define complexity classes for probabilistic and interactive computations.

Throughout this chapter, all numeric functions take integer arguments and produce integer values. All logarithms are taken to base 2. In particular, $\log n$ means $\lceil \log_2 n \rceil$.

5.2 Models of Computation

To develop a theory of the difficulty of computational problems, we need to specify precisely what a problem is, what an algorithm is, and what a measure of difficulty is. For simplicity, complexity theorists have chosen to represent problems as languages, to model algorithms by off-line multitape **Turing machines**, and to measure computational difficulty by the time and space required by a Turing machine. To justify these choices, some theorems of complexity theory show how to translate statements about, say, the time complexity of language recognition by Turing machines into statements about computational problems on more realistic models of computation. These theorems imply that the principles of complexity theory are not artifacts of Turing machines, but intrinsic properties of computation.

This section defines different kinds of Turing machines. The deterministic Turing machine models actual computers. The nondeterministic Turing machine is not a realistic model, but it helps classify the complexity of important computational problems. The alternating Turing machine models a form of parallel computation, and it helps elucidate the relationship between time and space.

5.2.1 Computational Problems and Languages

Computer scientists have invented many elegant formalisms for representing data and control structures. Fundamentally, all representations are patterns of symbols. Therefore, we represent an instance of a computational problem as a sequence of symbols.

Let Σ be a finite set, called the *alphabet*. A *word* over Σ is a finite sequence of symbols from Σ. Sometimes a word is called a *string*. Let Σ^* denote the set of all words over Σ. For example, if $\Sigma = \{\mathbf{0}, \mathbf{1}\}$, then

$$\Sigma^* = \{\lambda, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{01}, \mathbf{10}, \mathbf{11}, \mathbf{000}, \ldots\}$$

is the set of all binary words, including the empty word λ. The *length* of a word w, denoted by $|w|$, is the number of symbols in w. A *language* over Σ is a subset of Σ^*.

A *decision problem* is a computational problem whose answer is simply `yes` or `no`. For example, is the input graph connected, or is the input a sorted list of integers? A decision problem can be expressed as a membership problem for a language A: for an input x, does x belong to A? For a language A that represents connected graphs, the input word x might represent an input graph G, and $x \in A$ if and only if G is connected.

© 2004 by Taylor & Francis Group, LLC

For every decision problem, the representation should allow for easy parsing, to determine whether a word represents a legitimate instance of the problem. Furthermore, the representation should be concise. In particular, it would be unfair to encode the answer to the problem into the representation of an instance of the problem; for example, for the problem of deciding whether an input graph is connected, the representation should not have an extra bit that tells whether the graph is connected. A set of integers $S = \{x_1, \ldots, x_m\}$ is represented by listing the binary representation of each x_i, with the representations of consecutive integers in S separated by a nonbinary symbol. A graph is naturally represented by giving either its adjacency matrix or a set of adjacency lists, where the list for each vertex v specifies the vertices adjacent to v.

Whereas the solution to a decision problem is `yes` or `no`, the solution to an optimization problem is more complicated; for example, determine the shortest path from vertex u to vertex v in an input graph G. Nevertheless, for every optimization (minimization) problem, with objective function g, there is a corresponding decision problem that asks whether there exists a feasible solution z such that $g(z) \leq k$, where k is a given target value. Clearly, if there is an algorithm that solves an optimization problem, then that algorithm can be used to solve the corresponding decision problem. Conversely, if an algorithm solves the decision problem, then with a binary search on the range of values of g, we can determine the optimal value. Moreover, using a decision problem as a subroutine often enables us to construct an optimal solution; for example, if we are trying to find a shortest path, we can use a decision problem that determines if a shortest path starting from a given vertex uses a given edge. Therefore, there is little loss of generality in considering only decision problems, represented as language membership problems.

5.2.2 Turing Machines

This subsection and the next three give precise, formal definitions of Turing machines and their variants. These subsections are intended for reference. For the rest of this chapter, the reader need not understand these definitions in detail, but may generally substitute "program" or "computer" for each reference to "Turing machine."

A k-worktape **Turing machine** M consists of the following:

- A finite set of states Q, with special states q_0 (initial state), q_A (accept state), and q_R (reject state).
- A finite alphabet Σ, and a special blank symbol $\square \notin \Sigma$.
- The $k + 1$ linear tapes, each divided into cells. Tape 0 is the *input tape*, and tapes $1, \ldots, k$ are the *worktapes*. Each tape is infinite to the left and to the right. Each cell holds a single symbol from $\Sigma \cup \{\square\}$. By convention, the input tape is read only. Each tape has an access head, and at every instant, each access head scans one cell (see Figure 5.1).

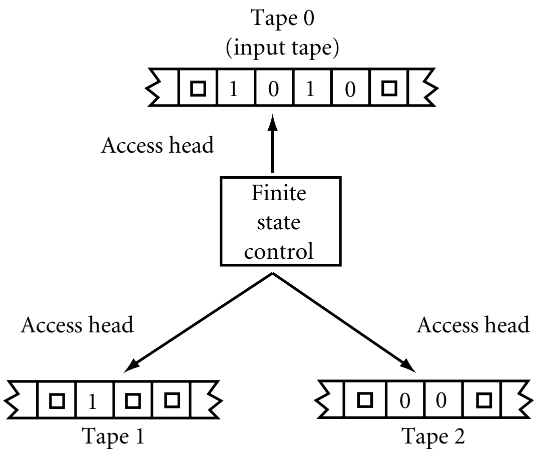


FIGURE 5.1 A two-tape Turing machine.

© 2004 by Taylor & Francis Group, LLC