

DATA STRUCTURES

Evangel P. Quiwa

*Department of Computer Science
College of Engineering
University of the Philippines*

The preparation of this book was supported through a textbook writing grant by the University of the Philippines System through the Office of the Vice-President for Academic Affairs.

Data Structures

Philippine Copyright ©2007 by Evangel P. Quiwa and Office of the Vice-President for Academic Affairs, University of the Philippines System

All rights reserved. No patent liability is assumed with respect to the use of the information contained herein. Every precaution has been taken in the preparation of this book; however, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

ISBN 971-897806-1

For feedback on this book please contact: **evangel.quiwa@up.edu.ph**

Send all inquiries to:

Electronics Hobbyists Publishing House

P.O. Box 4052

Manila

Printed in the Philippines

Cover design: Leslie Q. Quiwa

Cover photo: Tree Silhouette 3 by Christophe Libert

To Pilar

Preface

This book is based on ‘session notes’ which the author developed and used in teaching a course on Data Structures to sophomore Computer Science students over a period of several years. The book retains the original organization of the ‘notes’ in which a ‘session’ (the equivalent of one or more class meetings) begins with a list of OBJECTIVES (indicating what the student is expected to be able to do after the session), followed by a set of READINGS (which refer the student to specific reference materials for the session notes) and ends with an extended DISCUSSION of the material covered by the session (the main body) and a set of EXERCISES.

The first three sessions comprise an introduction to the study of data structures and algorithms proper, which is covered in the remaining twelve sessions. Session 1 begins with a question the answer to which impacts the rest of the sessions: ‘Given a problem P , how do we solve P on a computer?’ It highlights the fact that at the heart of the problem-solving process on a computer lie two intimately related tasks, viz., the ‘structuring of data’ and the ‘synthesis of algorithms’. Data structures and algorithms (or the two fused together as a single entity called an ‘object’) are the building blocks of computer programs.

Session 2 is a review of familiar mathematical entities, tools and concepts, and an introduction to some less familiar topics which will be expounded in their proper setting in later sessions.

Session 3 tackles two important issues about algorithms, viz., ‘How do we communicate an algorithm?’ and ‘How do we analyze an algorithm?’ For reasons articulated in this session, we choose pseudocode (called EASY) as our medium of communication and we implement algorithms as EASY procedures. To illustrate how EASY translates easily into the syntax of actual programming languages, we transcribe a number of EASY procedures for stacks, queues and binary trees in subsequent sessions into running Pascal and C programs.

The second half of Session 3 expounds on what it means to ‘analyze an algorithm.’ Using the analysis of insertion sort as the vehicle for exposition, we lead the student to a firm understanding of very important ‘asymptotic notations’ which will be used in the remainder of the book to denote the space and time complexity of running algorithms. The quest for suitable data structures and efficient algorithms to solve a given problem on a computer is a thread that runs throughout these Notes.

By now it should have become abundantly clear that we cannot study data structures apart from algorithms. Thus in Sessions 4 through 15 we will consider in turn each of the

following data structures — stacks, queues, deques, binary trees, priority queues, trees and forests, graphs, linear lists, generalized lists, sequential tables, binary search trees and hash tables — and associated algorithms. Applications of these data structures to the solution of both classical CS problems as well as real-world problems are discussed at great length in this book; often the solutions can be described as nothing less than ‘elegant’ as well. After all, true programming is nothing less than an art.

Acknowledgments

I thank all the authors whose books served as my references in preparing the ‘session notes’ for my Data Structures classes, and from which this book is derived.

I thank the University of the Philippines System for providing me, through the Office of the Vice President for Academic Affairs, a textbook writing grant.

I thank my colleagues in the Department of Computer Science who used the original ‘session notes’ in their own classes and encouraged me to consolidate them into a book.

I thank Dan Custodio who initiated me into the use of \LaTeX . I prepared this entire document in \LaTeX .

Writing a book of this size can be an overwhelming task. I thank my wife, Pilar, for her quiet support and encouragement without which I could not have completed this task.

Evangel P. Quiwa

February 8, 2007

Contents

Preface	v
1 Basic Concepts	1
1.1 How do we solve a problem on a computer?	2
1.2 What is a data structure?	3
1.3 What is an algorithm?	5
1.4 Implementing ADT's	8
1.4.1 The contiguous or sequential design	8
1.4.2 The linked design	9
1.5 Formulating a solution to the ESP	12
Summary	17
Exercises	17
Bibliographic Notes	19
2 Mathematical Preliminaries	21
2.1 Mathematical notations and elementary functions	22
2.1.1 Floor, ceiling and mod functions	22
2.1.2 Polynomials	23
2.1.3 Exponentials	23
2.1.4 Logarithms	24
2.1.5 Factorials	25
2.1.6 Fibonacci numbers	25
2.2 Sets	26
2.3 Relations	28
2.4 Permutations and Combinations	32
2.4.1 Rule of sum and rule of product	32

viii CONTENTS

2.4.2	Permutations	33
2.4.3	Combinations	34
2.5	Summations	35
2.5.1	Arithmetic series	35
2.5.2	Geometric series	36
2.5.3	Harmonic series	37
2.5.4	Miscellaneous sums	37
2.6	Recurrences	38
2.7	Methods of proof	42
2.7.1	Proof by mathematical induction	42
2.7.2	Proof by contradiction (<i>reductio ad absurdum</i>)	45
2.7.3	Proof by counterexample	45
	Summary	46
	Exercises	46
	Bibliographic Notes	48
3	Algorithms	49
3.1	Communicating an algorithm	50
3.1.1	The EASY assignment statement	51
3.1.2	The EASY unconditional transfer statements	51
3.1.3	The EASY conditional transfer statements	52
3.1.4	The EASY iteration statements	52
3.1.5	The EASY input/output statements	53
3.1.6	The EASY declaration statements	53
3.1.7	The EASY control statements	54
3.1.8	EASY program structure	54
3.1.9	Sample EASY procedures	55
3.2	Analyzing an algorithm	59
3.2.1	Analysis of insertion sort	59
3.2.2	Asymptotic notations	62
3.2.3	The O -notation	63
3.2.4	The Ω -notation	66
3.2.5	The Θ -notation	67
3.2.6	The preponderance of the O -notation	68
3.2.7	Comparative growth of functions	69
3.2.8	Complexity classes	69
	Summary	71
	Exercises	71
	Bibliographic Notes	73

4	Stacks	75
4.1	Sequential implementation of a stack	77
4.1.1	Implementing the auxiliary operations	78
4.1.2	Implementing the insert operation	78
4.1.3	Implementing the delete operation	79
4.1.4	A Pascal implementation of the array representation of a stack . . .	79
4.1.5	A C implementation of the array representation of a stack	81
4.2	Linked list implementation of a stack	83
4.2.1	Implementing the auxiliary stack operations	84
4.2.2	Implementing the insert operation	85
4.2.3	Implementing the delete operation	85
4.2.4	A Pascal implementation of the linked list representation of a stack	86
4.2.5	A C implementation of the linked list representation of a stack . . .	87
4.3	Applications of stacks	89
4.3.1	A simple pattern recognition problem	89
4.3.2	Conversion of arithmetic expressions from infix to postfix form . . .	90
4.4	Sequential implementation of multiple stacks	98
4.4.1	The coexistence of two stacks in a single array	99
4.4.2	The coexistence of three or more stacks in a single array	99
4.4.3	Reallocating memory at stack overflow	101
	Summary	106
	Exercises	106
	Bibliographic Notes	108
5	Queues and Deques	111
5.1	Sequential implementation of a straight queue	112
5.1.1	Implementing the insert operation	113
5.1.2	Implementing the delete operation	114
5.2	Sequential implementation of a circular queue	115
5.2.1	Implementing the insert and delete operations for a circular queue .	116
5.2.2	A C implementation of the array representation of a circular queue	118
5.3	Linked list implementation of a queue	120
5.3.1	Implementing the insert operation	120
5.3.2	Implementing the delete operation	121
5.3.3	A C implementation of the linked list representation of a queue . .	121
5.4	Application of queues: topological sorting	124

5.5	The deque as an abstract data type	132
5.6	Sequential implementation of a straight deque	133
5.6.1	Implementing the insert operation	134
5.6.2	Implementing the delete operation	135
5.7	Sequential implementation of a circular deque	135
5.7.1	Implementing the insert and delete operations for a circular deque .	137
5.8	Linked list implementation of a deque	138
5.8.1	Implementing the insert and delete operations	138
	Summary	139
	Exercises	140
	Bibliographic Notes	141

6 Binary Trees 143

6.1	Definitions and related concepts	144
6.2	Binary tree traversals	149
6.3	Binary tree representation in computer memory	153
6.4	Implementing the traversal algorithms	154
6.4.1	Recursive implementation of preorder and postorder traversal . . .	154
6.4.2	Iterative implementation of preorder and postorder traversal . . .	155
6.4.3	Two simple applications of the traversal algorithms	161
6.5	Threaded binary trees	162
6.5.1	Finding successors and predecessors in a threaded binary tree . . .	164
6.5.2	Traversing inorder threaded binary trees	167
6.5.3	Growing threaded binary trees	168
6.5.4	Similarity and equivalence of binary trees	171
6.6	Traversal by link inversion	172
6.7	Siklóssy traversal	175
	Summary	177
	Exercises	178
	Bibliographic Notes	181

7	Applications of binary trees	183
7.1	Calculating conveyance losses in an irrigation network	184
7.2	Heaps and the heapsort algorithm	192
7.2.1	Sift-up: converting a complete binary tree into a heap	193
7.2.2	The heapsort algorithm of Floyd and Williams	198
7.3	Priority queues	202
7.3.1	Unsorted array implementation of a priority queue	204
7.3.2	Sorted array implementation of a priority queue	205
7.3.3	Heap implementation of a priority queue	206
	Summary	207
	Exercises	208
	Bibliographic Notes	210
8	Trees and forests	211
8.1	Definitions and related concepts	212
8.2	Natural correspondence	214
8.3	Forest traversal	216
8.4	Linked representation for trees and forests	218
8.5	Sequential representation for trees and forests	218
8.5.1	Representing trees and forests using links and tags	219
8.5.2	Arithmetic tree representations	224
8.6	Trees and the equivalence problem	228
8.6.1	The equivalence problem	228
8.6.2	Degeneracy and the weighting rule for <i>union</i>	233
8.6.3	Path compression: the collapsing rule for <i>find</i>	235
8.6.4	Analysis of the union and find operations	236
8.6.5	Final solution to the equivalence problem	237
	Summary	239
	Exercises	239
	Bibliographic Notes	241

9	Graphs	243
9.1	Some pertinent definitions and related concepts	244
9.2	Representation of graphs	249
9.2.1	Adjacency matrix representation of a graph	249
9.2.2	Adjacency lists representation of a graph	250
9.3	Graph traversals	252
9.3.1	Depth first search	253
9.3.2	Breadth first search	265
9.4	Some classical applications of DFS and BFS	271
9.4.1	Identifying directed acyclic graphs	271
9.4.2	Topological sorting	272
9.4.3	Finding the strongly connected components of a digraph	274
9.4.4	Finding articulation points and biconnected components of a con- nected undirected graph	277
9.4.5	Determining whether an undirected graph is acyclic	282
	Summary	283
	Exercises	284
	Bibliographic Notes	286
10	Applications of graphs	287
10.1	Minimum-cost spanning trees for undirected graphs	288
10.1.1	Prim's algorithm	289
10.1.2	Kruskal's algorithm	301
10.2	Shortest path problems for directed graphs	312
10.2.1	Dijkstra's algorithm for the SSSP problem	312
10.2.2	Floyd's algorithm for the APSP problem	325
10.2.3	Warshall's algorithm and transitive closure	331
	Summary	333
	Exercises	333
	Bibliographic Notes	335
11	Linear lists	337
11.1	Linear Lists	338
11.2	Sequential representation of linear lists	339
11.3	Linked representation of linear lists	339
11.3.1	Straight singly-linked linear list	340

11.3.2	Straight singly-linked linear list with a list head	342
11.3.3	Circular singly-linked linear list	343
11.3.4	Circular singly-linked linear list with a list head	345
11.3.5	Doubly-linked linear lists	345
11.4	Linear lists and polynomial arithmetic	347
11.5	Linear lists and multiple-precision integer arithmetic	353
11.5.1	Integer addition and subtraction	353
11.5.2	Integer multiplication	356
11.5.3	Representing long integers in computer memory	359
11.5.4	EASY procedures for multiple-precision integer arithmetic	360
11.6	Linear lists and dynamic storage management	367
11.6.1	The memory pool	367
11.6.2	Sequential-fit methods: reservation	368
11.6.3	Sequential-fit methods: liberation	373
11.6.4	Buddy-system methods: reservation and liberation	378
11.7	Linear lists and UPCAT processing	388
11.7.1	Determining degree program qualifiers within a campus	389
11.7.2	Implementation issues	390
11.7.3	Sample EASY procedures	394
	Summary	395
	Exercises	395
	Bibliographic Notes	397
12	Generalized lists	399
12.1	Definitions and related concepts	400
12.2	Representing generalized lists in computer memory	402
12.3	Implementing some basic operations on generalized lists	407
12.4	Traversing pure lists	411
12.5	Automatic storage reclamation	413
12.5.1	Algorithms for marking and gathering	416
	Summary	420
	Exercises	420
	Bibliographic Notes	422

13 Sequential tables	423
13.1 The table ADT	424
13.2 Direct-address table	425
13.3 Static sequential tables	427
13.3.1 Linear search in a static sequential table	428
13.3.2 Binary search in an ordered static sequential table	431
13.3.3 Multiplicative binary search	435
13.3.4 Fibonaccian search in an ordered sequential table	441
Summary	444
Exercises	444
Bibliographic Notes	445
14 Binary search trees	447
14.1 Binary search trees	448
14.1.1 Implementing some basic operations for dynamic tables	450
14.1.2 Analysis of BST search	455
14.2 Height-balanced binary search trees	461
14.3 Rotations on AVL trees	464
14.4 Optimum binary search trees	477
Summary	485
Exercises	486
Bibliographic Notes	487
15 Hash tables	489
15.1 Converting keys to numbers	491
15.2 Choosing a hash function	493
15.2.1 The division method	493
15.2.2 The multiplicative method	496
15.3 Collision resolution by chaining	499
15.3.1 EASY procedures for chained hash tables	501
15.3.2 Analysis of hashing with collision resolution by chaining	503
15.4 Collision resolution by open addressing	504
15.4.1 Linear probing	506
15.4.2 Quadratic probing	507
15.4.3 Double hashing	508
15.4.4 EASY procedures for open-address hash tables	510

15.4.5	Ordered open-address hash tables	515
15.4.6	Analysis of hashing with collision resolution by open addressing	518
	Summary	519
	Exercises	520
	Bibliographic Notes	521
	Bibliography	523
	Index	525

NOTES

SESSION 1

Basic Concepts

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Explain the basic concept of an abstract data type and its role in the problem solving process on a computer.
2. Explain the basic concepts of a data structure and an algorithm and the close relationship between the structuring of data and the synthesis of algorithms.
3. Explain the notion of finiteness of an algorithm.
4. Describe the two basic addressing methods used to access the elements of a data structure.

READINGS KNUTH1[1997], pp. 1–9; AHO[1983], pp. 1–16; STANDISH[1994], pp. 3–11.

DISCUSSION

Given a problem P , how do we solve P on a computer? The tacit assumption in this question is that P is *solvable*; in fact, it may not be. A classic example of an *unsolvable* problem is the *halting problem*:

HP: Given a program Q and an input x , determine whether Q will halt on x .

The halting problem is unsolvable because there does not exist an algorithm to solve the problem. You will understand better the intriguing notion of unsolvability in a subsequent course on Automata Theory and Computability where you will encounter more interesting problems of the likes of HP. For now, we will set aside unsolvable problems and focus our attention on problems of the solvable kind.

There is vast array of problems, ranging from the mundane to the arcane, from the practical to the theoretical, from the tractable to the intractable, which can be solved on a computer. There are mundane problems whose solution on a computer can be described as nothing less than truly elegant. There are simple, practical problems which turn out to be inherently difficult and for which no efficient solutions are currently known. There are problems which take only a few seconds to solve on a computer even for large inputs;

2 SESSION 1. Basic Concepts

there are also problems which would take centuries to solve even for small inputs. The common thread which underlies this vast array of problems is this: there exist algorithms which a computer can be programmed to execute to solve such problems. Given some such problem, how do we go about formulating and implementing a solution on a computer?

1.1 How do we solve a problem on a computer?

To be more specific, consider the following familiar problem.

ESP: Given n courses, prepare a schedule of examinations for the n courses subject to the following conditions:

1. Examinations for courses with common students must be assigned different time slots (in other words, ‘*walang* conflicts’).
2. Examinations must be scheduled using the *minimum* number of time slots.

Let us call this the *exam-scheduling problem*, or ESP for short. In the absence of condition 2, a trivial solution to this problem is to schedule the n examinations using n different time slots. It is condition 2 which makes the scheduling problem interesting, and as we shall subsequently see, quite difficult to solve. How, then, do we develop and implement on the computer a solution to the exam-scheduling problem?

To put the task at hand in perspective, let us think in terms of domains. At the *problem domain*, we have on the one hand ‘raw data’ consisting of a list of the n courses and a list of students enrolled in each of these courses; and, on the other hand ‘processed data’ consisting of the generated schedule of examinations (see Figure 1.1). The latter (which comprise the output) is extracted from the former (which comprise the input) through the application of an algorithm which implements the problem requirements (no conflicts, minimal number of time slots). To accomplish this task, we use a machine called a computer.

At the *machine domain*, we have on the one hand a storage medium consisting of serially arranged bits which are organized into addressable units called bytes which in turn are grouped into machine words, etc.; and, on the other hand processing units which allow us to perform such basic operations as addition, subtraction, comparison, and so on.

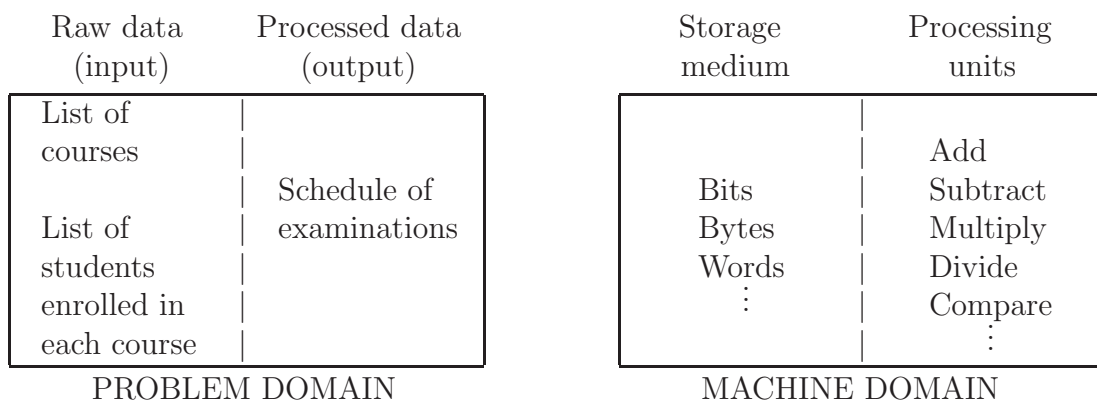


Figure 1.1 A problem to solve and a machine on which to solve the problem

To carry out the problem-solving process on a computer, we need to provide a link between the problem domain and the machine domain. We will call this link the *solution domain*. At the solution domain, we need to address two very intimately related tasks as we formulate and implement our solution to the given problem on a computer, namely:

1. the structuring of higher level data representations (lists, trees, graphs, tables, etc) from the basic information structures available at the level of the machine (bits, bytes, words, etc.) that will faithfully represent the information (raw and processed data) at the problem domain
2. the synthesis of algorithms (for searching, sorting, traversal, etc.) from the basic operations provided at the level of the machine (addition, subtraction, comparison, etc.) to manipulate these higher-level data representations so that we obtain the desired results

These two tasks, the *structuring of data* and the *synthesis of algorithms*, lie at the heart of the problem-solving process on a computer. Data structures and algorithms are the building blocks of computer programs; they constitute the main subject of study in these Notes.

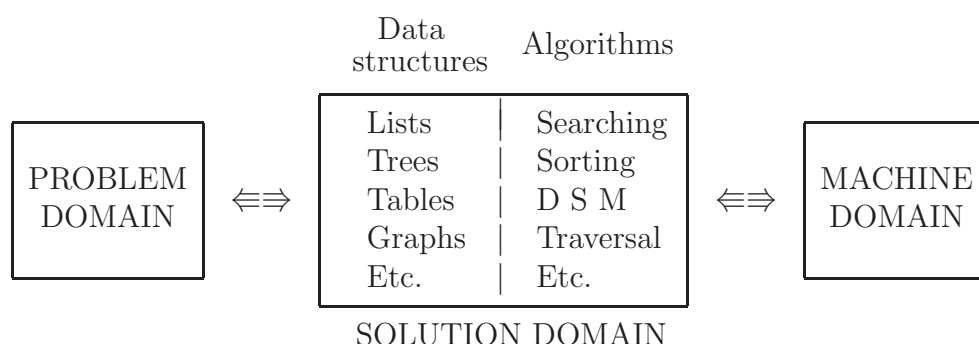


Figure 1.2 What this book is all about (and more)

1.2 What is a data structure?

The terms below should be familiar from a course in programming. Unfortunately, different authors define these terms differently, with varying levels of formalism. For our purposes, the following definitions should suffice.

1. **data type** — refers to the kind of data that variables may hold or take on in a programming language, and for which operations are automatically provided. For example:

PASCAL: integer, real, boolean and character

C: character, integer, floating point, double floating point and valueless

FORTRAN: integer, real, complex, logical and character

LISP: S-expression

Typically, a programming language also provides aggregate or compound data types, such as arrays, records, structures, and so forth. More complex, user-defined data types may be constructed out of these language data types. Operations on these different data types are provided through built-in operators, functions and other features of the language.

2. **abstract data type** or **ADT** — is a set of data elements with a set of operations defined on the data elements. A familiar and extremely useful ADT is the stack ADT:

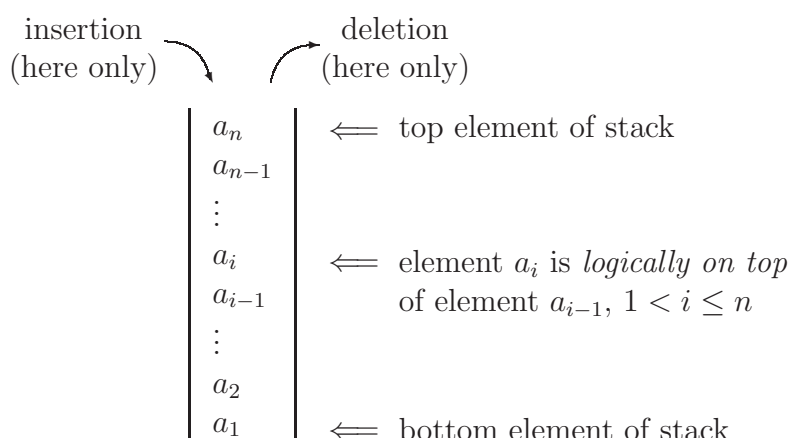


Figure 1.3 The stack ADT

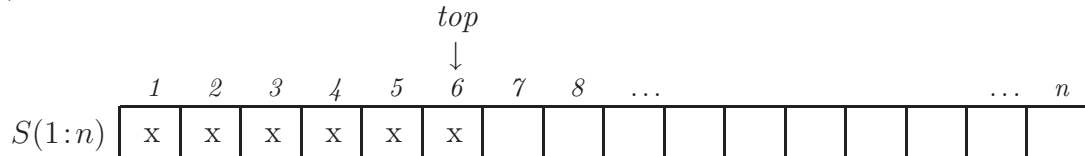
As an abstract data type, we can think of a stack as a linearly ordered set of data elements on which is imposed the discipline of last-in, first-out. The stack elements may be any entity, such as a token, a return address, intermediate values in a computation, etc. The basic operations on a stack are *insertion of an element* and *deletion of an element*, and these are restricted to the top of the stack only. Auxiliary operations include *initializing a stack* and *testing if a stack is empty*.

An ADT allows us to think about how a collection of data is organized, how the data elements are inter-related, and what operations are performed on them, without being concerned about details of implementation. Changing our point of view, we can look at the operations defined on the elements of an ADT as the responsibilities of the ADT to its users. Thus the stack ADT provides its user the ability to initialize a stack, test if a stack is empty, insert an element at the top of a stack and delete the top element of a stack. Whichever point of view we take, the key idea is that we know *what* an ADT does (*specification*); *how* it does what it does (*implementation*) is altogether another matter.

3. **data structure** — is the *implementation* or *realization*, using a specific programming language, of an ADT in terms of language data types or other data structures such that operations on the data structure are expressible in terms of directly executable procedures. Figure 1.4 depicts two implementations of the stack ADT: as a sequentially allocated one-dimensional array (sequential design) and as a singly-linked linear list (linked design). The elements marked ‘x’ comprise the stack. We will explore these alternative implementations of the stack ADT, along with the

corresponding procedures for insertion, deletion and other auxiliary operations in Session 4.

(a) array implementation



(b) linked-list implementation

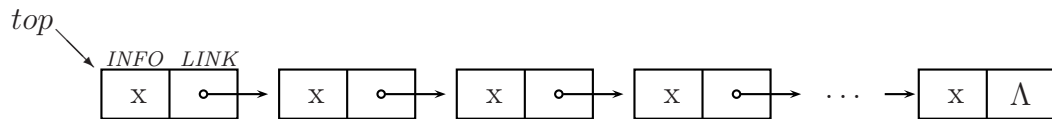


Figure 1.4 Two implementations of the stack ADT

1.3 What is an algorithm?

The concept of an algorithm is central to the study of computer science. An algorithm can be defined with varying degrees of rigor and formalism. For instance, if we make the claim that no algorithm exists to solve such and such a problem, we must define very rigorously what we mean by the term ‘algorithm’. For our purposes in these Notes, it suffices to adopt the following definition from KNUTH1[1997], pp. 4–6: an algorithm *is a finite set of instructions which, if followed, will accomplish a particular task*. It has five important properties.

1. *Finiteness* — an algorithm must terminate after a finite number of steps (we will keep an abiding interest in this very important property of an algorithm)
2. *Definiteness* — each step of an algorithm must be precisely defined (an instruction such as ‘Multiply by a number greater than zero’ won’t do)
3. *Input* — an algorithm takes some value, or set of values, as input
4. *Output* — an algorithm produces some value, or set of values, as output
5. *Effectiveness* — all of the operations to be performed must be sufficiently basic that they can in principle be done exactly and in finite time by a person using only pencil and paper

All the algorithms that we will study in this book satisfy these properties. In most cases we will find it quite easy to carry out the algorithm ‘by hand’ using pencil, paper and eraser. Often the real challenge is implementing the algorithm on a computer in a most efficient and elegant way.

6 SESSION 1. Basic Concepts

As an example, consider Euclid's algorithm for finding the greatest common divisor of two positive integers m and n :

Algorithm E: Euclid's algorithm

1. Divide m by n and let r be the remainder.
2. If r is zero, stop; $GCD = n$.
3. Set $m \leftarrow n$, $n \leftarrow r$ and go to 1.

Is Euclid's algorithm in fact an algorithm? Are the five properties satisfied? It is easy to see that the last four properties are satisfied, but it is not readily apparent that the first one is. Will the procedure terminate for *any* positive integers m and n ? Let us consider a specific example and see if we can deduce some general truth about the procedure. To this end, let $m = 608$ and $n = 133$.

Loop 1:

- Step 1. $\frac{608}{133} = 4$ remainder 76
Step 2. remainder is not zero, continue
Step 3. $m \leftarrow 133$, $n \leftarrow 76$

Loop 2:

- Step 1. $\frac{133}{76} = 1$ remainder 57
Step 2. remainder is not zero, continue
Step 3. $m \leftarrow 76$, $n \leftarrow 57$

Loop 3:

- Step 1. $\frac{76}{57} = 1$ remainder 19
Step 2. remainder is not zero, continue
Step 3. $m \leftarrow 57$, $n \leftarrow 19$

Loop 4:

- Step 1. $\frac{57}{19} = 3$ remainder 0.
Step 2. remainder is zero, stop; $GCD = 19$

We see from the results that the divisors used (133,76,57,19) constitute a decreasing sequence of positive integers. Such a sequence will always terminate, in the worst case, at 1 (when?). Euclid's algorithm *is* an algorithm.

An algorithm must not only be finite; it must be *very* finite. To illustrate what we mean by 'very finite', consider the familiar problem of solving a system of n linear equations in n unknowns. Such a linear system may be written in matrix form,

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

where

\mathbf{A} is the matrix of coefficients (of size n by n)

\mathbf{b} is the vector of constants (of size n)

\mathbf{x} is the vector of unknowns (of size n)

The solution vector $\bar{\mathbf{x}} = [\bar{x}_1, \bar{x}_2, \bar{x}_3, \dots, \bar{x}_n]^t$ can be obtained by using Cramer's rule,

$$\bar{x}_i = \frac{\det \mathbf{A}_i}{\det \mathbf{A}} \quad i = 1, 2, \dots, n$$

where

\bar{x}_i is the i th element of the solution vector $\bar{\mathbf{x}}$

$\det \mathbf{A}_i$ is the determinant of \mathbf{A}_i , where \mathbf{A}_i is \mathbf{A} with the i th column replaced by \mathbf{b}

$\det \mathbf{A}$ is the determinant of \mathbf{A}

It is clear that solving a system of n equations in n unknowns using Cramer's rule requires evaluating $n + 1$ determinants. Each such determinant can be found by using expansion by minors, which requires the evaluation of $n!$ elementary products, each elementary product requiring $n - 1$ multiplications. Thus the total number of multiplications performed is $(n - 1)n!(n + 1) = (n - 1)(n + 1)!$.

Now assume that we have a computer which can perform 1 billion multiplications per second. The table below shows the amount of time it will take this hypothetical computer to solve a linear system of n equations in n unknowns using Cramer's rule and expansion by minors to evaluate determinants.

n	Time required	Remarks
11	4.8 seconds	
13	17.4 minutes	
14	4.7 hours	
15	3.4 days	
17	3.2 years	
19	13.9 centuries	'Though this be madness, yet there is method in't.' - Shakespeare
21	712.3 millenia	'Though there be method in't, yet this is madness.' - Sarah Keepse

Imagine what it will take to solve 100 equations in 100 unknowns, or 1000 equations in 1000 unknowns. It is not uncommon to be solving linear systems of this size or larger; in a subsequent course on Numerical Methods you will study much more efficient algorithms for solving systems of linear equations. We see from this example that although a procedure is finite, it is practically useless unless it is finite enough. In the next session, we will elaborate on this idea further when we discuss a way of indicating the *time complexity* of an algorithm.

1.4 Implementing ADT's

Earlier we defined an ADT to be a collection of data elements with a collection of operations defined on the data elements. The word ‘abstract’ in the term ‘abstract data type’ signifies that an ADT is *implementation-independent*. When we solve a problem on a computer, it is useful to think first in terms of ADT's as we formulate a solution to the given problem. By initially leaving out details of implementation, we are able to better cope with the complexities of the problem-solving process. Thinking at a higher level of abstraction allows us to gain a deeper insight into the problem we are solving and leads us not only to a ‘working’ solution (*puwede na*) but to a solution that is ‘elegant’ as well.

The culmination of the problem-solving process on a computer is, of course, a running program. The ADT's in terms of which we have expressed our solution will eventually have to be implemented using the constructs of a specific programming language as data structures with their associated procedures. There are two basic ways by which we can realize an ADT as a data structure, namely, the **contiguous** or **sequential** design (or representation) and the **linked** design.

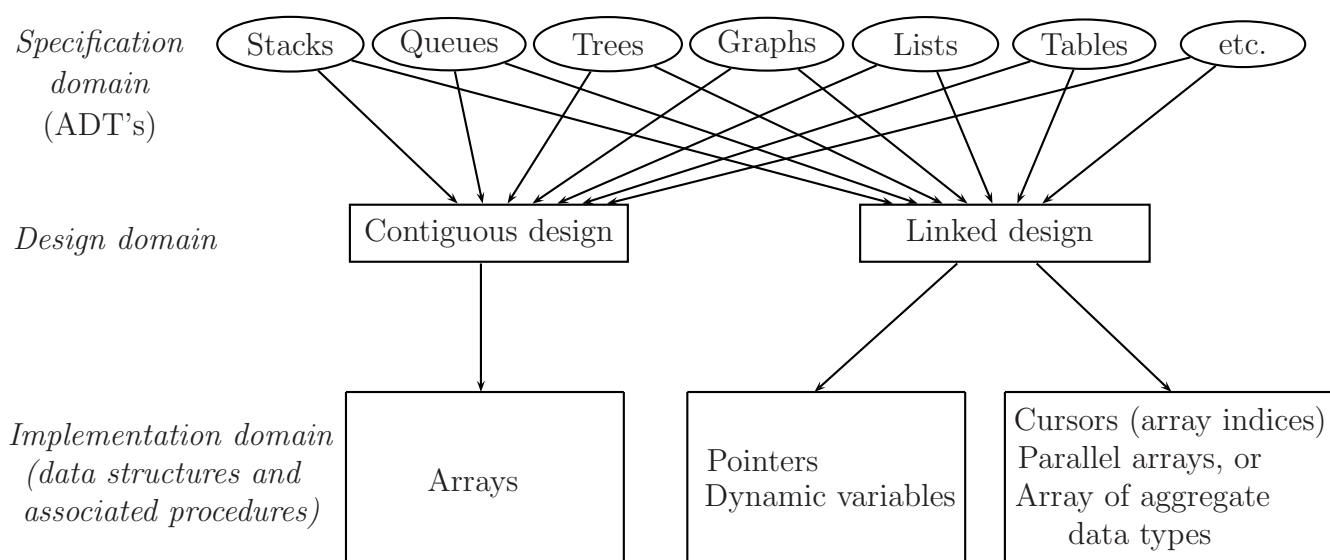


Figure 1.5 From ADT's to data structures

1.4.1 The contiguous or sequential design

We use the contiguous design when we deal with a structure that is essentially *static* and whose size is known beforehand. The implementing construct for the contiguous design is the *array* which is available as a compound data type in many programming languages. The data elements are stored in contiguous locations in computer memory and are accessed by using the the usual array-indexing features provided by the language.

Consider the Pascal declaration

var A: array[1..50] of integer ;

1	
2	
3	
⋮	⋮
⋮	⋮
<i>i</i>	
⋮	⋮
⋮	⋮
⋮	⋮
49	
50	

which reserves 50 successive cells, each of size c , say. The address of the first of these cells is called the base address, which we denote as β . If now we want to store the value 611953 in the i th cell of the array **A**, where $1 \leq i \leq 50$, we write

A(i) := 611953

This value is actually stored at address $\beta + c * (i - 1)$, but we need not refer to this address at all. We simply specify the index i ; the runtime system performs the address computation automatically. While this addressing method is conceptually easy to use, there are many structures and fundamental operations on these structures that require more flexibility than this method provides.

1.4.2 The linked design

We use the linked design when we need to manipulate *dynamic* structures which change in size and shape at run time. Central to this representation is the concept of a *node*, a *field* and a *pointer*.

The implementing construct for the linked design is the *node*. A *node* is an addressable unit of memory which consists of named segments called *fields*. A node can be realized in various ways, for instance, as a Pascal *record*, as a C *structure*, and so on. In these Notes we will represent nodes as indicated in Figure 1.6.

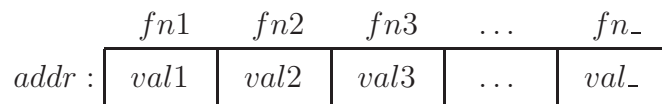


Figure 1.6 A node and its fields

where

$addr$ is the address of the node

$fn1, fn2, \dots$ are the names of the fields comprising the node

$val1, val2, \dots$ are the values stored in the fields of the node

10 SESSION 1. Basic Concepts

A node has no name, so we refer to it through its address. To access a specific field of a node, we use the notation

field name (address of node)

which signifies that a field name, qualified by the address of a node, is a *variable* whose value is the content of the field.

As an example, consider the node shown below.

INFO LINK
 $\alpha :$

ADA	1000
-----	------

Node α has two fields, named *INFO* and *LINK*. The content of the *INFO* field is the string ‘ADA’ while that of the *LINK* field is the address 1000. To access these values, we write $INFO(\alpha)$ and $LINK(\alpha)$, respectively. Thus, we have $INFO(\alpha) = \text{‘ADA’}$ and $LINK(\alpha) = 1000$. If now we want to replace ‘ADA’ by ‘LEM’ we write $INFO(\alpha) \leftarrow \text{‘LEM’}$ (where ‘ \leftarrow ’ is read as ‘gets’).

Now consider the nodes shown below. Can you deduce an underlying structure?

3000:

ADA	1000
-----	------

 1000:

LEM	4000
-----	------

 4000:

LES	Λ
-----	-----------

When a field, in this instance the *LINK* field, of a node contains the address of another node, we say that the *LINK* field of the former contains a *pointer* to the latter. Such a relationship can be graphically represented by an arrow emanating from the former and directed to the latter node. Hence, an equivalent way of representing the linked nodes shown above is as the *linked list* shown below:



The value 3000 is assigned to the pointer variable α which then effectively becomes the pointer to the list. It is easy to see that the following relations are true for this list:

$INFO(\alpha) = \text{‘ADA’}$
 $INFO(LINK(\alpha)) = \text{‘LEM’}$
 $INFO(LINK(LINK(\alpha))) = \text{‘LES’}$

The symbol ' Λ ' in the *LINK* field of the last node in the list represents the special *null* address (nil in Pascal, NULL in C) which by convention is *not* the address of anything. In the present case we use ' Λ ' to identify the last node in the linked list; we will find other uses for ' Λ ' later.

As a simple exercise, what is the result of executing the following instructions on the list?

$$\begin{aligned}\beta &\leftarrow LINK(\alpha) \\ LINK(\alpha) &\leftarrow LINK(\beta) \\ LINK(\beta) &\leftarrow \Lambda\end{aligned}$$

A simple implementation in C

One way to implement in C the node structure shown above is by using a C *structure*, as in the following declaration:

```
typedef struct node Node;
struct node
{
    char INFO[3];
    Node *LINK;
};
```

If now we declare **alpha** to be a pointer to a structure of type **Node**, as in

```
Node *alpha;
```

then we access the fields (the structure elements) of the node pointed to by **alpha**, thus

```
alpha->INFO
alpha->LINK
```

which correspond to our notation $INFO(\alpha)$ and $LINK(\alpha)$, respectively.

Linked design: the memory pool

Whenever we use the linked design we assume the existence of a source of the nodes that we use to construct linked structures. In these Notes, we refer to this region of memory as the *memory pool*. We assume that we have available two procedures, viz., $GETNODE(\alpha)$ and $RETNODE(\alpha)$ which allow us to get a node from, and return a node to, the memory pool respectively. In $GETNODE$, α is a pointer to the node obtained from the pool; in $RETNODE$, α is a pointer to the node to be returned to the pool. We assume that $GETNODE$ issues an error message and returns control to the runtime system in case of unsuccessful allocation. To invoke these procedures, we write **call** $GETNODE(\alpha)$ or **call** $RETNODE(\alpha)$, as the case may be.

In C, the analog of procedures $GETNODE$ and $RETNODE$ are the routines **malloc()** and **free()**, respectively. The function **malloc()** allocates memory from a region of memory called the *heap* (the analog of our memory pool) and returns a pointer to it; the function **free()** returns previously allocated memory to the heap for possible reuse.

The following C statements illustrate the use of these functions for the structure declared above:

12 SESSION 1. Basic Concepts

```
alpha = malloc(sizeof(node)); /* allocate space for a node from the heap */
:
free(alpha); /* return node to the heap */
```

In our notation, the equivalent statements would be

```
call GETNODE( $\alpha$ )
:
call RETNODE( $\alpha$ )
```

1.5 Putting it all together: formulating a solution to the ESP

Consider again the exam-scheduling problem: We are given n courses and a list of students enrolled in each course. We want to come up with a schedule of examinations ‘without conflicts’ using a minimal number of time slots. Let us now apply the ideas we discussed above in formulating a solution to this problem.

To be more specific, consider the courses, and corresponding class lists, shown in Figure 1.7. For brevity, only initials are shown; we assume that each initial identifies a unique student. We have here the problem data in its ‘raw’ form. We can deduce from this raw data two kinds of information, viz., values of certain quantities and relationships among certain data elements. For instance, we can easily verify that there are 11 courses, that there are 12 students enrolled in C01, 14 in C02, etc.; that the maximum class size is 15, that the average class size is 14, and so forth. In terms of relationships, we can see that RDA and ALB are classmates, that RDA and MCA are not, that C01 and C02 have two students in common, that C01 and C03 have none, and so on. Clearly, not all of this information is useful in solving the ESP. Our first task, then, is to extract from the given raw data those quantities and those relationships which are relevant to the problem we are supposed to solve, and to set aside those which are not.

Course	S	t	u	d	e	n	t	s
C01	RDA	ALB	NAC	EDF	BMG	VLJ	IVL	LGM EGN KSO EST VIV
C02	MCA	EDF	SLF	ADG	BCG	AAI	RRK	LGM RLM JGP LRQ WAR KLS DDY
C03	ABC	BBD	GCF	ADG	AKL	BCL	MIN	JGP RSQ DBU IEY RAW ESZ
C04	ANA	JCA	CAB	NAC	GCF	GLH	VLJ	LLM MAN PEP PQQ ERR SET MAV REW
C05	BBC	EDD	HSE	ELG	ISH	JEI	EMJ	RRK TPL RER EPS AVU CDW ELY
C06	ALA	MCA	ABB	BCF	GLH	AKL	HGN	RON JGP ALQ EPR ABT KEV YEZ
C07	CDC	ISH	ABI	DHJ	ESM	FBM	RMN	PEP VIR JLS LOT MAV TEX
C08	AAA	HLA	BBD	WRE	ECG	HLH	DHJ	RON TSO PQQ MBT REW BAX TRY BDZ
C09	MCA	JCA	BCF	EGG	AAI	XTK	WIL	CSM HLO RSP APR RER JET DBU
C10	RDA	BBB	CLC	ECG	MNH	EMJ	JOK	ARM NFM EGN RCN RSP LEQ YIR AVU
C11	ADB	WDB	BKC	CLC	SDE	UKF	BMG	HRH BTK LGM QJP EPS KLS BST YNZ

Figure 1.7 Class lists for 11 courses

A careful analysis of the exam-scheduling problem should convince you that the fact that RDA and ALB are both taking C01 is not relevant to the ESP, but the fact that RDA is taking both C01 and C10 is. More to the point, the *really* relevant information is that there is a student who is taking both C01 and C10, thereby establishing a relationship between C01 and C10; the identity of the student, in this case RDA, is in itself immaterial. Thus, in solving the ESP, *it suffices to simply indicate which courses have common students*.

The process by which we set aside extraneous information from the given raw data and cull only that which is relevant to the problem we are solving is called *data abstraction*. Our next task is to devise a *data representation* for those aspects of the problem data that we have abstracted out. This is where the notion of an ADT plays an important role.

The CS literature is replete with ADT's which have been found to be extremely versatile tools in solving problems on a computer. Many of these derive from familiar mathematical structures such as sets, trees, graphs, tables, and the like. It is not very often that we need to invent an entirely new ADT; usually, all it takes is to see an old one in a new light. Take, for instance, the graph. A graph may be defined, for now, as *a set of vertices connected by edges*. Figure 1.8 depicts a graph.

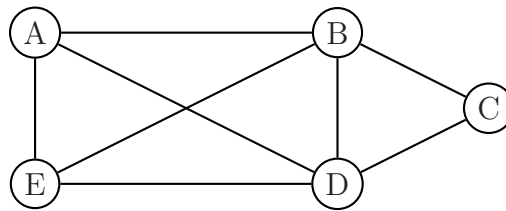


Figure 1.8 A graph on 5 vertices and 8 edges

Earlier, we found that to solve the ESP, it suffices to have a data representation which indicates which courses have common students. Can a graph serve this purpose? Indeed, it can, and in a most natural way. Specifically, we adopt the following conventions:

1. A vertex of the graph represents a course.
2. Two vertices (courses) are connected by an edge if one or more students are taking both courses. Such vertices are said to be *adjacent*.

Figure 1.9 shows the graph which represents the relevant relationships abstracted out from the data given in Figure 1.7 following the conventions stated above. For instance, vertices C01 and C02 are connected by an edge because two students, namely EDF and LGM, are taking both courses. In fact, LGM is also taking C11; hence, an edge joins vertices C01 and C11, and another edge joins vertices C02 and C11. Clearly, examinations for these three courses must be scheduled at different times. In general, the data representation shown in Figure 1.9 immediately identifies which courses *may not have* examinations scheduled at the same time; the crucial task, then, is to find a way to identify which courses *may have* examinations scheduled at the same time such that we minimize the number of time slots needed. You may want, at this point, to verify that the graph of Figure 1.9 does in fact capture all the relevant relationships implicit in the class lists of Figure 1.7.

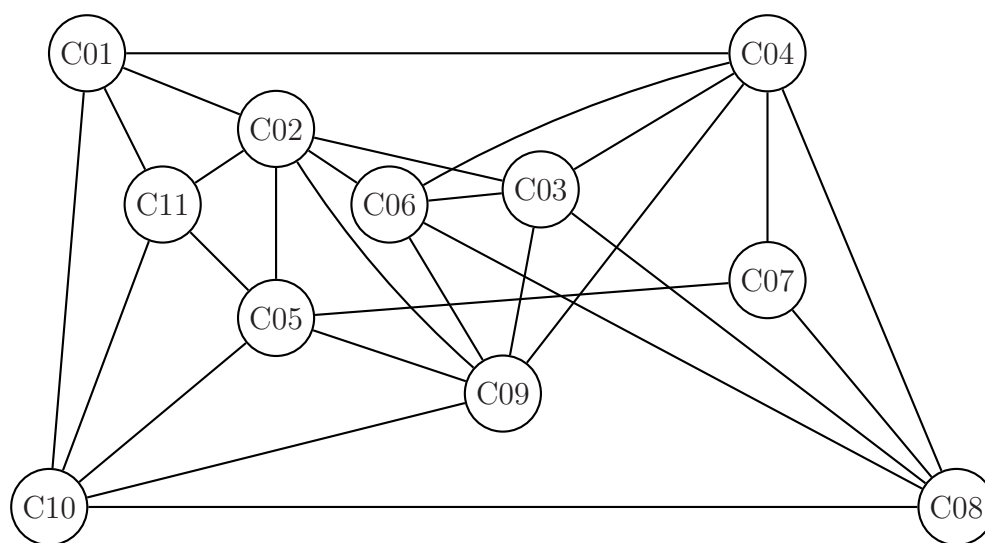


Figure 1.9 Graph representing the class lists of Figure 1.7

Now we get to the heart of the problem: using the graph of Figure 1.9, how do we determine which courses may have examinations scheduled at the same time? To this end, we apply a technique called *graph coloring*: we color the vertices of the graph such that no two adjacent vertices are assigned the same color, using the least number of colors. We then assign to the same time slot the examinations for all vertices (courses) which have the same color. In graph theory, the least number of colors needed to color a graph is called its *chromatic number*. In the context of the exam-scheduling problem, this number represents the minimum number of time slots needed for a conflict-free schedule. [Suggestion: Try coloring the graph of Figure 1.9; remember, no adjacent vertices may have the same color and use as few colors as possible.]

If you have colored the graph as suggested, you will have noticed that to obtain a *legal* coloring (no adjacent vertices are assigned the same color) is easy enough, but to obtain a coloring that is also *optimal* (uses as few colors as possible) is not quite that easy anymore. In fact, this latter requirement makes the graph coloring problem (and the ESP, as well) extremely difficult to solve; such a problem is said to be *intractable*. This problem has its beginnings in the middle of the nineteenth century (c. 1850); to date, the brightest minds in mathematics and in CS have yet to find an algorithm that is guaranteed to give an optimal solution, apart from the brute force approach in which we try all possibilities. Unfortunately, trying all possibilities takes exponential time in the size of the input, making this approach computationally very expensive even for graphs of ‘practical’ size. (More on this in Session 3.)

Actually, the graph coloring problem is just one in a large class of intractable problems called *NP-complete problems*, all of which take at least exponential time to solve. In practice, such problems are solved using algorithms which yield an approximate solution (for instance, one that is suboptimal) in a reasonable amount of time. Such an algorithm is called a *heuristic*.

For the graph coloring problem, the following heuristic produces a legal, though not necessarily optimal, coloring (AHO[1983], p. 5).

Algorithm G: Approximate graph coloring

1. Select an uncolored vertex and color it with a new color.
2. For each remaining uncolored vertex, say v , determine if v is connected by an edge to some vertex already colored with the new color; if it is not, color vertex v with the new color.
3. If there are still uncolored vertices, go to 1; else, stop.

Let us apply this algorithm to the graph of Figure 1.9. We choose C01 and color it red. C02, C04, C10 and C11 cannot be colored red since they are adjacent to C01; however, anyone among C03, C04 through C09 can be colored red. We choose C03 and color it red. Now, C06, C08 and C09 can no longer be colored red, but either C05 or C07 can be. We choose C05 and color it red; no more remaining uncolored vertex can be colored red. Next, we choose C02 and color it blue. C06, C09 and C11 cannot be colored blue, but any one among C04, C07, C08 and C10 can be. We choose C04 and color it blue. Now, C07 and C08 can no longer be colored blue, but C10 can be; we color C10 blue. Next, we choose C06 and color it green. C08 and C09 cannot be colored green, but C07 and C11 can be. We color C07 and C11 green. Finally, we color C08 and C09 yellow. Figure 1.10 summarizes the results.

Vertices	Color
C01, C03, C05	Red
C02, C04, C10	Blue
C06, C07, C11	Green
C08, C09	Yellow

Figure 1.10 A coloring of the graph of Figure 1.9

It turns out that the coloring of Figure 1.10 is optimal. To see this, note that vertices C02, C03, C06 and C09 constitute a subgraph in which there is an edge connecting every pair of vertices. Such a subgraph is called a *4-clique*. (Can you find another 4-clique in Figure 1.9?) Clearly, four colors are needed to color a 4-clique, so the coloring of Figure 1.10 is optimal. In terms of the original exam-scheduling problem, the colors are the time slots, and all vertices (courses) with the same color may have examinations assigned to the same time slot, for instance:

Courses	Schedule of exams
C01, C03, C05	Monday 8:30 – 11:30 AM
C02, C04, C10	Monday 1:00 – 4:00 PM
C06, C07, C11	Tuesday 8:30 – 11:30 AM
C08, C09	Tuesday 1:00 – 4:00 PM

Figure 1.11 An optimal solution to the ESP of Figure 1.7

16 SESSION 1. Basic Concepts

We have solved a particular instance of the ESP. But more important than the solution that we obtained (Figure 1.11) is the conceptual framework that we developed to arrive at the solution. This is more important because now we have the means to solve *any other* instance of the ESP. At the heart of this conceptual framework is a data representation (a graph) which captures the essential relationships among the given data elements, and an algorithm (to color a graph, albeit suboptimally) which produces the desired result (or something that approximates this). These two, together, constitute an abstract data type. While our ultimate goal is to implement on a computer the method of solution that we have formulated, we have deliberately left out, until now, any consideration of implementation issues. We have formulated a solution to the ESP without having to think in terms of any specific programming language.

Finally, with our method of solution firmly established, we get down to details of implementation. This task involves:

1. choosing a *data structure* to represent a graph in the memory of a computer
2. writing the *code* to convert Algorithm G into a running program

One obvious way to represent a graph on n vertices is to define a matrix, say M , of size n by n , such that $M(i, j) = 1$ (or *true*) if an edge connects vertices i and j , and 0 (or *false*), otherwise. This is called the *adjacency matrix* representation of a graph (more on this in Session 9). Figure 1.12 depicts the adjacency matrix M for the graph of Figure 1.9.

	<i>C01</i>	<i>C02</i>	<i>C03</i>	<i>C04</i>	<i>C05</i>	<i>C06</i>	<i>C07</i>	<i>C08</i>	<i>C09</i>	<i>C10</i>	<i>C11</i>
$M =$	$\begin{bmatrix}$	0	1	0	1	0	0	0	0	1	1
<i>C01</i>		1	0	1	0	1	1	0	0	1	1
<i>C02</i>		0	1	0	1	0	1	0	1	1	0
<i>C03</i>		1	0	1	0	0	1	1	1	0	0
<i>C04</i>		0	1	0	0	0	1	0	1	1	1
<i>C05</i>		0	1	1	1	0	0	1	1	0	0
<i>C06</i>		0	0	0	1	1	0	0	1	0	0
<i>C07</i>		0	0	1	1	0	1	0	0	1	0
<i>C08</i>		0	1	1	1	1	0	0	0	1	0
<i>C09</i>		1	0	0	0	1	0	1	1	0	1
<i>C10</i>		1	1	0	0	1	0	0	0	1	0
<i>C11</i>											

Figure 1.12 Adjacency matrix representation of the graph of Figure 1.9

The adjacency matrix representation of a graph is readily implemented as an *array* in many programming languages. For instance, we could specify the matrix M as follows:

Language	Array declaration
C	int M[11][11];
Pascal	var M:array[1..11,1..11] of 0..1;
FORTRAN	integer* 1 M(1:11,1:11)

Having chosen a particular data structure, in this case a two-dimensional array, to represent a graph, we finally set out to write the code to implement Algorithm G as a computer program. This is a non-trivial but straightforward task which is left as an exercise for you to do. With this, our solution to the exam-scheduling problem is complete.

Summary

- Solving a problem on a computer involves two intimately related tasks: the structuring of data and the synthesis of algorithms. *Data structures* and *algorithms* are the building blocks of computer programs.
- A set of data elements plus a set of operations defined on the data elements constitute an *abstract data type* (ADT). The word ‘abstract’ means an ADT is implementation independent.
- The implementation, or realization, of an ADT in terms of the constructs of a specific programming language is a *data structure*. The usual implementations are *sequential* (using arrays) and *linked* (using dynamic variables), or a mix of the two.
- Thinking in terms of ADT’s when conceptualizing a solution to a problem helps one cope with the complexities of the problem-solving process on a computer. Only after the solution has been completely formulated should one begin to transcribe it into a running program.
- An *algorithm* is a finite set of well-defined instructions for solving a problem in a finite amount of time. An algorithm can be carried out ‘by hand’ or executed by a computer.
- An algorithm must terminate after a finite amount of time. An algorithm which, though finite, takes years or centuries or millenia to execute is not finite enough to be of any practical use. An example of such an algorithm is Cramer’s rule for solving systems of linear equations. Much faster algorithms are available to solve such linear systems.
- The *only* algorithm that is guaranteed to yield an optimal solution to the exam-scheduling problem (ESP) would require an inordinately large amount of time even for inputs of moderate size. A problem such as the ESP is said to be *intractable*. The ESP can be solved suboptimally in a reasonable amount of time by using an approximate algorithm, or a *heuristic*.

Exercises

1. Consult a book on Automata Theory and make a list of five unsolvable problems. Consult a book on Algorithms and make a list of five tractable and five intractable problems.
2. What happens if initially $m < n$ in Euclid’s algorithm?

18 SESSION 1. Basic Concepts

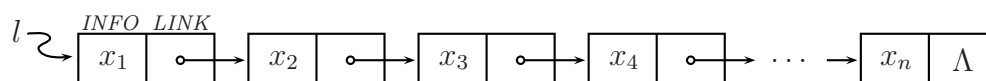
3. Apply Euclid's algorithm to find the GCD of the following pairs of integers.

- (a) 2057 and 1331 (b) 469 and 3982 (c) 610 and 377 (d) 611953 and 541

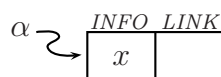
4. Give a recursive version of Euclid's algorithm.

5. Find the time required to solve n linear equations in n unknowns using Cramer's rule on a computer that can perform 1 *trillion* multiplications per second for $n = 19, 21, 22, 23$ and 24 .

6. Given a linked list pointed to by l



and a new node pointed to by α :



- Write an algorithm to insert the new node at the head of the list, i.e., node α becomes the first node of list l .
 - Write an algorithm to insert the new node at the tail of the list, i.e., node α becomes the last node of list l .
 - Assume that the nodes in list l are sorted in increasing order of the contents of the *INFO* field, i.e., $x_1 < x_2 < x_3 < \dots < x_n$. Write an algorithm to insert the new node into the list such that the list remains sorted.
7. (a) Draw the graph which represents the relevant information to solve the exam-scheduling problem for the class lists given below.
 (b) Apply Algorithm G to color the graph in (a). Is the coloring optimal?

```

C01  ADB WDB BKC OBC SDE UKF BMG NOH BTK LGM EPS KLS ODT YNZ
C02  BEB XEB CLC DMC TEE VLF CNG ISH  CUK MHM RKP FQS LMS CTT YNZ
C03  SEA OBC DMC FDG NOH FNJ KPK BSM OGM FHN SDN STP MFQ ZJR BWU
C04  NDA KDA CDF FHG BBI YUK XJL DTM LMO STP BQR SFR KFT ECU
C05  BBB IMA CCD XSE FDG IMH EIJ SPN UYO ERQ ODT SFW CBX USY CEZ
C06  DEC JTH BCI EIJ FTM GCM SNN QFP WJR KMS MPT NBV UFX
C07  BMA NDA BCB CDF HMH BLL IHN SPN KHP BMQ FQR BCT LFV ZFZ
C08  CCC FED ITE FMG JTH KFI FNJ SSK UQL PEP SFR FQS BWU DEW FMY
C09  BOA KDA DBB OBC HDF HMH WMJ MMM NBN QFP ERQ FSR TFT NBV SFW
C10  BCC CCD HDF BEG BLL CDL NJN KHP STQ ECU JFY SBW FTZ
C11  NDA FEF TMF BEG CDG BBI SSK MHM SMM KHP MSQ XBR LMS EEY
C12  SEA BMB OBC FEF CNG WMJ JWJ MHM FHN LTO PEP FTT WJV
  
```

8. Algorithm G is a *greedy algorithm* in that it tries to color as many uncolored vertices as it legally can with the current color before turning to another color. The idea is that this will result in as few colors as possible. Show that this technique does *not* necessarily yield an optimal coloring for the graph in Item 7(a).
9. Write a program, using a language of your choice, to implement the solution to the ESP as formulated in this session. Program input will consist of a set of class lists

in the manner of Figure 1.7. Program output should consist of: (a) the adjacency matrix for the graph which indicates which courses have common students, in the manner of Figure 1.12, and (b) the generated schedule of exams, in the manner of Figure 1.11. Test your program using: (a) the class lists in Item 7, and (b) the class lists given below.

```

C01  RDA ALB  NAC EDF  BMG VLJ  IVL  LGM EGN KSO EST  VIV
C02  MCA EDF  SLF  ADG BCG  AAI  RRK LGM RLM JGP  LRQ WAR KLS  DDY
C03  ABC BBD  GCF  ADG AKL  BCL MIN  JGP RSQ DBU IEY  RAW ESZ
C04  ANA JCA  CAB NAC  GCF GLH VLJ  LLM MAN PEP PQQ ERR SET  MAV REW
C05  BBC EDD  HSE ELG  ISH  JEI  EMJ RRK TPL  RER EPS  AVU CDW ELY
C06  ALA MCA ABB BCF  GLH AKL HGN RON JGP  ALQ EPR ABT KEV YEZ
C07  CDC ISH  ABI DHJ  ESM FBM RMN PEP  VIR  JLS  LOT MAV TEX
C08  AAA HLA  BBD WRE ECG HLH DHJ  RON TSO PQQ MBT REW BAX TRY BDZ
C09  MCA JCA  BCF EGG  AAI  XTK WIL  CSM HLO RSP APR RER JET DBU
C10  RDA BBB  CLC ECG MNH EMJ JOK  ARM NFM EGN RCN RSP  LEQ YIR AVU
C11  ADB WDB BKC CLC  SDE  UKF BMG HRH BTK LGM QJP EPS  KLS  BST YNZ

```

Bibliographic Notes

The basic concepts discussed in this introductory session can be found in most books on Data Structures. The references listed in the READINGS for this session are particularly recommended. Section 1.2 (*Blending Mathematics, Science and Engineering*) and section 1.3 (*The Search for Enduring Principles in Computer Science*) of STANDISH[1994] provide an excellent take-off point for our study of data structures and algorithms. Section 1.1 (*Algorithms*) of KNUTH1[1997] is a nifty exposition on the concept of an algorithm. Section 1.1 (*From Problems to Programs*) of AHO[1983] provides us with a template to follow as we solve problems on a computer. Our approach in solving the exam-scheduling problem (ESP) parallels that given in this reference in solving a traffic light controller problem.

NOTES

SESSION 2

Mathematical Preliminaries

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Solve problems involving the modulo, exponential, logarithmic and other elementary functions.
2. Solve problems involving relations, summations and simple recurrences.
3. Solve elementary counting problems.
4. Prove the validity, or otherwise, of mathematical formulas and propositions using formal methods of proof.

READINGS KNUTH1[1997], pp. 10–26, 39–50, 75–84; STANDISH[1994], pp. 689–731; CORMEN[2001], pp. 51–56, 1058–1060, 1070–1077, 1094–1096; any good college Algebra book

DISCUSSION

In addition to English (and occasionally Taglish) there are essentially two other ‘languages’ which we will use in these Notes — an ‘algorithmic language’ for communicating algorithms (to be described in Session 3) and the ‘language of mathematics’ for describing the properties of data structures and the behavior of algorithms. The level of the mathematics that we will need ranges from the elementary to the esoteric, from the familiar to the abstruse. When it is of the latter kind (which is beyond the scope of this book) we will simply give the end result (e.g., a formula) and refer you to where you will find a detailed discussion or derivation.

Fortunately, most of the mathematical notations, concepts and techniques that we will employ are of the elementary kind, and should be familiar to you from your courses in algebra and computer programming. We will review these briefly in this session, even as we point out those aspects which are particularly relevant to our present study. For instance you have used logarithmic and exponential functions before, but you probably did not appreciate at all the tremendous difference between their rates of growth. Now you should.

Some portions of the material in this session may be new to you. In that case, consider the presentation as a brief introduction to what we will take up in detail in subsequent sessions.

2.1 Mathematical notations and elementary functions

2.1.1 Floor, ceiling and mod functions

- (a) The **floor of x** , denoted $\lfloor x \rfloor$, is the greatest integer less than or equal to x , where x is any real number. For example:

$$\lfloor 3.14 \rfloor = 3, \quad \lfloor \frac{1}{2} \rfloor = 0, \quad \lfloor -\frac{1}{2} \rfloor = -1$$

- (b) The **ceiling of x** , denoted $\lceil x \rceil$, is the smallest integer greater than or equal to x , where x is any real number. For example:

$$\lceil 3.14 \rceil = 4, \quad \lceil \frac{1}{2} \rceil = 1, \quad \lceil -\frac{1}{2} \rceil = 0$$

- (c) For any two real numbers x and y , the **mod** function, denoted $x \bmod y$, is defined as

$$x \bmod y = \begin{cases} x & \text{if } y = 0 \\ x - y \left\lfloor \frac{x}{y} \right\rfloor & \text{if } y \neq 0 \end{cases} \quad (2.1)$$

For example:

$$10 \bmod 3 = 1, \quad 24 \bmod 8 = 0, \quad -5 \bmod 7 = 2, \quad 3.1416 \bmod 1 = 0.1416$$

The following formulas involving the floor, ceiling and mod functions are readily verified. For all real x, y and z , except as otherwise indicated:

- (a) $\lceil x \rceil = \lfloor x \rfloor = x$ iff x is an integer
- (b) $\lceil \frac{x}{2} \rceil + \lfloor \frac{x}{2} \rfloor = x$ iff x is an integer
- (c) $\lceil x \rceil = \lfloor x \rfloor + 1$ iff x is *not* an integer
- (d) $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- (e) $\lfloor -x \rfloor = -\lceil x \rceil$
- (f) $\lceil -x \rceil = -\lfloor x \rfloor$
- (g) $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
- (h) $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$
- (i) $\lfloor x \pm n \rfloor = \lfloor x \rfloor \pm n$ n an integer
- (j) $\lceil x \pm n \rceil = \lceil x \rceil \pm n$ n an integer
- (k) $x = \lfloor x \rfloor + x \bmod 1$
- (l) $z(x \bmod y) = zx \bmod zy$

We will use the following identities in Session 15. For integer p, q and m :

$$\begin{aligned} (p + q) \bmod m &= (p \bmod m + q \bmod m) \bmod m \\ (pq) \bmod m &= [(p \bmod m)(q \bmod m)] \bmod m \end{aligned} \quad (2.3)$$

2.1.2 Polynomials

A **polynomial in n of degree d** , where d is a positive integer, is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^d a_i n^i = a_0 + a_1 n + a_2 n^2 + \dots + a_d n^d \quad (2.4)$$

where a_0, a_1, \dots, a_d are the *coefficients* of the polynomial and $a_d \neq 0$. In most of our encounters with polynomials d is 1, 2 or 3 and $a_d > 0$, yielding

$$\begin{aligned} p(n) &= a_0 + a_1 n && \text{(linear in } n\text{)} \\ p(n) &= a_0 + a_1 n + a_2 n^2 && \text{(quadratic in } n\text{)} \\ p(n) &= a_0 + a_1 n + a_2 n^2 + a_3 n^3 && \text{(cubic in } n\text{)} \end{aligned}$$

2.1.3 Exponentials

An **exponential** is an expression of the form x^p where p is called an *exponent* or *power*. Listed below are some familiar rules for exponentials. For all real a, b, x and y :

$$\begin{aligned} (a) \quad & x^0 = 1 \quad x \neq 0 \\ (b) \quad & x^{-a} = \frac{1}{x^a} \\ (c) \quad & x^a x^b = x^{a+b} \\ (d) \quad & \frac{x^a}{x^b} = x^{a-b} \\ (e) \quad & (x^a)^b = x^{ab} \\ (f) \quad & (xy)^a = x^a y^a \\ (g) \quad & \left(\frac{x}{y}\right)^a = \frac{x^a}{y^a} \\ (h) \quad & x^{\frac{1}{n}} = \sqrt[n]{x} \quad \text{integer } n \geq 2 \end{aligned} \quad (2.5)$$

In most of our encounters with exponentials the exponent is an integer, say n , as in:

$$\begin{aligned} 2^n + 2^n &= 2 \cdot 2^n = 2^{n+1} \\ 8^n &= (2^3)^n = 2^{3n} \\ \frac{8^n}{4^n} &= \left(\frac{8}{4}\right)^n = 2^n \end{aligned}$$

The exponential function grows very fast, as shown by the following examples (verify):

$$\begin{aligned} 2^{10} &= 1024 \\ 2^{20} &= 1,048,576 \\ 2^{30} &= 1,073,741,824 \\ 2^{40} &= 1,099,511,627,776 \\ 2^{50} &= 1,125,899,906,842,624 \\ 2^{64} &= 18,446,744,073,709,351,616 \end{aligned}$$

2.1.4 Logarithms

Let $b > 1$ and q be positive real numbers. The real number p such that $q = b^p$ is called the **logarithm of q to the base b** :

$$\log_b q = p \quad \text{iff} \quad q = b^p \quad (2.6)$$

Note that a logarithm is an exponent; the logarithm of q to the base b is the exponent to which b must be raised to yield q . Hence an equivalent definition of the logarithm is

$$q = b^{\log_b q} \quad (2.7)$$

Listed below are some familiar identities for logarithms. For all positive real numbers $b > 1$, $c > 1$, x and y :

$$\begin{aligned} (a) \quad & \log_b 1 = 0 \\ (b) \quad & \log_b b = 1 \\ (c) \quad & \log_b x < x \\ (d) \quad & \log_b(xy) = \log_b x + \log_b y \\ (e) \quad & \log_b\left(\frac{x}{y}\right) = \log_b x - \log_b y \\ (f) \quad & \log_b x^y = y \log_b x \\ (g) \quad & \log_c x = \frac{\log_b x}{\log_b c} \\ (h) \quad & x^{\log_b y} = y^{\log_b x} \end{aligned} \quad (2.8)$$

In most of our encounters with the logarithm, the base b is equal to 2, as in the following examples:

$$\begin{aligned} \log_2 2^n &= n \\ \log_2 \frac{n}{2} &= \log_2 n - 1 \\ \log_2 \sqrt{n} &= \log_2 n^{\frac{1}{2}} = \frac{1}{2} \log_2 n \\ \log_{16} n &= \frac{\log_2 n}{\log_2 16} = \frac{1}{4} \log_2 n \\ 2^{\log_2 n} &= n \\ 2^{3 \log_2 n} &= (2^{\log_2 n})^3 = n^3 \end{aligned}$$

The logarithmic function grows very slowly, as shown below:

$$\begin{aligned} \log_2 1 &= 0 \\ \log_2 2 &= 1 \\ \log_2 1024 &= 10 \\ \log_2 1,048,576 &= 20 \\ \log_2 1,073,741,824 &= 30 \\ \log_2 1,099,511,627,776 &= 40 \\ \log_2 1,125,899,906,842,624 &= 50 \end{aligned}$$

Finally, the following notational shorthand is often used with logarithms:

$$\begin{aligned} \log_b^k x &\equiv (\log_b x)^k \\ \log_b \log_b x &\equiv \log_b(\log_b x) \end{aligned}$$

2.1.5 Factorials

For any integer $n \geq 0$, the **factorial of n** , denoted $n!$, is defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases} \quad (2.9)$$

The factorial function grows extremely fast, as shown by the following examples (verify):

$$\begin{aligned} 5! &= 120 \\ 10! &= 3,628,800 \\ 15! &= 1,307,674,368,000 \\ 20! &= 2,432,902,008,176,640,000 \\ 25! &= 15,551,210,043,330,985,984,000,000 \end{aligned}$$

An approximate value of $n!$ is given by *Stirling's approximation*

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n}\right) \quad (2.10)$$

where $e = 2.7182818284\dots$ is the base of the natural logarithms.

2.1.6 Fibonacci numbers

The Fibonacci numbers, F_n , $n \geq 0$ are defined by the recurrence

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-2} + F_{n-1} & \text{if } n \geq 2 \end{cases} \quad (2.11)$$

Here are the first 12 Fibonacci numbers:

$$\begin{array}{cccccccccccc} F_0 & F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & F_{10} & F_{11} \dots \\ 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 \dots \end{array}$$

There are hundreds of identities involving Fibonacci numbers. One formula that is particularly useful allows us to find F_n *directly*, i.e., we do not have to start from F_0 .

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \quad (2.12)$$

where

$$\phi = \frac{1}{2}(1 + \sqrt{5}) = 1.61803\dots \quad (2.13)$$

26 SESSION 2. Mathematical Preliminaries

is called the *golden ratio* or the *divine proportion* and $\hat{\phi} = \frac{1}{2}(1 - \sqrt{5}) = -0.61803\dots$ is the conjugate of ϕ . Since $|\hat{\phi}| < 1$, $\hat{\phi}^n$ decreases as n increases; thus ϕ^n accounts for most of F_n , as indicated in the following formula.

$$F_n = \begin{cases} \left\lfloor \frac{\phi^n}{\sqrt{5}} \right\rfloor & \text{if } n \text{ is even} \\ \left\lceil \frac{\phi^n}{\sqrt{5}} \right\rceil & \text{if } n \text{ is odd} \end{cases} \quad (2.14)$$

Eq.(2.14) also implies that

$$F_n > \frac{\phi^n}{\sqrt{5}} - 1 \quad (2.15)$$

We will make use of Fibonacci trees in the analysis of Fibonaccian search in Session 13 and Eq.(2.15) in particular in the analysis of AVL trees in Session 14.

2.2 Sets

A **set** is an *unordered* collection of *distinct* elements. A set is denoted by enclosing its elements in braces.

Example 2.1. The sets $\{a,b,c,d\}$, $\{b,a,d,c\}$, $\{a,b,b,a,c,d\}$ and $\{a,c,d,c,b\}$ are the same set. In the last two representations some of the elements have been unnecessarily listed once too often.

Example 2.2. The following sets of numbers are frequently encountered and are assigned the indicated symbolic names.

$$\mathbb{N} = \text{the set of } \textit{natural numbers} = \{0, 1, 2, 3, \dots\}$$

$$\mathbb{Z} = \text{the set of } \textit{integers} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$$

$$\mathbb{R} = \text{the set of } \textit{real numbers}$$

Set membership

Let P be the set $\{a, b, c, d\}$. We write $a \in P$ to indicate that ‘ a is a member of P ’ or that ‘ a is in P ’. We write $x \notin P$ to indicate that ‘ x is not in P ’.

Examples 2.1 and 2.2 illustrate two ways of specifying the members of a set, viz., by enumerating them all, or by listing enough elements to indicate a recognizable pattern. Another way is to specify the properties which the elements comprising the set must satisfy.

Example 2.3. To define the set of *rational numbers* we could write

$$Q = \{a/b \mid a, b \in \mathbb{Z}, b \neq 0\}$$

where ‘|’ is read ‘such that’. Note that in this particular case, the set Q is defined in terms of the set \mathbb{Z} .

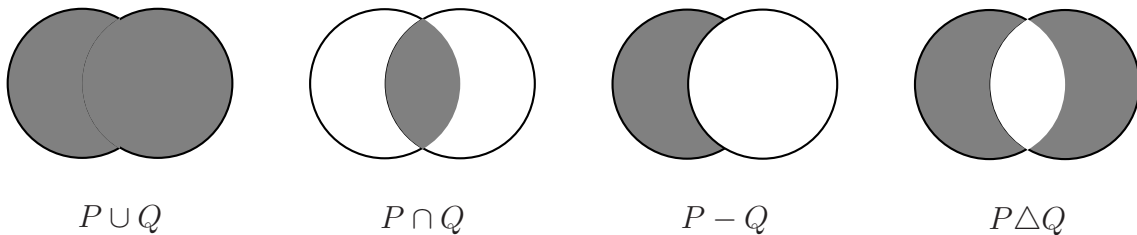
Cardinality of a set

The number of elements in a set S , denoted $|S|$, is called the *cardinality* or *size* of S . A set is *finite* if its cardinality is a natural number; otherwise, it is *infinite*. A set with no elements is called the *empty* or *null* set, denoted \emptyset or $\{\}$. It follows that $|\emptyset| = 0$. A set with only one element is called a *singleton*. To indicate that a finite set has cardinality n , we refer to it as an n -set.

Basic set operations

Given two sets P and Q , we can generate new sets by applying one or more of the following basic *set operations*.

$$\begin{aligned}
 P \cup Q & \text{ (the \textit{union} of } P \text{ and } Q) & = & \{x \mid x \in P \text{ or } x \in Q\} \\
 P \cap Q & \text{ (the \textit{intersection} of } P \text{ and } Q) & = & \{x \mid x \in P \text{ and } x \in Q\} \\
 P - Q & \text{ (the \textit{difference} of } P \text{ and } Q) & = & \{x \mid x \in P \text{ and } x \notin Q\} \\
 P \Delta Q & \text{ (the \textit{symmetric difference} of } P \text{ and } Q) & = & \{x \mid x \in P \cup Q \text{ and } x \notin P \cap Q\}
 \end{aligned}$$



Subsets and set equality

Let P and Q be sets. We say that P is a *subset* of Q , denoted $P \subseteq Q$, if every element of P is also an element of Q . P is a *proper subset* of Q , denoted $P \subset Q$, if in addition Q has an element that is not in P . Every set is a subset of itself, thus $P \subseteq P$. The null set is a subset of any set, thus $\emptyset \subseteq P$.

P and Q are *equal*, denoted $P = Q$, iff $P \subseteq Q$ and $Q \subseteq P$. In other words, the sets P and Q are equal if they have exactly the same elements.

In applications involving sets, the elements of any set are implicitly assumed to be members of some larger set \mathbb{U} called the *universe*. Thus $P \subseteq \mathbb{U}$ and $Q \subseteq \mathbb{U}$. The *complement* of P , denoted \overline{P} , is the set of all elements in \mathbb{U} but not in P , i.e., $\overline{P} = \mathbb{U} - P$.

Partition of a set

Two sets P and Q are *disjoint* if they have no elements in common, that is, $P \cap Q = \emptyset$. Let P_1, P_2, \dots, P_n be non-null subsets of P . The set $\{P_1, P_2, \dots, P_n\}$ is a *partition* of P iff (a) $P_i \cap P_j = \emptyset$, $1 \leq i, j \leq n$, $i \neq j$ and (b) $P = \bigcup_{i=1}^n P_i$. In plain English: $\{P_1, P_2, \dots, P_n\}$ is a partition of P if every element in P appears in exactly one P_i .

Example 2.4. Let $P = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Each of the following sets is a partition of P . (In each case, how would you describe the partitioning?)

$$\{\{0\}, \{1, 3, 5, 7, 9\}, \{2, 4, 6, 8, 10\}\}$$

$$\{\{0, 1\}, \{2, 3, 5, 7\}, \{4, 6, 8, 9, 10\}\}$$

$$\{\{0, 1, 2, 3, 5, 8\}, \{4, 6, 7, 9, 10\}\}$$

$$\{\{1, 2, 4, 8\}, \{0, 3, 5, 6, 7, 9, 10\}\}$$

2.3 Relations

Let P and Q be sets. The *Cartesian product*, or *cross product*, of P and Q , denoted $P \times Q$, is the set

$$P \times Q = \{\langle p, q \rangle \mid p \in P, q \in Q\} \quad (2.16)$$

The element $\langle p, q \rangle$ is called an *ordered pair*. The term ‘ordered’ signifies that the order of the quantities inside the $\langle \rangle$ is meaningful; thus, $\langle p, q \rangle$ and $\langle q, p \rangle$ are two different ordered pairs. By the same token, for $\langle p, q \rangle, \langle r, s \rangle \in P \times Q$, $\langle p, q \rangle = \langle r, s \rangle$ iff $p = r$ and $q = s$.

Example 2.5. Let $P = \{x, y\}$ and $Q = \{x, y, z\}$. Then,

$$P \times Q = \{\langle x, x \rangle, \langle x, y \rangle, \langle x, z \rangle, \langle y, x \rangle, \langle y, y \rangle, \langle y, z \rangle\}$$

$$Q \times P = \{\langle x, x \rangle, \langle x, y \rangle, \langle y, x \rangle, \langle y, y \rangle, \langle z, x \rangle, \langle z, y \rangle\}$$

Note that $P \times Q \neq Q \times P$.

Example 2.6. Let $S = \{1, 2, 3, 4\}$. Then,

$$S \times S = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 1 \rangle, \langle 4, 2 \rangle, \langle 4, 3 \rangle, \langle 4, 4 \rangle\}$$

It should be clear from the above examples that

$$\begin{aligned} |P \times Q| &= |Q \times P| = |P| \cdot |Q| \\ |S \times S| &= |S|^2 \end{aligned}$$

provided that P , Q and S are finite sets.

Binary relations

Let P , Q and S be sets. A **binary relation**, say R , on P and Q is a subset of the Cartesian product $P \times Q$. Similarly, a binary relation R on S is a subset of the Cartesian product $S \times S$. We will write $x R y$ (read: ‘ x is related to y ’) to signify that $\langle x, y \rangle \in R$.

Example 2.7. Let $S = \{1, 2, 3, 4\}$. Then, the ‘ $<$ ’ relation R on S is the set

$$R = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle\}$$

which is a subset of $S \times S$ (see Ex. 2.6 above). For instance, the ordered pair $\langle 2, 3 \rangle$ is in R because $2 < 3$, but the ordered pair $\langle 1, 1 \rangle$ is not in R because $1 \not< 1$.

Example 2.8. In general, we can define the ‘ $<$ ’ relation R on \mathbb{R} (the set of real numbers) thus

$$R = \{\langle x, y \rangle \mid x, y \in \mathbb{R} \text{ and } x < y\}$$

Example 2.9. Let $P = \{p_1, p_2, \dots, p_m\}$ be a set of mothers and $Q = \{q_1, q_2, \dots, q_n\}$ be a set of children. We can define a relation $R \subseteq P \times Q$ thus

$$R = \{\langle p_i, q_j \rangle \mid p_i \text{ is the mother of } q_j\}$$

Properties of binary relations

1. **Reflexivity** — a binary relation R on S is *reflexive* if

$$x R x$$

for all $x \in S$. For example, ‘ $=$ ’ and ‘ \leq ’ are reflexive relations on \mathbb{R} , but ‘ $<$ ’ is not.

2. **Symmetry** — a binary relation R on S is *symmetric* if

$$x R y \text{ implies } y R x$$

for all $x, y \in S$. For example, ‘ $=$ ’ is a symmetric relation on \mathbb{R} but ‘ $<$ ’ and ‘ \leq ’ are not. The relation ‘*is a brother of*’ is symmetric on the set of all males but is not symmetric on the set of all people.

3. **Transitivity** — a binary relation R on S is *transitive* if

$$x R y \text{ and } y R z \text{ implies } x R z$$

for all $x, y, z \in S$. For example, ‘ $=$ ’, ‘ $<$ ’ and ‘ \leq ’ are transitive relations on \mathbb{R} . The relation ‘*is a prerequisite to*’ is transitive on the set of courses in the CS curriculum, but the relation ‘*is the mother of*’ is not transitive on any set of people.

30 SESSION 2. Mathematical Preliminaries

4. **Irreflexivity** — a binary relation R on S is *irreflexive* if

$$x \not R x$$

or equivalently $\langle x, x \rangle \notin R$, for all $x \in S$. For example, ' $<$ ' is an irreflexive relation on \mathbb{R} since $x \not< x$ for any real number x .

5. **Antisymmetry** — a binary relation R on S is *antisymmetric* if

$$x R y \quad \text{and} \quad y R x \quad \text{implies} \quad x = y$$

for all $x, y \in S$. For example, ' \leq ' is an antisymmetric relation on \mathbb{R} since $x \leq y$ and $y \leq x$ imply $x = y$.

Example 2.10. Let $\mathbb{Z}^+ = \{x \mid x \in \mathbb{Z}, x > 0\}$ be the set of positive integers. Now define the relation R by

$$x R y \quad \text{if} \quad x \text{ exactly divides } y$$

for $x, y \in \mathbb{Z}^+$. We claim that R is a reflexive, antisymmetric and transitive relation on \mathbb{Z}^+ .

Proof:

- (a) Reflexivity: Clearly x exactly divides x for all positive integers x .
- (b) Antisymmetry: We need to show that if $x R y$ and $y R x$ then $x = y$ for $x, y \in \mathbb{Z}^+$. In plain English: we need to show for positive integers x and y that if x exactly divides y and y exactly divides x , then x equals y . This is obvious, but let us prove it anyway.

$x R y$ implies that $y = ax$ for some positive integer a and $y R x$ implies that $x = by$ for some positive integer b . Substituting we have $y = ax = aby$ which implies that $ab = 1$ since $y \neq 0$. Likewise, since a and b are positive integers, $ab = 1$ implies that $a = b = 1$. Hence $x = y$.

- (c) Transitivity: We need to show that if $x R y$ and $y R z$ then $x R z$ for $x, y, z \in \mathbb{Z}^+$. In plain English: we need to show for positive integers x, y and z that if x exactly divides y and y exactly divides z , then x exactly divides z . This is also obvious, but let us prove it anyway.

$x R y$ implies that $y = ax$ for some positive integer a and $y R z$ implies that $z = by$ for some positive integer b . Substituting we have $z = by = bax$ for some positive integer ba . Hence, $x R z$.

Note that R is *not* symmetric; for instance, 3 exactly divides 9 but 9 obviously does not exactly divide 3.

Example 2.11. For $x, y, n \in \mathbb{Z}, n > 0$, we say that ‘ x is congruent to y modulo n ’, denoted $x \equiv y \pmod{n}$, if $x \bmod n = y \bmod n$, or equivalently, if $x - y$ is an integral multiple of n . Define the relation R by

$$x R y \quad \text{if} \quad x - y \text{ is an integral multiple of } n$$

for $x, y, n \in \mathbb{Z}, n > 0$. For instance, with $n = 5$ we have $12 R 2$, $-7 R 8$, $20 R 0$ and $21 R -4$ since 10, -15 , 20 and 25 are multiples of 5. Equivalently, we find that $12 \bmod 5 = 2 \bmod 5 = 2$, $-7 \bmod 5 = 8 \bmod 5 = 3$, $20 \bmod 5 = 0 \bmod 5 = 0$, and $21 \bmod 5 = -4 \bmod 5 = 1$ (verify). Now, we claim that R is a reflexive, symmetric and transitive relation on \mathbb{Z} .

Proof:

- (a) Reflexivity: Clearly, $x - x = 0$ is a multiple of n for all integers x .
- (b) Symmetry: We need to show that $x R y$ implies $y R x$ for $x, y \in \mathbb{Z}$. In plain English: for integers x, y and $n > 0$, if $x - y$ is an integral multiple of n then $y - x$ is also an integral multiple of n . Once again, this is quite obvious but let us prove it anyway.

Let $x - y = cn$ for some integer c . Then, $y - x = -(x - y) = -cn$.

- (c) Transitivity: We need to show that $x R y$ and $y R z$ implies $x R z$ for $x, y, z \in \mathbb{Z}$. In plain English: for integers x, y, z and $n > 0$, if $x - y$ is an integral multiple of n and $y - z$ is an integral multiple of n then $x - z$ is also an integral multiple of n . To prove:

Let $x - y = cn$ and $y - z = dn$ for some integers c and d . Then, $x - z = x - y + dn = cn + dn = (c + d)n$.

A relation that is reflexive, antisymmetric and transitive, such as the relation ‘ x exactly divides y ’ on the set of positive integers in Example 2.10, is called a **partial order**. Other examples of partial order are the relations

$$x R y \quad \text{if} \quad x \leq y \quad \text{for } x, y \in \mathbb{R}$$

$$p R q \quad \text{if} \quad p \subseteq q \quad \text{for } p, q \subset \mathbb{U}$$

A partial order R on a set S is a **total order** if either $x R y$ or $y R x$ for all $x, y \in S$; in such a case, x and y are said to be *comparable*. Thus R is a total order if every pair of elements in S are comparable. For example, the relation ‘ \leq ’ on \mathbb{Z} is a total order since for all integers x and y we have either $x \leq y$ or $y \leq x$. However, the relation ‘exactly divides’ on \mathbb{Z}^+ is *not* a total order since, for instance, 4 does not exactly divide 7 and 7 does not exactly divide 4 (in such a case 4 and 7 are said to be *incomparable*).

A relation that is reflexive, symmetric and transitive, such as the relation ‘ $x - y$ is an integral multiple of n ’ on the set of integers in Example 2.11, is called an **equivalence relation**. We will encounter these three very important relations in subsequent sessions, and also in some of your other CS and EEE courses.

2.4 Permutations and Combinations

A recurring question in our study of data structures and algorithms is ‘How many?’ For instance, we might ask ‘How many distinct binary trees can we construct given n unlabelled nodes?’ or ‘In how many ways can we arrange n distinct elements?’ Sometimes finding the answer to such counting problems requires no more than the application of plain common sense; at other times finding the answer requires no less than deep mathematical insight or the application of clever mathematical techniques. In any case, we definitely would not want to have to list all possibilities and then count them one by one! You will take up counting theory in a course on Discrete Mathematics; for our present purposes the following simple rules and combinatorial formulas should help us deal with such problems.

2.4.1 Rule of sum and rule of product

In FORTRAN IV, a variable name is defined as ‘a sequence of at most six *alphanumeric* (i.e., uppercase alphabetic or numeric) characters, the first of which is alphabetic.’ We ask, ‘What is the total number of distinct FORTRAN IV names, say n_t , that we can construct?’ If we let $n_i, 1 \leq i \leq 6$, denote the number of names with exactly i characters, then clearly we have

$$n_t = n_1 + n_2 + n_3 + n_4 + n_5 + n_6$$

where $n_1 = 26$ since the first, and only, character in a 1-character name must be alphabetic. What is n_2 ? Since the second character in a 2-character name is either alphabetic or numeric, we can choose this character in one of $26 + 10 = 36$ ways. This is an application of the so-called *rule of sum*.

Rule of sum: Let P and Q be finite, disjoint sets. The number of ways to choose an element from P or Q is equal to the number of elements in P plus the number of elements in Q . In symbols: $|P \cup Q| = |P| + |Q|$.

Since we can choose the first character in one of 26 ways, and for each such choice, we can choose the second character in one of 36 ways, it follows that the number of distinct 2-character names is $n_2 = 26 \times 36$. This is an application of the so-called *rule of product*.

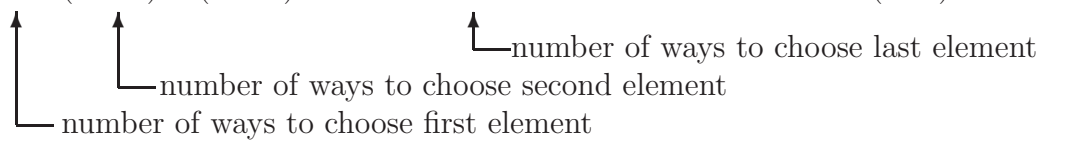
Rule of product: Let P and Q be finite sets. The number of ways to choose an ordered pair from P and Q is equal to the number of ways to choose the first element times the number of ways to choose the second element. In symbols: $|P \times Q| = |P| \cdot |Q|$.

By the rule of sum we find that there are $26 + (26 \times 36)$ names with at most two characters (here P is the set of 1-character names and Q is the set of 2-character names). By the rule of product we find that there are $(26 \times 36) \times 36$ names with exactly three characters (here P is the set of 2-character names and Q is the set of alphanumeric characters). By a repeated application of these two rules we find that the number of distinct FORTRAN IV names is

$$\begin{aligned} n_t &= 26 + (26 \times 36) + (26 \times 36) \times 36 + (26 \times 36^2) \times 36 + (26 \times 36^3) \times 36 + (26 \times 36^4) \times 36 \\ &= 1,617,038,306 \end{aligned}$$

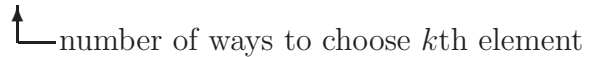
2.4.2 Permutations

Let S be a finite set. A **permutation** of S is an *arrangement* of all the elements of S . The term ‘arrangement’ signifies that the *order* of the elements in a permutation is *relevant*. For instance, if $S = \{a, b, c\}$, then abc and acb are two different permutations of S . There are actually six permutations of the set S , namely, abc, acb, bac, bca, cab and cba . In general, for an n -set S , there are $n!$ permutations of S . Denote this quantity $P(n, n)$; then

$$P(n, n) = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1 = n! \quad (2.17)$$


In arriving at Eq.(2.17) we have simply applied the rule of product repeatedly.

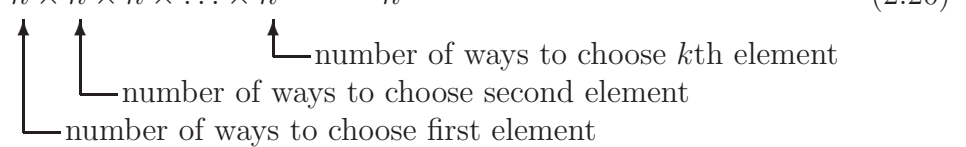
A **k-permutation** of S is an ordered sequence of k *distinct* elements of S . For instance, the 2-permutations of $S = \{a, b, c\}$ are ab, ba, ac, ca, bc and cb . The number of k -permutations of an n -set S , denoted $P(n, k)$ is simply

$$P(n, k) = n \times (n-1) \times (n-2) \times \dots \times [n - (k-1)] \quad (2.18)$$


An alternative expression for $P(n, k)$ can be derived by multiplying the right-hand side of Eq.(2.18) by $\frac{(n-k)!}{(n-k)!}$, thus:

$$\begin{aligned} P(n, k) &= n \times (n-1) \times \dots \times [n - (k-1)] \cdot \frac{(n-k)!}{(n-k)!} \\ &= \frac{n \times (n-1) \times \dots \times [n - (k-1)] \times (n-k) \times [n - (k+1)] \times \dots \times 2 \times 1}{(n-k)!} \\ &= \frac{n!}{(n-k)!} \end{aligned} \quad (2.19)$$

In Eqs.(2.17) and (2.18), the k elements are assumed to be distinct. Let us denote by $P_r(n, k)$ the number of k -permutations of S in which *repetitions are allowed* (such permutations are called *redundant permutations*); then, by the rule of product we have

$$P_r(n, k) = n \times n \times n \times \dots \times n = n^k \quad (2.20)$$


For instance, there are $3^2 = 9$ redundant 2-permutations of $S = \{a, b, c\}$, namely, $aa, ab, ac, ba, bb, bc, ca, cb$ and cc .

2.4.3 Combinations

Let S be a finite set. A **combination** of S is a *selection* of the elements of S . The term ‘selection’ signifies that the order of the elements in a combination is *irrelevant*. For instance, if $S = \{a, b, c\}$, then abc and acb are the same combination. In fact, all $3! = 6$ permutations of S correspond to only one combination.

A **k-combination** of S is a selection of k *distinct* elements of S . For instance, the 2-combinations of $S = \{a, b, c\}$ are ab, ac and bc . The number of k -combinations of an n -set S , denoted $C(n, k)$ or alternatively $\binom{n}{k}$ (read: ‘ n choose k ’), can be expressed in terms of the number of k -permutations of S . Since for every combination of k elements there are $k!$ permutations of the same elements, it follows that there are $k!$ times more k -permutations than there are k -combinations. Hence we have

$$C(n, k) \equiv \binom{n}{k} = \frac{P(n, k)}{k!} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n \quad (2.21)$$

Note that Eq.(2.21) is symmetric in n and $n - k$, that is

$$\binom{n}{k} = \binom{n}{n-k} \quad (2.22)$$

since

$$\binom{n}{n-k} = \frac{n!}{(n-k)![n-(n-k)]!} = \frac{n!}{(n-k)!k!} = \binom{n}{k}$$

Applying Eq.(2.21) we find that

$$\binom{n}{0} = \binom{n}{n} = 1 \quad (2.23)$$

which simply means that there is only one way to choose none or all of the n elements from a set of n elements.

In Eq.(2.21), the k elements are assumed to be distinct. Let us denote by $C_r(n, k)$ the number of k -combinations of an n -set S in which *repetitions are allowed* (such combinations are called *redundant combinations*). To find the expression for $C_r(n, k)$, we note that in any k -combination with repetitions, an element may be repeated at most $k - 1$ times. If we take these repeated elements as if they were distinct, we get the equivalent problem of finding the number of k -combinations of an $(n + k - 1)$ -set *without repetitions*. From Eq.(2.21) this is $C(n + k - 1, k)$; hence we have

$$C_r(n, k) = C(n + k - 1, k) = \binom{n + k - 1}{k} \quad (2.24)$$

For example, the number of redundant 4-combinations of $S = \{a, b, c\}$ is

$$\binom{3 + 4 - 1}{4} = \binom{6}{4} = \frac{6!}{4!2!} = 15$$

The combinations are: $aaaa, aaab, aaac, aabb, aabc, aacc, bbbb, bbba, bbbc, bbac, bbcc, cccc, ccca, cccb$ and $ccab$.

2.5 Summations

Let a_1, a_2, \dots, a_n be any sequence of numbers. We denote the sum $a_1 + a_2 + \dots + a_n$ as

$$\sum_{k=1}^n a_k = a_1 + a_2 + a_3 + \dots + a_n \quad (2.25)$$

If $n \leq 0$, the value of the summation is defined to be zero.

2.5.1 Arithmetic series

The ordered sequence $a_1, a_2, \dots, a_k, a_{k+1}, \dots, a_n$ is an **arithmetic series** if the difference d between *every* two consecutive terms is a constant, i.e., $d = a_{k+1} - a_k, 1 \leq k < n$. Let the first and last terms of an arithmetic series be a_1 and $a_n \equiv a_1 + nd$, respectively, and let S_A denote the sum of the series. Then

$$S_A = a_1 + (a_1 + d) + (a_1 + 2d) + \dots + a_n = n \left(\frac{a_1 + a_n}{2} \right) \quad (2.26)$$

This formula is readily obtained by writing the series forward and backward and then adding corresponding terms:

$$\begin{array}{rcl} S_A & = & a_1 + (a_1 + d) + (a_1 + 2d) + \dots + (a_n - 2d) + (a_n - d) + a_n \\ S_A & = & a_n + (a_n - d) + (a_n - 2d) + \dots + (a_1 + 2d) + (a_1 + d) + a_1 \\ \hline 2S_A & = & (a_1 + a_n) + (a_1 + a_n) + (a_1 + a_n) + \dots + (a_1 + a_n) + (a_1 + a_n) + (a_1 + a_n) \end{array}$$

Dividing both sides of the last equation by 2 yields Eq.(2.26).

A particular arithmetic series that occurs frequently in the analysis of algorithms is the sum of the first n positive integers:

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} \quad (2.27)$$

Any arithmetic series can be converted into a form that involves this particular series and then evaluated using Eq.(2.27).

We often encounter arithmetic series when we determine the number of times that an instruction or a group of instructions is executed within a loop or nested loops. Consider, for instance the two segments of code shown below which utilize the familiar **for** construct for looping. How many times is the instruction I executed?

Example 2.12.

```

for  $i \leftarrow 1$  to  $n$  do
  for  $j \leftarrow i$  to  $n$  do
     $I$ 
  endfor
endfor

```

36 SESSION 2. Mathematical Preliminaries

Let T be the number of times that I is executed; then we have:

$$\begin{aligned} T &= \sum_{i=1}^n \sum_{j=i}^n 1 \\ &= \sum_{i=1}^n (n - i + 1) = n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2} \end{aligned}$$

Example 2.13.

```

for  $i \leftarrow 0$  to  $n$  do
  for  $j \leftarrow 1$  to  $3i + 1$  do
     $I$ 
  endfor
endfor

```

$$\begin{aligned} T &= \sum_{i=0}^n \sum_{j=1}^{3i+1} 1 \\ &= \sum_{i=0}^n (3i + 1) = \underbrace{1 + 4 + 7 + 10 + \dots + (3n + 1)}_{n+1 \text{ terms}} \\ &= (n + 1) \left[\frac{1 + (3n + 1)}{2} \right] \quad \text{by Eq.(2.26)} \\ &= \frac{(n + 1)(3n + 2)}{2} \end{aligned}$$

Alternatively, we could have recast the series into a form such that Eq.(2.27) can be used:

$$\begin{aligned} T &= \sum_{i=0}^n \sum_{j=1}^{3i+1} 1 \\ &= \sum_{i=0}^n (3i + 1) = 3 \sum_{i=0}^n i + \sum_{i=0}^n 1 \\ &= 3 \left[\frac{n(n+1)}{2} \right] + (n + 1) \\ &= \frac{(n + 1)(3n + 2)}{2} \end{aligned}$$

2.5.2 Geometric series

The ordered sequence $a_1, a_2, \dots, a_k, a_{k+1}, \dots, a_n$ is a **geometric series** if the ratio r of every two consecutive terms is a constant, i.e., $r = \frac{a_{k+1}}{a_k}, 1 \leq k < n$. The geometric series is increasing if $r > 1$ and decreasing if $r < 1$.

Let the first and last terms of a geometric series be a_1 and $a_n \equiv a_1 r^{n-1}$, respectively, and let S_G denote the sum of the series. Then

$$S_G = \underbrace{a_1 + a_1 r + a_1 r^2 + a_1 r^3 + \dots + a_1 r^{n-1}}_{n \text{ terms}} = a_1 \left(\frac{r^n - 1}{r - 1} \right) \quad (2.28)$$

This formula is readily obtained as follows:

$$\begin{array}{r} r S_G = \quad \quad a_1 r + a_1 r^2 + a_1 r^3 + \dots + a_1 r^{n-1} + a_1 r^n \\ S_G = a_1 + a_1 r + a_1 r^2 + a_1 r^3 + \dots + a_1 r^{n-1} \\ \hline r S_G - S_G = -a_1 + 0 + 0 + 0 + \dots + 0 + a_1 r^n \end{array}$$

Solving the last equation for S_G yields Eq.(2.28).

A particular geometric series (in which $a_1 = 1$ and $r = x$) that occurs frequently in the analysis of algorithms is the sum

$$\sum_{i=0}^n x^i = \underbrace{1 + x + x^2 + x^3 + \dots + x^n}_{n+1 \text{ terms}} = \frac{x^{n+1} - 1}{x - 1}, \quad x \neq 1 \quad (2.29)$$

Eq.(2.29) is also called an **exponential series**.

Example 2.14. Find the decimal equivalent of 11111111_2 .

$$\begin{aligned} 11111111_2 &= 2^0 + 2^1 + 2^2 + \dots + 2^6 + 2^7 \\ &= \sum_{i=0}^7 2^i = 2^8 - 1 = 255_{10} \end{aligned}$$

2.5.3 Harmonic series

The n th harmonic number of order 1, $n \geq 0$, is defined as

$$H_n = \sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \approx \log_e n + \gamma \quad (2.30)$$

where $\gamma = 0.5772156649\dots$ is Euler's constant. We will encounter harmonic numbers in Session 13 in the analysis of linear search. Eq.(2.30) is called the **harmonic series**.

2.5.4 Miscellaneous sums

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad (2.31)$$

$$\sum_{i=1}^n i^3 = \left[\sum_{i=1}^n i \right]^2 = \left[\frac{n(n+1)}{2} \right]^2 \quad (2.32)$$

$$\sum_{i=1}^n \frac{i}{2^i} = 2 - \frac{n+2}{2^n} \quad (2.33)$$

$$\begin{aligned} \sum_{i=1}^n [\log_2 i] &= 0 + \underbrace{1+1}_{2 \text{ 1's}} + \underbrace{2+2+2+2}_{4 \text{ 2's}} + \underbrace{3+\dots+3}_{8 \text{ 3's}} + \dots \\ &= (n+1)[\log_2(n+1)] - 2^{\lceil \log_2(n+1) \rceil + 1} + 2 \end{aligned} \quad (2.34)$$

2.6 Recurrences

A **recurrence** is an equation that expresses a function in terms of itself. The following equations are recurrences:

$$(a) \quad F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-2) + F(n-1) & \text{if } n \geq 2 \end{cases}$$

$$(b) \quad C(n) = \begin{cases} 0 & \text{if } n = 1 \\ C(n-1) + n - 1 & \text{if } n \geq 2 \end{cases}$$

$$(c) \quad M(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \times M(n-1) + 1 & \text{if } n \geq 2 \end{cases}$$

We take note of the following facts regarding Eqs.(a) through (c):

1. The value of the function is known for certain values of the parameter n . These are called *boundary conditions*.
2. When the function appears on both sides of the equation, the value of the parameter on the right-hand side (RHS) is smaller than that on the left-hand side (LHS).

We encounter recurrences when we deal with recursively defined functions as in Eq.(a) which defines the Fibonacci numbers, and when we analyze the behavior of a recursive algorithm (or an algorithm that is implemented as a recursive procedure). For example, Eq.(b) denotes the number of *comparisons* performed by bubble sort to sort n keys, and Eq.(c) denotes the number of *moves* needed to solve the Towers of Hanoi problem for n disks.

To solve a recurrence equation is to express the function on the LHS *explicitly* in terms of its parameter(s), as in the following solutions to the recurrence equations listed above.

$$\begin{aligned} (a) \quad F(n) &= \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right] \\ (b) \quad C(n) &= \frac{(n-1)n}{2} \\ (c) \quad M(n) &= 2^n - 1 \end{aligned}$$

Solving recurrences is not always an easy task. Even a ‘simple looking’ recurrence such as Eq.(a) turns out to be rather difficult to solve. The solution, which we encountered earlier as Eq.(2.12), was first discovered by the French-English mathematician Abraham de Moivre (1667-1754) who used, for the first time, a new mathematical entity called a *generating function* to obtain the solution. (For a complete derivation of Eq.(a), see KNUTH1[1997], pp. 82–83.) On the other hand, the solutions to recurrences (b) and (c) are easily obtained, as we will shortly see.

The following examples illustrate how we arrive at a recurrence equation and how we go about solving it.

Example 2.15. Bubble sort

We are given n keys and we want to arrange them from smallest to largest. A simple though not quite efficient algorithm to do this is *bubble sort*. The algorithm consists of $n - 1$ ‘passes’ in which at the i th pass the i th smallest key migrates upward to the i th position. The figure below depicts bubble sort in action. The dashed lines indicate keys that are already sorted (from smallest to largest); the solid lines indicate keys that are yet to be sorted. The ‘ $\underline{\quad}$ ’) indicates the comparisons performed; if the key below is smaller than the key above, the two are swapped.

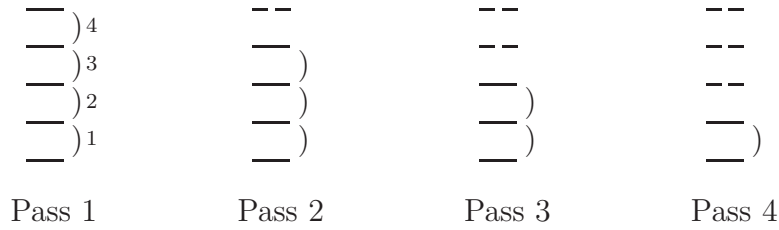


Figure 2.1 Bubble sort for $n = 5$ keys

Denote by $C(n)$ the number of comparisons performed by bubble sort to sort n keys. Then we have

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ C(n-1) + n - 1 & \text{if } n \geq 2 \end{cases} \quad (2.35)$$

This recurrence equation follows from the following observations:

- (a) There is no comparison to be performed if there is only one key.
- (b) It takes $n - 1$ comparisons (with swaps as needed) for the smallest key to migrate upward to the top position. This done, what we have is simply a smaller instance of the same sorting problem, this time on $n - 1$ keys. It takes bubble sort $C(n - 1)$ comparisons to sort $n - 1$ keys.

To solve Eq.(2.35), we substitute the RHS repeatedly until we get to $C(1)$. For instance, with $n = 6$ we have

$$\begin{aligned}
 C(6) &= C(5) + 5 \\
 &= C(4) + 4 + 5 \\
 &= C(3) + 3 + 4 + 5 \\
 &= C(2) + 2 + 3 + 4 + 5 \\
 &= C(1) + 1 + 2 + 3 + 4 + 5 \\
 &= 0 + 1 + 2 + 3 + 4 + 5 \\
 &= \sum_{i=1}^5 i = \frac{5 \times 6}{2} = 15 \text{ comparisons}
 \end{aligned}$$

We find that $C(6)$ is an arithmetic series whose sum is given by Eq.(2.27). In general, the number of comparisons performed by bubble sort to sort n keys is

$$C(n) = \sum_{i=1}^{n-1} = \frac{(n-1)n}{2} \quad (2.36)$$

For instance, it takes bubble sort 499,500 comparisons to sort 1000 keys. If the keys are initially reverse sorted, this is also the number of swaps performed.

Example 2.16. Towers of Hanoi problem

In the Towers of Hanoi problem (THP), we are given three pegs and a stack of n disks of different sizes where a smaller disk is on top of a larger disk. Initially all the disks are in some ‘source’ peg S, as shown in Figure 2.2. Our task is to transfer the disks from the source peg to some ‘target’ peg T, using an ‘intermediate’ peg I as needed, and subject to the following rules:

- (a) Only one disk may be moved at a time.
- (b) A larger disk may not be placed on top of a smaller disk.

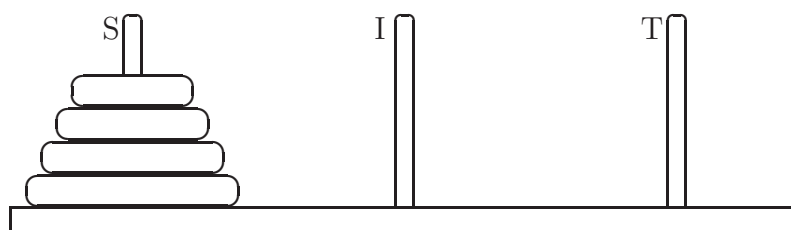


Figure 2.2 Initial configuration for Towers of Hanoi problem

The key to solving the THP lies in recognizing that when the largest disk is moved from the source peg to the target peg, the rest of the disks must be in the intermediate peg as shown in Figure 2.3. This observation leads us to the following algorithm:

Algorithm THP

If $n = 1$, then move the disk from the source peg to the target peg; otherwise

1. Move the topmost $n - 1$ disks from the source peg to the intermediate peg.
2. Move the largest disk from the source peg to the target peg.
3. Move the $n - 1$ disks from the intermediate peg to the target peg.

Of course we cannot move $n - 1$ disks all at one time in steps 1 and 3; however, we recognize these steps as simply smaller instances of the original problem. Hence we carry out steps 1 and 3 by applying algorithm THP once again. Thus the problem of moving n disks reduces to two smaller problems of moving $n - 1$ disks, where each of these smaller problems reduces to two yet smaller problems of moving one disk less, and so on, until

finally the problem becomes that of moving just one disk from the source to the target peg.

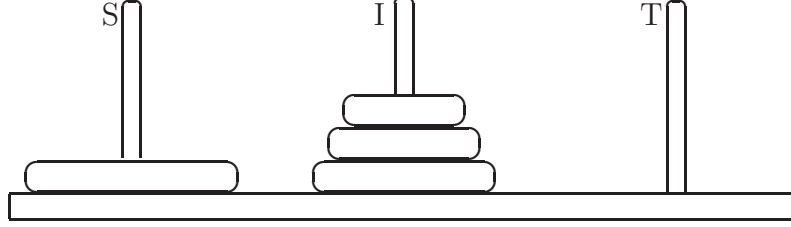


Figure 2.3 Configuration just before largest disk is moved to target peg

Let $M(n)$ denote the number of moves required to solve the Towers of Hanoi problem with n disks. From the above considerations we find that algorithm THP immediately translates to the following recurrence equation:

$$M(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \times M(n-1) + 1 & \text{if } n \geq 2 \end{cases} \quad (2.37)$$

To solve Eq.(2.37), we substitute the RHS repeatedly until we arrive at $M(1)$ and then we multiply from the inside out. For instance, with $n = 5$ we have

$$\begin{aligned} M(5) &= 2 \times M(4) + 1 \\ &= 2 \times [2 \times M(3) + 1] + 1 \\ &= 2 \times [2 \times [2 \times M(2) + 1] + 1] + 1 \\ &= 2 \times [2 \times [2 \times [2 \times M(1) + 1] + 1] + 1] + 1 \\ &= 2 \times [2 \times [2 \times [2 \times 1 + 1] + 1] + 1] + 1 \\ &= 2 \times [2 \times [2 \times [2^2 + 2 + 1] + 1] + 1] + 1 \\ &= 2 \times [2^3 + 2^2 + 2 + 1] + 1 \\ &= 2^4 + 2^3 + 2^2 + 2 + 1 \\ &= \sum_{i=0}^4 2^i = 2^5 - 1 = 31 \text{ moves} \end{aligned}$$

We find that $M(5)$ is an exponential series whose sum is given by Eq.(2.29). In general, for the THP with n disks we have

$$M(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \quad (2.38)$$

Standish, in STANDISH[1994], p. 86, tells of a rumor that somewhere in Asia there is a group of monks who is hard at work transferring 64 golden disks following the (sacred?) rules stated above. The monks work 24 hours a day, moving one disk per second, and never make a wrong move. Once their work is done it is believed that the next *Maha Pralaya* will begin in which the universe will dissolve and revert to its unmanifested state.

How soon will this universal cataclysm come to pass? There are $60 \times 60 \times 24 \times 365\frac{1}{4} = 31,557,600$ seconds per year, therefore the next *Maha Pralaya* will commence in $\frac{2^{64}-1}{31,557,600} = 584,542,046,091$ years (minus a few thousand, assuming that the monks started a few thousand years ago).

In Examples 2.15 and 2.16 above, we solved the recurrence equations for bubble sort and the THP for a *particular* value of n by repeated substitutions. In both cases the results yielded clearly recognizable patterns, from which we drew the solution for *any* value of n . That Eq.(2.36) and Eq.(2.38) are the solutions to the recurrences (2.35) and (2.37), respectively can be proved formally by using mathematical induction. This, and other methods of proof, are discussed in the next section.

2.7 Methods of proof

2.7.1 Proof by mathematical induction

Suppose that $S(n)$ is some statement about the integer n and we want to prove that $S(n)$ is true for *all* positive integers n . A proof by induction on n proceeds as follows:

1. (*Basis step*): We prove that $S(n)$ is true for some starting value (or values) of n , say n_0 (or n_0, n_0+1, n_0+2, \dots)
2. (*Inductive step*): We assume that $S(n)$ is true for $n_0, n_0+1, n_0+2, \dots, n$ (this is called the *induction hypothesis*); then we prove that $S(n+1)$ is true.

Consider the usual case where $n_0 = 1$. Essentially, we reason out as follows:

$S(1)$ is true (the basis step)

Since $S(1)$ is true, then $S(2)$ is true	}	since $S(n+1)$ is true whenever $S(n)$ is true (the inductive step)
Since $S(2)$ is true, then $S(3)$ is true		
Since $S(3)$ is true, then $S(4)$ is true		
\vdots		
Hence $S(n)$ is true for all n		

Example 2.17. Consider the following sums of consecutive positive odd integers:

$$\begin{array}{rcl}
 1 & = & 1 = 1^2 \\
 1 + 3 & = & 4 = 2^2 \\
 1 + 3 + 5 & = & 9 = 3^2 \\
 1 + 3 + 5 + 7 & = & 16 = 4^2 \\
 1 + 3 + 5 + 7 + 9 & = & 25 = 5^2
 \end{array}$$

From these five observations we make the following conjecture:

$$S(n) : \sum_{k=1}^n (2k-1) = 1 + 3 + 5 + \dots + (2n-1) = n^2 \quad (2.39)$$

We prove Eq.(2.39) as follows:

Basis step: Take $n = 1$.

$$\sum_{k=1}^1 (2k-1) = 1 = 1^2 = 1 \quad (\text{Hence } S(1) \text{ is true.})$$

Inductive step: Assume that $S(1), S(2), \dots, S(n)$ are true; we now prove that $S(n+1)$ is true.

$$\begin{aligned} S(n+1) : \sum_{k=1}^{n+1} (2k-1) &= \underbrace{1+3+5+\dots+(2n-1)}_{n^2} + \underbrace{[2(n+1)-1]}_{2n+1} \\ &= (n+1)^2 \quad \text{Q.E.D.} \end{aligned}$$

Example 2.18. Consider this time the following sums:

$$\begin{aligned} 1^3 &= 1 \\ 2^3 &= 3 + 5 \\ 3^3 &= 7 + 9 + 11 \\ 4^3 &= 13 + 15 + 17 + 19 \\ 5^3 &= 21 + 23 + 25 + 27 + 29 \end{aligned}$$

From these five observations we make the following conjecture:

$$\begin{aligned} S(n) : \sum_{k=1}^n k^3 &= 1^3 + 2^3 + 3^3 + \dots + n^3 \\ &= 1 + (3+5) + (7+9+11) + (13+15+17+19) + \dots \\ &= \text{sum of the first } (1+2+3+\dots+n) \text{ positive odd integers} \\ &= (1+2+3+\dots+n)^2 \quad \text{from Eq.(2.39)} \\ &= \left[\frac{n(n+1)}{2} \right]^2 \quad \text{from Eq.(2.27)} \quad (2.40) \end{aligned}$$

We prove Eq.(2.40) as follows:

Basis step: Take $n = 1$

$$\sum_{k=1}^1 k^3 = 1^3 = \left[\frac{1(1+1)}{2} \right]^2 = 1 \quad (\text{Hence } S(1) \text{ is true.})$$

Inductive step: Assume that $S(1), S(2), \dots, S(n)$ are true; we now prove that $S(n+1)$ is true.

$$\begin{aligned}
S(n+1) : \sum_{k=1}^{n+1} k^3 &= \underbrace{1^3 + 2^3 + 3^3 + \cdots + n^3}_{\left[\frac{n(n+1)}{2}\right]^2} + (n+1)^3 \\
&= \frac{n^4 + 2n^3 + n^2}{4} + n^3 + 3n^2 + 3n + 1 \\
&= \frac{n^4 + 6n^3 + 13n^2 + 12n + 4}{4} \\
&= \frac{(n^2 + 3n + 2)^2}{4} \\
&= \left[\frac{(n+1)(n+2)}{2} \right]^2 \quad \text{Q.E.D.}
\end{aligned}$$

Example 2.19. This example illustrates the case where the basis step involves more than one value of n . Consider the statement:

$$S(n) : F_n \leq \left(\frac{5}{3}\right)^n \quad n \geq 0 \quad (2.41)$$

We prove Eq.(2.41) as follows:

Basis step: Take $n = 0$ and $n = 1$.

$$F_0 = 0 \leq \left(\frac{5}{3}\right)^0$$

$$F_1 = 1 \leq \left(\frac{5}{3}\right)^1$$

Inductive step: Assume that $S(0), S(1), \dots, S(n)$ are true; we now prove that $S(n+1)$ is true.

$$\begin{aligned}
S(n+1) : F_{n+1} &= F_{n-1} + F_n && \text{by Eq.(2.11)} \\
&\leq \left(\frac{5}{3}\right)^{n-1} + \left(\frac{5}{3}\right)^n && \text{by Eq.(2.41)} \\
&\leq \left(\frac{5}{3}\right)^{n-1} \left[\left(\frac{5}{3}\right) + 1\right] \\
&\leq \left(\frac{5}{3}\right)^{n-1} \left(\frac{8}{3}\right) \\
&\leq \left(\frac{5}{3}\right)^{n-1} \left(\frac{24}{9}\right) \\
&\leq \left(\frac{5}{3}\right)^{n-1} \left(\frac{25}{9}\right) \\
&\leq \left(\frac{5}{3}\right)^{n-1} \left(\frac{5}{3}\right)^2 \\
&\leq \left(\frac{5}{3}\right)^{n+1} && \text{Q.E.D.}
\end{aligned}$$

2.7.2 Proof by contradiction (*reductio ad absurdum*)

Suppose we want to prove that some statement or theorem T is true. A proof by contradiction proceeds as follows:

1. We assume that T is false.
2. We show that this assumption leads to a contradiction, i.e., it implies that some established fact or known property is false.
3. Since our assumption that T is false is false, we conclude that T is true.

Example 2.20. A classic example of this method of proof is found in Book IX of Euclid's *Elements* and has to do with primes. Recall that all positive integers $n > 1$ are either *prime* (those with exactly two positive divisors, viz., 1 and n itself) or *composite* (those with more than two positive divisors). Consider now the following statement:

T (Euclid): There are infinitely many primes.

We prove this statement to be true as follows:

1. Assume that T is false in that there is some largest prime, say P_k .
2. Consider the number

$$N = P_1 \times P_2 \times P_3 \times \dots \times P_k + 1$$

where P_1, P_2, \dots, P_k are all the primes in order. Clearly, $N > P_k$ and since P_k by our assumption in 1 is the largest prime, then N is not prime. However, none of the P_1, P_2, \dots, P_k evenly divides N because there is always a remainder of 1, which means N is prime (divisible only by 1 and N itself). This is a contradiction; N cannot be prime and be non-prime as well.

3. Since the assumption that P_k is the largest prime is false (leads to a contradiction), we conclude that T is true.

2.7.3 Proof by counterexample

To prove that some statement T is false, it suffices to give one example which exhibits the falsity of T .

Example 2.21. Consider the statement

T : The number $p(n) = n(n-1) + 41$ is prime for all $n \geq 1$.

Indeed $p(n)$ is prime for $n = 1, 2, \dots, 40$ (verify); however, $p(41) = 41(40) + 41 = 41(40+1) = 41 \times 41 = 1681$ is not prime. Hence, T is false.

This example also illustrates the danger of hastily jumping to a conclusion.

Summary

- Floors and ceilings, polynomials, exponentials, logarithms and factorials are some of the familiar mathematical entities that are useful in describing the properties of data structures and the behavior of algorithms.
- Two simple, albeit extremely useful, rules in solving elementary counting problems are the rule of sum and the rule of product.
- Sums or sums of sums arise naturally in describing loops or nested loops in iterative algorithms. Formulas for the sum of arithmetic and exponential series are useful in solving summation problems.
- Recurrences arise naturally in describing recursive algorithms. Simple recurrences can be solved by converting them into summations through repeated substitution.
- Three methods for proving the truth or validity of certain formulas or propositions or theorems are proof by mathematical induction, proof by contradiction and proof by counterexample.

Exercises

1. Prove the following identities.

$$(a) m \bmod n < \frac{m}{2}, \quad 0 < n < m$$

$$(d) \log_c x = \frac{\log_b x}{\log_b c}$$

$$(b) z(x \bmod y) = zx \bmod zy$$

$$(e) x^{\log_b y} = y^{\log_b x}$$

$$(c) \lceil -x \rceil = -\lfloor x \rfloor$$

2. Calculate $\lfloor x \rfloor + x \bmod 1$ given that: (a) $x = 3.1416$ (b) $x = -3.1416$
3. Find p given that: (a) $2^{64} = 10^p$ (b) $2^p = 10^{100}$
4. Find the sequence generated by the function

$$[h(K) - i \cdot p(K)] \bmod m \quad 0 \leq i \leq m - 1$$

where $h(K) = K \bmod m$ and $p(K) = 1 + \left\lfloor \frac{K}{m} \right\rfloor \bmod (m - 2)$, given $K = 1803$ and $m = 13$. Describe the sequence.

5. The powerset $\mathcal{P}(S)$ of an n -set S , $n \geq 0$, is the set of all subsets of S . For example, given the set $S = \{a, b, c, d\}$, then $\mathcal{P}(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}, \{a, b, c, d\}\}$. Using the rule of product, show that $|\mathcal{P}(S)| = 2^n$.
6. You are taking a CS exam and you must answer 9 out of 12 problems. In how many ways can you select the 9 problems if:
 - (a) you can choose *any* 9 out of the 12 problems
 - (b) you must choose 4 from the first 6 and 5 from the last 6 problems
 - (c) you must choose *at least* 4 from the first 6 problems

7. Determine whether the relation R is reflexive, symmetric, antisymmetric and/or transitive.

$$(a) \ x R y \quad \text{if} \quad x + y \text{ is even} \quad x, y \in \mathbb{Z} \text{ (the set of integers)}$$

$$(b) \ x R y \quad \text{if} \quad x + y \text{ is odd} \quad x, y \in \mathbb{Z}$$

$$(c) \ x R y \quad \text{if} \quad x - y \text{ is even} \quad x, y \in \mathbb{Z}$$

$$(d) \ x R y \quad \text{if} \quad x - y \text{ is odd} \quad x, y \in \mathbb{Z}$$

$$(e) \ x R y \quad \text{if} \quad \gcd(x, y) = 1 \quad x, y \in \mathbb{Z}^+ \text{ (the set of positive integers)}$$

8. Which of the relations in Item 7 is a partial order? an equivalence relation?

9. Find a closed form expression for $T(n)$.

$$(a) \ T(n) = \sum_{i=1}^n \sum_{j=1}^{3i-1} 1$$

$$(b) \ T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=1}^j 1$$

$$(c) \ T(n) = \sum_{i=1}^n \sum_{j=0}^{n-i} \sum_{k=j+1}^{j+i} 1$$

10. Solve the following recurrences.

$$(a) \ T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n \geq 2 \end{cases}$$

$$(b) \ T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \frac{1}{2} T(n-1) + 1 & \text{if } n \geq 2 \end{cases}$$

$$(c) \ T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(\frac{n}{2}) + 1 & \text{if } n = 2^m \geq 2 \end{cases}$$

11. Prove by mathematical induction.

$$(a) \ \left\lfloor \frac{n}{2} \right\rfloor = \begin{cases} n/2 & n > 0, \text{ } n \text{ even} \\ (n-1)/2 & n > 0, \text{ } n \text{ odd} \end{cases}$$

$$(b) \ \left\lceil \frac{n}{2} \right\rceil = \begin{cases} n/2 & n > 0, \text{ } n \text{ even} \\ (n+1)/2 & n > 0, \text{ } n \text{ odd} \end{cases}$$

$$(c) \ \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad n \geq 1$$

$$(d) \ \sum_{i=1}^n \frac{i}{2^i} = 2 - \frac{n+2}{2^n} \quad n \geq 1$$

$$(e) \sum_{i=1}^n i \cdot i! = (n+1)! - 1 \quad n \geq 1$$

$$(f) \sum_{i=1}^n i \cdot 2^{i-1} = (n-1)2^n + 1 \quad n \geq 1$$

$$(g) \sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1}, \quad x \neq 1 \quad n \geq 0$$

$$(h) 2^n < n! \quad n \geq 4$$

$$(i) \sum_{i=1}^n \frac{1}{i^2} < 2 - \frac{1}{n} \quad n \geq 2$$

12. Find a formula for the sum

$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \cdots + \frac{1}{n \cdot (n+1)}$$

and prove that it is correct by induction on n .

13. From the following equations deduce a general rule and prove it by induction.

$$1 = 1$$

$$2 + 3 + 4 = 1 + 8$$

$$5 + 6 + 7 + 8 + 9 = 8 + 27$$

$$10 + 11 + 12 + 13 + 14 + 15 + 16 = 27 + 64$$

14. Give a simple geometric (or pictorial) representation of Eq.(2.39).

15. Verify that the numbers $p(n) = n(n-1) + 41$ are in fact prime for $1 \leq n \leq 40$.

16. Let $p(n)$ be the number of ways of writing n as a sum of positive integers, disregarding order. For instance:

$$p(1) = 1 \quad \text{since } 1 = 1$$

$$p(2) = 2 \quad \text{since } 2 = 1 + 1 = 2$$

$$p(3) = 3 \quad \text{since } 3 = 1 + 1 + 1 = 1 + 2 = 3$$

$$p(4) = 5 \quad \text{since } 4 = 1 + 1 + 1 + 1 = 1 + 1 + 2 = 1 + 3 = 2 + 2 = 4$$

Prove or disprove: $p(n)$ is prime for all positive n .

Bibliographic Notes

Part VIII (Appendix: Mathematical Background) of CORMEN[2001] and Section 1.2 (Mathematical Preliminaries) of KNUTH1[1997] give an extensive and very thorough treatment of the topics covered in this session. COHEN[1978] and GRIMALDI[1994] are additional references for the material on combinatorial mathematics.

SESSION 3

Algorithms

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Explain the relative merits between using a formal programming language and pseudocode in communicating algorithms.
2. Explain the notions of time and space complexities of an algorithm.
3. Implement algorithms as EASY procedures.
4. Analyze simple algorithms and state their time complexity using the O-notation.

READINGS HOROWITZ[1976], pp. 8–15; CORMEN[2001], pp. 5–27, 41–46; STANDISH[1994], pp. 205–230; KNUTH1[1997], pp. 107–111; KNUTH3[1998], pp. 80–102.

DISCUSSION

In Session 1, we defined the concept of an algorithm and we elaborated on certain requisites that it must possess, foremost among which is ‘finiteness’. In this session, we will continue our study of algorithms as we address two recurring issues pertaining to algorithms, namely:

1. How do we communicate an algorithm?

and, more importantly,

2. How do we analyze an algorithm?

The medium that we use in communicating an algorithm depends on who we are communicating the algorithm to. Clearly, we must use a formal programming language in communicating an algorithm to a computer, simply because this is the language a computer ‘understands’. For reasons that we will explain shortly, we will use pseudocode, instead of a formal programming language, in communicating an algorithm to another person (or to one’s self). Thus we will use pseudocode when we discuss algorithms in

class, and we will use whatever programming language suits us when we implement such algorithms on a machine. The first part of this session will be devoted to a description of the pseudocode that we will use in this book.

In the second part of this session we will look into the meaning of the phrase ‘*analyze an algorithm*’ and into the notion of the *time complexity* and *space complexity* of a running algorithm. As our vehicle for exposition we will analyze a well-known sorting algorithm called *insertion sort*. Finally, we will discuss three *asymptotic notations* for characterizing the performance of running algorithms and measuring their comparative efficiencies.

3.1 Communicating an algorithm

There is no common notation used in communicating an algorithm among the references listed in the bibliography. Some authors use a formal programming language while others choose a less formal notation. For instance, TENENBAUM[1986], KORFHAGE[1987] and STANDISH[1994] use Pascal, SEDGEWICK[1990] and WEISS[1993] use C and GOODRICH[1998] uses JAVA. KNUTH1[1997] thru KNUTH3[1998] specify algorithms initially in plain English and then ‘implements’ these in an assembly language-like notation called ‘MIX’ for purposes of detailed analysis. HOROWITZ[1976] and CORMEN[2001] use pseudocode, a notation which utilizes familiar programming constructs without the usual detailed rules imposed by a formal programming language. NIEVERGELT[1993] presents a strong case for the use of informal notation (p. 43): *‘In addressing human readers, we believe an open-ended somewhat informal notation is preferable to the straightjacket of any one programming language. The latter becomes necessary if and when we execute a program, but during the incubation period when our understanding slowly grows toward a firm grasp of an idea, supporting human intuition is much more important than formality. Thus we describe data structures and algorithms with the help of figures, words, and programs as we see fit in any particular instance.’*

In this book, we will specify most of the algorithms we will study through a three-step process:

1. We state the algorithm in plain English (as with Algorithm E and Algorithm G in Session 1).
2. We give an example of the algorithm at work by performing the algorithm ‘by hand’ (as with Algorithm E and Algorithm G in Session 1).
3. We ‘implement’ the algorithm in pseudocode.

We choose pseudocode over a formal programming language as our vehicle for implementing an algorithm for the following reasons:

1. A formal programming language is designed to communicate an algorithm to a computer (so it can execute it); we want a medium that will communicate an algorithm to the student (so that he/she will understand it better)
2. We want a notation that is not encumbered by too many low-level details which can distract us from the high-level ideas that we want to communicate

3. A high-level analysis of an algorithm is facilitated by its concise specification in pseudocode.
4. Transcribing the algorithms from our pseudocode to a formal programming language is an excellent learning experience for the student.

The notation we will use is called EASY. It is very similar to the pseudocode called SPARKS in HOROWITZ[1976]. (EASY was used in the first course ever on Data Structures conducted in U.P. in the mid 70's by Dr. Oscar Ibarra, a visiting professor from the University of Minnesota.) The following are the statements and salient features of EASY.

3.1.1 The EASY assignment statement

The general form of the EASY assignment statement is

$$variable \leftarrow expression$$

where ' \leftarrow ' is read as 'gets' and *expression* may be of type arithmetic, logical or character. To construct these expressions, the following types of operators are provided: (a) arithmetic operators: $+$, $-$, $/$, \times , $^$; (b) logical operators: **and**, **or**, **not**; (c) relational operators: $<$, \leq , $=$, \neq , $>$, \geq , etc.; (d) miscellaneous mathematical notations: $\lfloor \rfloor$, $\lceil \rceil$, mod, \log_2 , etc. The following are valid EASY assignment statements.

$$\begin{aligned} middle &\leftarrow \lfloor (lower + upper)/2 \rfloor \\ \alpha &\leftarrow LINK(\beta) \\ cond &\leftarrow \text{IsEmptyQueue}(Q) \\ S &\leftarrow 'S' \end{aligned}$$

3.1.2 The EASY unconditional transfer statements

- (a) the **go to** statement

$$\text{go to } label$$

The **go to** statement causes a transfer to the statement labeled *label*. In general, we avoid the use of the **go to** statement.

- (b) the **exit** statement

$$\text{exit}$$

The **exit** statement causes transfer to the statement immediately following the loop which contains the **exit** statement.

3.1.3 The EASY conditional transfer statements(a) the **if** statement

```

if cond then  $S_1$ 
      else  $S_2$ 

```

where S_1 and S_2 consist of one or more EASY statements. In the latter case, the group of statements is enclosed in square brackets. The **else** clause is optional.

(b) the **case** statement

```

case

  : cond1 :  $S_1$ 
  : cond2 :  $S_2$ 
    ⋮
  : condn :  $S_n$ 
  : else :  $S_{n+1}$ 

endcase

```

The **case** is used where we would otherwise use a sequence of **if** statements of the form

```

if cond1 then  $S_1$ 
      else if cond2 then  $S_2$ 
      else . . .

```

3.1.4 The EASY iteration statements(a) the **while** statement

```

while cond do

   $S$ 

endwhile

```

The statement or group of statements S is repeatedly executed for as long as *cond* is true. Some statement in S must either change *cond* to false (normal exit) or cause transfer to a statement outside the loop (premature exit). If *cond* is initially false, S is not executed at all.

(b) the **repeat-until** statement

```

repeat

   $S$ 

until cond

```

The statement or group of statements S is repeatedly executed for as long as $cond$ is false. Some statement in S must either change $cond$ to true (normal exit) or cause transfer to a statement outside the loop (premature exit). S is executed at least once regardless of the initial value of $cond$.

(c) the **loop-forever** statement

```

loop
     $S$ 
forever

```

Some statement in S must cause an exit from this otherwise infinite loop.

(d) the **for** statement

```

for  $var \leftarrow start$  to  $finish$  by  $step$  do
     $S$ 
endfor

```

var is a variable while $start$, $finish$ and $step$ are arithmetic expressions. var is initially set to the value of $start$ and is incremented [decremented] by the value of $step$ after every execution of S . Looping continues until var exceeds [becomes less than] the value of $finish$ (normal exit) or some statement in S causes transfer to a statement outside the **for**-loop (premature exit). $step$ is assumed to be 1 by default; it may be negative.

3.1.5 The EASY input/output statements

```

input  $list$ 
output  $list$ 

```

where $list$ is a sequence of one or more variable names or quoted strings (for output).

3.1.6 The EASY declaration statements

(a) the **array** statement

```

array  $name(l_1:u_1, l_2:u_2, \dots, l_i:u_i, \dots, l_n:u_n), \dots$ 

```

where $name$ is the name given to the array and l_i and u_i are the lower and upper bounds for the i th dimension. By default, l_i is 1. More than one array may be declared in a single **array** statement.

(b) the **node** statement

$$\mathbf{node}(f_1, f_2, \dots, f_i, \dots, f_n)$$

where f_i is a field name. The **node** statement specifies the node structure of the nodes used in a procedure.

3.1.7 The EASY control statements

(a) the **call** statement

$$\mathbf{call} \textit{procname}(ap_1, ap_2, \dots, ap_i, \dots, ap_n)$$

The **call** statement transfers control to the procedure named *procname*.

(b) the **return** statement

$$\mathbf{return} \quad \text{or} \quad \mathbf{return}(\textit{expression})$$

The **return** statement transfers control back to the procedure which called the procedure containing the **return** statement.

(c) the **stop** statement

$$\mathbf{stop}$$

The **stop** statement terminates execution of the procedure and returns control to the operating system.

(d) the **end** statement

$$\mathbf{end}$$

The **end** statement marks the physical end of a procedure. In the absence of a **return** statement, reaching the **end** statement implies a return.

3.1.8 EASY program stucture

An EASY program is a collection of one or more EASY procedures. An EASY procedure has the form

$$\mathbf{procedure} \textit{procname}(fp_1, fp_2, \dots, fp_i, \dots, fp_n)$$

$$S$$

$$\mathbf{end} \textit{procname}$$

where *procname* is the name given to the procedure and fp_1, fp_2, \dots are formal parameters. A procedure is invoked using the **call** statement

call *procname*($ap_1, ap_2, \dots, ap_i, \dots, ap_n$)

where *procname* is the name of the procedure and ap_1, ap_2, \dots are the actual parameters, which correspond on a one-to-one basis by type and size, with the formal parameters fp_1, fp_2, \dots in the called procedure. A procedure can be implemented as a function by using the statement

return (*expression*)

where the value of *expression* is delivered as the value of the function procedure. A function is called implicitly, for instance, by using an assignment statement, as in

$var \leftarrow procname(ap_1, ap_2, \dots, ap_i, \dots, ap_n)$

Association of actual to formal parameters is via *call by reference*. This means that when a procedure is called, the address of each actual parameter in the calling procedure is passed to the called procedure, thereby giving access to the called procedure to the memory locations occupied by the actual parameters. Whatever the called procedure does on these parameters is ‘seen’ by the calling procedure.

Variables which are generally accessed by all procedures in an EASY program will, for convenience, be considered *global*. Which variables these are will usually be obvious from context.

EASY comments are indicated by the symbol ‘▷’, as in

▷ *Calculate allocation factors a and b.*

Other features of EASY, such as the construct called an EASY *data structure*, will be discussed at the appropriate time and place in these Notes.

3.1.9 Sample EASY procedures

(a) **Binary search** — Given an array A of size n whose elements are sorted in non-decreasing order, and a query x , find i such that $x = A(i)$.

```

procedure BINARY_SEARCH( $A, n, x$ )
array  $A(1:n)$ 
 $lower \leftarrow 1; upper \leftarrow n$ 
while  $lower \leq upper$  do
     $middle \leftarrow \lfloor (lower + upper)/2 \rfloor$ 
    case
         $:x = A(middle)$ : return( $middle$ )      ▷ successful search
         $:x > A(middle)$ :  $lower \leftarrow middle + 1$ 
         $:x < A(middle)$ :  $upper \leftarrow middle - 1$ 
    endcase
endwhile
return(0)      ▷ unsuccessful search
end BINARY_SEARCH

```

Procedure 3.1 Binary search

The binary search algorithm is familiar to most CS students. It is implemented here as an EASY function which returns the index of the matching element in A , if such an element is found, or 0 if none. A sample calling program is:

```

procedure MAIN
array  $T(1:10000)$ 
   $\vdots$ 
input  $m, q, T(1:m)$ 
   $\vdots$ 
 $index \leftarrow \text{BINARY\_SEARCH}(T, m, q)$ 
if  $index = 0$  then ...
       $\vdots$ 
      else ...
       $\vdots$ 
end MAIN

```

Binary search is an extremely important algorithm for searching an ordered array. We will study the algorithm in depth in Session 13. For now, its implementation as Procedure 3.1 is intended merely to illustrate some of the constructs of EASY.

(b) **Towers of Hanoi problem** — We described this problem in Session 2 where we also gave a recursive algorithm to solve it. The algorithm is again shown here for quick reference.

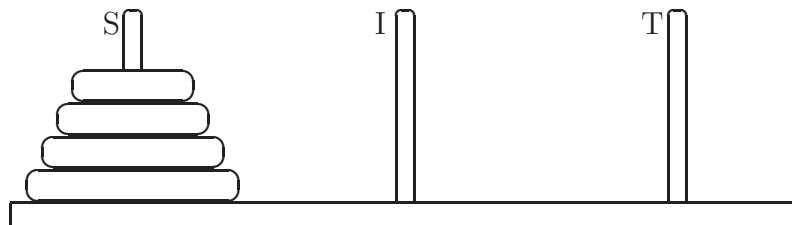


Figure 3.1 The Towers of Hanoi problem revisited

Algorithm THP

If $n = 1$ then move the disk from the source peg to the target peg; otherwise

1. Move the topmost $n - 1$ disks from the source peg to the intermediate peg.
2. Move the largest disk from the source peg to the target peg.
3. Move the $n - 1$ disks from the intermediate peg to the target peg.

The following EASY procedure implements Algorithm THP. The parameter n is assumed to be a positive integer. The parameters S, T and I are character variables with values 'S', 'T' and 'I', respectively.


```

procedure THP( $n, S, T, I$ )
if  $n = 1$  then output 'Move disk from peg'  $S$  'to peg'  $T$ 
      else [ call THP( $n - 1, S, I, T$ )
              output 'Move disk from peg'  $S$  'to peg'  $T$ 
              call THP( $n - 1, I, T, S$ ) ]
end THP

```

Procedure 3.2 Solution to the Towers of Hanoi problem

Notice that procedure THP is simply a straightforward transcription of Algorithm THP into EASY. This is one desirable property of recursive algorithms; they translate *directly* into recursive programs, statement by statement. This is because most of the 'housekeeping' is handled by the runtime system rather than the program itself.

A sample output of procedure THP, generated when invoked with the statement

```

:
 $n \leftarrow 4$ ;  $S \leftarrow 'S'$ ;  $T \leftarrow 'T'$ ;  $I \leftarrow 'I'$ 
call THP( $n, S, T, I$ )
:

```

is shown below.

```

Move disk from peg S to peg I
Move disk from peg S to peg T
Move disk from peg I to peg T
Move disk from peg S to peg I
Move disk from peg T to peg S
Move disk from peg T to peg I
Move disk from peg S to peg I
Move disk from peg S to peg T
Move disk from peg I to peg T
Move disk from peg I to peg S
Move disk from peg T to peg S
Move disk from peg I to peg T
Move disk from peg S to peg I
Move disk from peg S to peg T
Move disk from peg I to peg T

```

You will recall from Session 2 that it takes $2^n - 1$ moves to solve the Towers of Hanoi problem for n disks. Thus we get 15 lines of output for $n = 4$, one line for each move. Despite its brevity, procedure THP can be quite prodigious in producing lines of output; for $n = 20$, it generates 1,048,575 lines of the form shown above.

(c) **List inversion** — Given a singly-linked linear list pointed to by l , invert the list *in-place*. Figure 3.2 depicts the list l before and after the inversion.

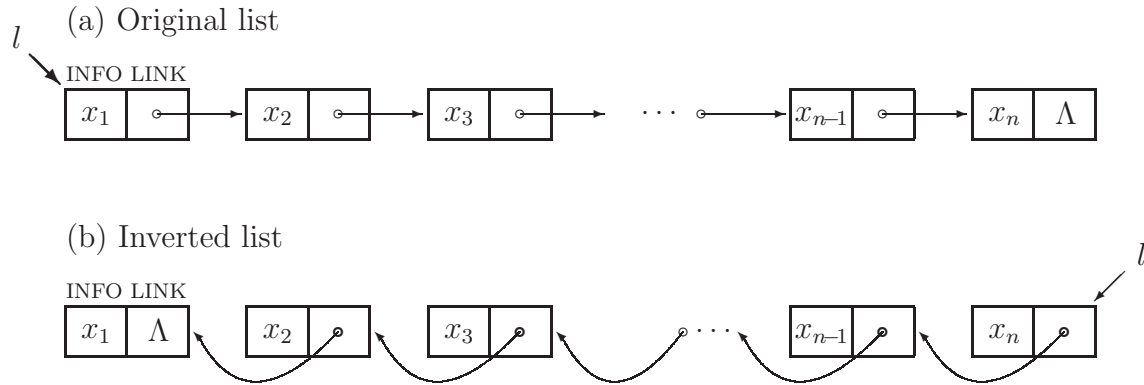


Figure 3.2 Inverting a list in-place

Procedure INVERT performs the list inversion in-place, utilizing only three auxiliary pointer variables α , β and γ . Figure 3.3 shows the configuration of the list and the position of the pointers immediately after the second loop of the **while** statement is executed.

```

procedure INVERT( $l$ )
  node(INFO, LINK)
   $\beta \leftarrow \Lambda$ ;  $\alpha \leftarrow l$ 
  while  $\alpha \neq \Lambda$  do
     $\gamma \leftarrow \beta$ 
     $\beta \leftarrow \alpha$ 
     $\alpha \leftarrow \text{LINK}(\beta)$ 
     $\text{LINK}(\beta) \leftarrow \gamma$ 
  endwhile
   $l \leftarrow \beta$ 
end INVERT

```

Procedure 3.3 Inverting a list in place

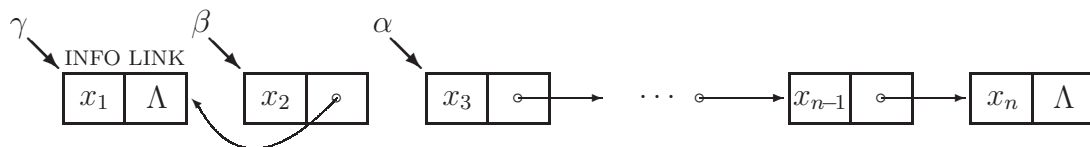


Figure 3.3 Configuration of list l after two loops of the **while** statement

This example (see KNUTH1[1997], pp. 269 and 546) illustrates the use of the link design in EASY. The inversion algorithm is quite elegant, and it is best understood by ‘walking through’ the procedure using a sample linked list of, say, five nodes. [Do that now.] This technique of using three pointer variables to effect the link inversion in-place will be utilized in subsequent algorithms.

3.2 Analyzing an algorithm

When using an algorithm, we can usually identify a parameter which characterizes the size of the input, say n , to the algorithm, or equivalently, the size of the problem to be solved. For instance, if we are using a sorting algorithm, n would be the number of data elements to be sorted. If we are using an algorithm on trees, n would be the number of nodes in the tree; in an algorithm on graphs, n would be the number of vertices in the graph.

An algorithm requires both *time* and *space* for it to accomplish its task. In general, the amount of time and space it uses depends on the size of the input to the algorithm. We *analyze an algorithm* by determining the amount of time and space that the algorithm requires for a given size of input. More to the point, we determine the *rate of growth*, or *order of growth*, of the time and space utilized by the algorithm *as the size of the input increases*. This rate of growth, expressed as a function of the size of the input, and evaluated in the limit, is called the ***asymptotic time complexity*** and ***asymptotic space complexity*** of the algorithm. To analyze an algorithm, therefore, is to determine its asymptotic time and space complexities.

Ordinarily, we are more interested in the time complexity, or equivalently the *running time*, of an algorithm rather than in its space complexity, or its *running space*. It is obvious that the actual running time of an algorithm depends on other factors apart from the size of the input, for instance, the compiler, operating system and machine used to implement the algorithm. However, we want our analysis to be independent from such transient or accidental factors and to depend solely on something more fundamental, viz., the *size of the input*.

To this end, we will use as our model of computation a generic, one-processor random-access machine, or **RAM**. We assume that our RAM executes its instructions, presented to it in our pseudocode (EASY), one after another (i.e., there are no concurrent operations), and that it is able to access any cell in its memory in a constant amount of time. Furthermore, we assume that our RAM takes a constant amount of time to execute an instruction in our pseudocode. Let us call the time it takes our RAM to execute an EASY instruction the ***cost*** of the instruction. This cost may vary from one instruction to another, but it is constant for any given instruction. For example, assume that the cost of executing the instruction $x \leftarrow y$ is c_1 and that the cost of executing the instruction $\alpha \leftarrow LINK(\beta)$ is c_2 . The values of c_1 and c_2 are not necessarily equal; however, we assume both values to be constants. Thus the cost of executing the instruction $x \leftarrow y$ m times is simply $m \cdot c_1$.

In terms of our model of computation, *the time complexity or running time of an algorithm is the total cost incurred, expressed as a function of the size of the input, in executing the algorithm to completion.*

3.2.1 Analysis of insertion sort

Let us now apply these ideas and analyze a well-known sorting algorithm called *insertion sort*. The algorithm is presented below as an EASY procedure (ignore for the moment the two columns on the right). The input to the procedure is an array $A(1:n) = (a_1, a_2, \dots, a_n)$;

we assume that for any two elements a_i and a_j in A , one of the following relations holds: $a_i < a_j$, $a_i = a_j$ or $a_i > a_j$. The procedure rearranges the elements in-place in non-decreasing order.

```

procedure INSERTION_SORT( $A, n$ )
  ▷ Given an array  $A$  containing a sequence of  $n$  elements  $A(1), A(2), \dots, A(n)$ ,
  ▷ procedure sorts the elements in-place in non-decreasing order.
  array  $A(1 : n)$ 
1  for  $i \leftarrow 2$  to  $n$  do
2     $key \leftarrow A(i)$ 
3     $j \leftarrow i - 1$ 
4    while  $j > 0$  and  $A(j) > key$  do
5       $A(j + 1) \leftarrow A(j)$ 
6       $j \leftarrow j - 1$ 
7    endwhile
8     $A(j + 1) \leftarrow key$ 
9  endfor
end INSERTION_SORT

```

Procedure 3.4 Insertion sort

To understand the inner workings of the algorithm, we imagine the array $A(1:n)$ to be divided into two parts at any instance during the computations: a subarray $A(1:i-1)$ whose elements are already sorted and a subarray $A(i:n)$ whose elements are yet to be sorted. This is illustrated in the figure below which shows the state of the computations at the start of the i th iteration. The key idea is to insert the *current element* a_i in its proper position in the subarray $A(1:i-1)$ such that the resulting subarray $A(1:i)$ remains sorted after the insertion. We then imagine the boundary between the sorted and yet-to-be-sorted subarrays to be moved one position up.

$$\underbrace{a_1 \ a_2 \ a_3 \ a_4 \ \cdots \ a_{i-1}}_{\text{sorted subarray}} \quad \underbrace{a_i \ a_{i+1} \ a_{i+2} \ \cdots \ a_{n-1} \ a_n}_{\text{yet-to-be-sorted subarray}}$$

Figure 3.4 The array A at the start of the i th iteration of insertion sort

When inserting a_i in the subarray $A(1:i-1)$, one of three possibilities may arise:

Case 1. $a_i \geq a_{i-1}$, in which case a_i is already in its proper position

Case 2. $a_i < a_1$, in which case $a_{i-1}, a_{i-2}, \dots, a_1$ must be shifted one position up so that a_i can be placed in the position originally occupied by a_1

Case 3. $a_k \leq a_i < a_{i-1}$, $1 \leq k < i-1$, in which case elements $a_{i-1}, a_{i-2}, \dots, a_{k+1}$ must be shifted one position up so that a_i can be placed in the position originally occupied by a_{k+1} .

As a simple exercise, apply the algorithm on the input sequence ‘6 3 2 8 4 3 9 1’ and identify which of these three cases obtains at each iteration.

Let us now find the running time for procedure INSERTION_SORT given above. Each executable EASY instruction is numbered and the quantity $c_k, k = 1, 2, \dots, 7$ is the time cost of executing instruction k . The number of times each instruction is executed is indicated in the last column. For instance, the **for**-loop test is executed n times (with $i = 2, 3, \dots, n$ and finally with $i = n + 1$ at which point a normal exit from the **for**-loop occurs). For each iteration of the **for** loop, the number of times the **while**-loop test is executed, say t_i , depends on whether Case 1, 2 or 3 as described above obtains. For Case 1, $t_i = 1$ and the body of the **while** loop is not executed at all. For Case 2, $t_i = i$ and the body of the **while**-loop is executed $i - 1$ times, i.e., all of the elements in the sorted subarray $A(1:i-1)$ are moved up. For Case 3, $1 < t_i < i$, i.e., some of the elements in the sorted subarray $A(1:i-1)$ are moved up.

According to our model of computation, an instruction which costs c_k to execute contributes $c_k \cdot m$ to the total cost if it is executed m times, and that the running time of an algorithm is simply the total cost incurred in executing the algorithm to completion. Thus the running time of insertion sort, as implemented by procedure INSERTION_SORT, is simply

$$T(n) = c_1 n + (c_2 + c_3)(n - 1) + c_4 \sum_{i=2}^n t_i + (c_5 + c_6) \sum_{i=2}^m (t_i - 1) + c_7(n - 1) \quad (3.1)$$

The notation $T(n)$ on the LHS of Eq.(3.1) indicates that the running time T is a function of the input size n . However, the running time of an algorithm depends also on the *nature* of the input, in addition to its size. Thus for inputs of size n , we can categorize the running time of an algorithm as either **best-case**, **worst-case** or **average-case**, depending on *which* input of size n is given.

Best-case analysis of insertion sort

Insertion sort attains its best-case running time if the input array is already sorted. For this input, Case 1 always obtains and $t_i = 1$ for $i = 2, 3, \dots, n$. That is, for all i , no element in the sorted subarray $A(1:i-1)$ needs to be moved during the i th iteration since a_i is already where it should be. Thus Eq.(3.1) becomes

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3)(n - 1) + c_4 \sum_{i=2}^n 1 + (c_5 + c_6) \sum_{i=2}^m (1 - 1) + c_7(n - 1) \\ &= c_1 n + (c_2 + c_3)(n - 1) + c_4(n - 1) + (c_5 + c_6)(0) + c_7(n - 1) \\ &= \underbrace{(c_1 + c_2 + c_3 + c_4 + c_7)}_a n + \underbrace{(-c_2 - c_3 - c_4 - c_7)}_b \\ &= a n + b \end{aligned} \quad (3.2)$$

where a and b are constants. We see that the best-case running time of insertion sort is a **linear** function of n .

Worst-case analysis of insertion sort

Insertion sort attains its worst-case running time if the input array is reverse-sorted. For this input, Case 2 always obtains and $t_i = i$ for $i = 2, 3, \dots, n$. That is, for all i , all the elements in the sorted subarray $A(1:i-1)$ must be moved one position up so that a_i can be placed where a_1 used to be. Thus Eq.(3.1) becomes

$$\begin{aligned}
 T(n) &= c_1 n + (c_2 + c_3)(n-1) + c_4 \sum_{i=2}^n i + (c_5 + c_6) \sum_{i=2}^n (i-1) + c_7(n-1) \\
 &= c_1 n + (c_2 + c_3)(n-1) + c_4 \left[\sum_{i=1}^n i - 1 \right] + (c_5 + c_6) \sum_{i=1}^{n-1} i + c_7(n-1) \\
 &= c_1 n + (c_2 + c_3)(n-1) + c_4 \left[\frac{n(n+1)}{2} - 1 \right] + (c_5 + c_6) \left[\frac{(n-1)n}{2} \right] + c_7(n-1) \\
 &= \underbrace{\frac{1}{2}(c_4 + c_5 + c_6)}_a n^2 + \underbrace{c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7}_b n + \underbrace{(-c_2 - c_3 - c_4 - c_7)}_c \\
 &= a n^2 + b n + c
 \end{aligned} \tag{3.3}$$

where a , b and c are constants. We see that the worst-case running time of insertion sort is a **quadratic** function of n .

Average-case analysis of insertion sort

For insertion sort, a best-case input is an array that is already sorted and a worst-case input is an array that is reverse-sorted. If the input were a *random sequence* of n elements, we can assume that, *on the average*, one-half of the elements in the sorted subarray $A(1:i-1)$ are moved when a_i is inserted into the subarray, or that $t_i = i/2$ for $i = 2, 3, \dots, n$. Substituting this in Eq.(3.1) yields an average-case running time for insertion sort that is also a quadratic function of n . [Verify.]

3.2.2 Asymptotic notations

Now we get to the heart of the matter. Consider again Eq.(3.2) and Eq.(3.3) which indicate that the best-case and worst-case running times of insertion sort are linear and quadratic functions, respectively, of the size n of the input. There are two important observations that we need to point out regarding these equations.

First, recall that we arrived at these formulas by assigning abstract costs to the instructions of a hypothetical language (EASY) as they are executed by a hypothetical machine (RAM). Suppose we use instead an actual programming language (say C or Pascal or FORTRAN) on a real computer (say a Pentium III PC or an Apple G4 or a mainframe IBM 4381). Will the best-case running time still be a linear function of n , and will the worst-case running time still be a quadratic function of n ? The answer is ‘Yes’; only the coefficients a , b and c will change depending on the particular compiler and computer used. Thus Eq.(3.2) and Eq.(3.3) capture two fundamental performance characteristics

of insertion-sort as an algorithm, namely, linear best-case and quadratic worst-case running times, and these characteristics are independent from the particular software and hardware used to implement the algorithm.

Second, recall that our primary interest in analyzing an algorithm is to determine the *asymptotic* time and space complexities of the algorithm, i.e., how the running time and running space grow as the size of the input increases. For instance, how does Eq.(3.3) behave as n increases? It is not difficult to see that as n becomes larger and larger, the quadratic term accounts for most of the running time. Put another way, the quadratic term *dominates* the other terms in the equation; the dominant term suffices to explain the limiting behavior of the equation. Thus, we can simply drop all other terms and retain only the dominant one. Similar arguments can be made about Eq.(3.2).

The ideas which we want to convey in the preceding two paragraphs are neatly captured by a number of so-called **asymptotic notations** which computer scientists use to describe the time and space complexities of running algorithms. The three most commonly used are the **O-notation**, the **Ω -notation**, and the **Θ -notation**.

3.2.3 The O-notation

DEFINITION 3.1. We say that $g(n)$ is $O(f(n))$ if there exist two positive constants K and n_0 such that $0 \leq g(n) \leq K \cdot f(n)$ for all $n \geq n_0$.

The O-notation specifies an *upper bound* for a function *to within a constant factor* (K) *for sufficiently large n* ($n \geq n_0$). We do *not* indicate the values of the constants K and n_0 . Figure 3.5 shows the intuition behind the O-notation. Note that for $n \geq n_0$, the value of $g(n)$ is less than or at most equal to $K \cdot f(n)$. Put another way, $g(n)$ grows no faster than $f(n)$.

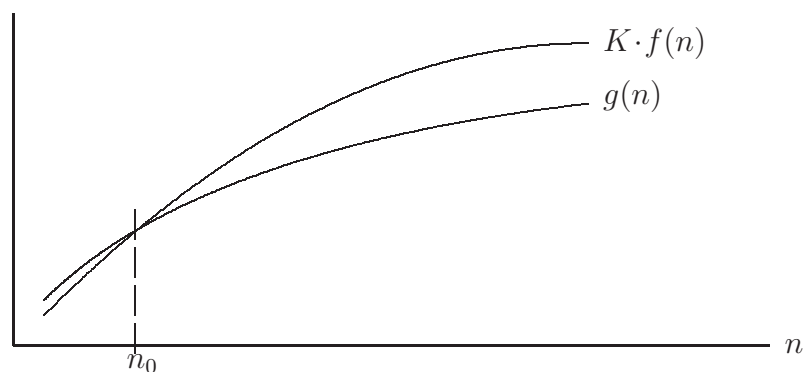


Figure 3.5 $g(n)$ is bounded from above by $f(n)$ to within a constant factor

The following examples should help us better understand the meaning of the O-notation.

64 SESSION 3. Algorithms

Example 3.1. On the basis of Eq.(3.3) we claim that

Fact: The running time of insertion sort is $O(n^2)$.

We reason out as follows:

1. The O-notation indicates an *upper bound*. When we say that the running time of insertion sort is $O(n^2)$, we mean that the running time is *at most* some constant times n^2 for *any* input of size n . This follows from Eq.(3.3) which indicates that the *worst case* running time of insertion sort is quadratic in n .
2. The O-notation is an *asymptotic notation*. It describes the limiting behavior of the function $T(n)$ in Eq.(3.3) as n increases. Hence we retain only the dominant term in $T(n)$, viz., $a n^2$. Furthermore, we also drop the coefficient a in $O(n^2)$ since we can consider it to be absorbed by the constant K in our definition of the O-notation.

The statement ‘The running time of insertion sort is $O(n^2)$ ’ gives us a picture of insertion sort behaving at its worst. As such, it is a guarantee that the running time cannot be any worse than quadratic in n ; it cannot for instance be cubic or exponential in n . Likewise, it is a measure that we can use when comparing insertion sort with other sorting algorithms.

Technically, it is correct to say that the running time of insertion sort is $O(n^3)$ or $O(2^n)$, since anything that is bounded from above by a quadratic in n is also bounded from above by a cubic or an exponential in n . However, we will use the O-notation to specify a *tight* upper bound, so the bounds $O(n^3)$ or $O(2^n)$ on insertion sort will *not* be acceptable.

Example 3.2. Assume that we have an algorithm whose input is of size n and which executes in

$$g(n) = 1^3 + 2^3 + 3^3 + \dots + n^3$$

steps. We assert that the algorithm has time complexity $O(n^4)$. To prove our assertion, let $K = 1$ and $n_0 = 1$. Using Eq.(2.40) and DEFINITION 3.1 of the O-notation, we have:

$$\left[\frac{n(n+1)}{2} \right]^2 \leq 1 n^4$$

or, simplifying

$$n^2 + n \leq 2 n^2$$

from which, finally

$$n \leq n^2$$

which is true for all $n \geq 1$.

Q.E.D.

We obtain the same result, without having to explicitly indicate the values of K and n_0 , as follows:

$$\begin{aligned}
 g(n) &= 1^3 + 2^3 + 3^3 + \dots + n^3 \\
 &= \left[\frac{n(n+1)}{2} \right]^2 && \text{from Eq.(2.40)} \\
 &= \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} \\
 &= O(n^4) && \text{Q.E.D.}
 \end{aligned}$$

As in Example 3.1, we retain only the dominant term $\frac{n^4}{4}$; furthermore, we discard the constant factor $\frac{1}{4}$.

Example 3.3. Find the time complexity of the following segment of EASY code:

```

for  $i \leftarrow 1$  to  $n - 1$  do
  for  $j \leftarrow i + 1$  to  $n$  do
    for  $k \leftarrow 1$  to  $j$  do
       $S$ : statements which take  $O(1)$  time
    endfor
  endfor
endfor

```

Let $T(n)$ be the number of times S is executed; then we have:

$$\begin{aligned}
 T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=1}^j 1 \\
 &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n j \\
 &= \sum_{i=1}^{n-1} \left[\sum_{j=1}^n j - \sum_{j=1}^i j \right] \\
 &= \sum_{i=1}^{n-1} \left[\frac{n(n+1)}{2} - \frac{i(i+1)}{2} \right] \\
 &= \frac{n(n+1)}{2} \sum_{i=1}^{n-1} 1 - \frac{1}{2} \sum_{i=1}^{n-1} i^2 - \frac{1}{2} \sum_{i=1}^{n-1} i \\
 &= \frac{n(n+1)}{2} (n-1) - \frac{1}{2} \frac{(n-1)n[2(n-1)+1]}{6} - \frac{1}{2} \frac{(n-1)n}{2} \\
 &= \frac{n^3}{3} - \frac{n}{3} \\
 &= O(n^3)
 \end{aligned}$$

3.2.4 The Ω -notation

DEFINITION 3.2. We say that $g(n)$ is $\Omega(f(n))$ if there exist two positive constants K and n_0 such that $0 \leq K \cdot f(n) \leq g(n)$ for all $n \geq n_0$.

The Ω -notation specifies a *lower bound* for a function *to within a constant factor* (K) for sufficiently large n ($n \geq n_0$). We do *not* indicate the values of the constants K and n_0 . Figure 3.6 shows the intuition behind the Ω -notation. Note that for $n \geq n_0$, the value of $g(n)$ is greater than or at least equal to $K \cdot f(n)$. Put another way, $g(n)$ grows no slower than $f(n)$.

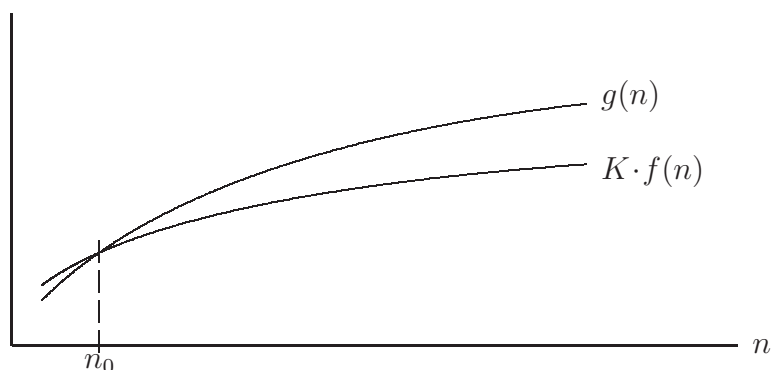


Figure 3.6 $g(n)$ is bounded from below by $f(n)$ to within a constant factor

The following examples should help us better understand the meaning of the Ω -notation.

Example 3.4. On the basis of Eq.(3.2) we claim that

Fact: The running time of insertion sort is $\Omega(n)$.

We reason out as follows:

1. The Ω -notation indicates a *lower bound*. When we say that the running time of insertion sort is $\Omega(n)$, we mean that the running time is *at least* some constant times n for *any* input of size n . This follows from Eq.(3.2) which indicates that the *best case* running time of insertion sort is linear in n .
2. The Ω -notation is an *asymptotic notation*. It describes the limiting behavior of the function $T(n)$ in Eq.(3.2) as n increases. Hence we retain only the dominant term in $T(n)$, viz., an . Furthermore, we also drop the coefficient a in $\Omega(n)$ since we can consider it to be absorbed by the constant K in our definition of the Ω -notation.

The statement ‘The running time of insertion sort is $\Omega(n)$ ’ gives us a picture of insertion sort behaving at its best. It tells us that the running time cannot be any better than linear in n ; this is already as best as it can get.

Example 3.5. Given two functions $f(n)$ and $g(n)$ we say that

$$g(n) = O(f(n)) \quad \text{if and only if} \quad f(n) = \Omega(g(n))$$

3.2.5 The Θ -notation

DEFINITION 3.3. We say that $g(n)$ is $\Theta(f(n))$ if there exist positive constants K_1 , K_2 and n_0 such that $0 \leq K_1 \cdot f(n) \leq g(n) \leq K_2 \cdot f(n)$ for all $n \geq n_0$.

The Θ -notation specifies a *lower* and an *upper bound* for a function *to within constant factors* (K_1 and K_2) *for sufficiently large n* ($n \geq n_0$). We do *not* indicate the values of the constants K_1 , K_2 and n_0 . Figure 3.7 shows the intuition behind the Θ -notation. Note that for $n \geq n_0$, the value of $g(n)$ lies between $K_1 \cdot f(n)$ and $K_2 \cdot f(n)$. Put another way, $g(n)$ grows neither faster nor slower than $f(n)$.

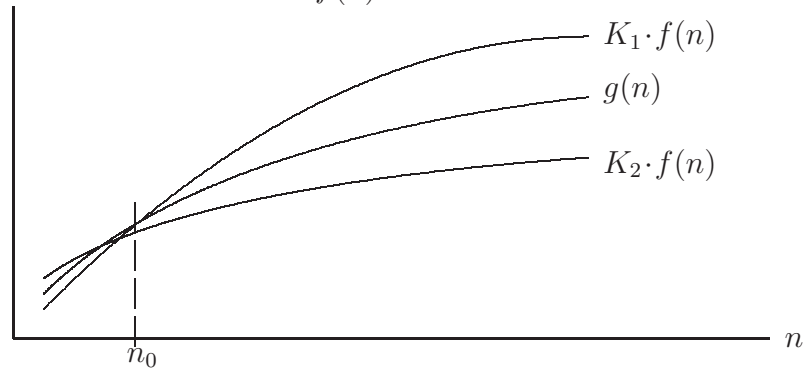


Figure 3.7 $g(n)$ is bounded from above and below by $f(n)$ to within constant factors

The following theorem relates the Θ -notation to the O - and Ω -notations.

THEOREM 3.1. For any two functions $f(n)$ and $g(n)$, $g(n) = \Theta(f(n))$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$.

The following examples should help us better understand the meaning of the Θ -notation.

Example 3.6. Given the polynomial $p(n) = \frac{n^2}{2} + 2n - 6$, we claim that $p(n) = \Theta(n^2)$. To prove our claim, we show that

$$K_1 n^2 \leq \frac{n^2}{2} + 2n - 6 \leq K_2 n^2 \quad \text{for all } n \geq n_0$$

for some positive constants K_1 , K_2 and n_0 . Dividing by n^2 , we obtain

$$K_1 \leq \frac{1}{2} + \frac{2}{n} - \frac{6}{n^2} \leq K_2$$

The left-hand inequality is satisfied for $n \geq 3$ if we choose $K_1 \leq \frac{1}{2}$, as you may easily verify. Similarly, the right-hand inequality is satisfied for $n \geq 1$ if we choose $K_2 \geq 1$ (verify). Hence, a particular choice of the constants is $K_1 = \frac{1}{2}$, $K_2 = 1$ and $n_0 = 3$; clearly, there are many other possible choices. Therefore it follows that $p(n) = \frac{n^2}{2} + 2n - 6 = \Theta(n^2)$. This example is actually just an instance of a more general result, as embodied in the following theorem:

THEOREM 3.2. Let $p(n) = \sum_{i=0}^d a_i n^i = a_0 + a_1 n + a_2 n^2 + \dots + a_d n^d$, where $a_d > 0$; then, $p(n) = \Theta(n^d)$.

Example 3.7. On the basis of Eq.(3.2), Eq.(3.3), THEOREM 3.1 and THEOREM 3.2, we claim:

1. The running time of insertion sort is *neither* $\Theta(n)$ *nor* $\Theta(n^2)$.
2. The best-case running time of insertion sort is $\Theta(n)$.
3. The worst-case running time of insertion sort is $\Theta(n^2)$.

3.2.6 The preponderance of the O-notation

Of the three asymptotic notations defined above, we will extensively use the O-notation which, you may recall, describes the worst-case behavior of an algorithm. There are compelling reasons for our interest in the worst-case performance of an algorithm, viz.,

1. We gain a deeper insight into an algorithm when it behaves at its worst than when it behaves at its best. (This applies not only to algorithms, but to people as well.) With this insight, we can better compare an algorithm with other competing algorithms.
2. Knowing the worst-case behavior of an algorithm *assures* us that things can't be any worse than this. (*Sagad na.*)
3. The average-case behavior of an algorithm may be just as bad as its worst-case behavior. For instance, the average-case and worst-case running times of insertion sort are both quadratic in n .

The following simple operations on the O-notation are readily verified by using DEFINITION 3.1:

$$\begin{array}{lll}
 (a) & f(n) & = O(f(n)) \\
 (b) & c \cdot f(n) & = O(f(n)) \quad \text{where } c \text{ is a constant} \\
 (c) & O(f(n)) + O(g(n)) & = O(\max[f(n), g(n)]) \\
 (d) & O(f(n)) \cdot O(g(n)) & = O(f(n) \cdot g(n))
 \end{array} \tag{3.4}$$

Example 3.8. To prove Eq. (3.4c), we proceed as follows:

Let

$$T_1(n) = O(f(n)); \text{ then } T_1(n) \leq c_1 \cdot f(n) \text{ for } n \geq n_1$$

$$T_2(n) = O(g(n)); \text{ then } T_2(n) \leq c_2 \cdot g(n) \text{ for } n \geq n_2$$

Then

$$\begin{aligned}
 T_1(n) + T_2(n) &\leq c_1 \cdot f(n) + c_2 \cdot g(n) \\
 &\leq (c_1 + c_2) \cdot \max[f(n), g(n)] \text{ for } n \geq \max[n_1, n_2] \\
 &= O(\max[f(n), g(n)]) \quad \text{Q.E.D.}
 \end{aligned}$$

3.2.7 Comparative growth of functions

In applying the O -, Ω - and Θ -notations, we need to identify which term of an equation which describes the running time or running space of an algorithm has the fastest rate of growth, so that we can discard the others and retain only this asymptotically dominant one. For instance, how do we express the function $T_1(n) = 2^{\frac{n}{2}-2} + 50n^{10} - 100$ in terms of the O -notation? What about $T_2(n) = n^2 + 8n(\log_b n)^6 + 18n$? To this end we apply the following equations which describe the comparative growth of common functions.

1. Polynomial vs exponential — for all positive real constants d and $x > 1$:

$$\lim_{n \rightarrow \infty} \frac{n^d}{x^n} = 0 \quad (3.5)$$

In plain English: any positive exponential function grows faster than any polynomial function.

2. Polylogarithmic vs polynomial — for all positive real constants k and d :

$$\lim_{n \rightarrow \infty} \frac{(\log_b n)^k}{n^d} = 0 \quad (3.6)$$

In plain English: any positive polynomial function grows faster than any polylogarithmic function.

Thus we have: $T_1(n) = 2^{\frac{n}{2}-2} + 50n^{10} - 100 = \frac{2^{\frac{n}{2}}}{2^2} + 50n^{10} - 100 = O(2^{\frac{n}{2}})$ and $T_2(n) = n^2 + 8n(\log_b n)^6 + 18n = O(n^2)$.

3.2.8 Complexity classes

The table below shows the different complexity classes of the various algorithms we will encounter in this book. You may not be familiar with some of the examples listed; in due time you will be.

Complexity class	Common name	Typical example
$O(1)$	constant	boundary-tag technique
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	linear search
$O(n \log n)$	$n \log n$	heapsort
$O(n^2)$	quadratic	insertion sort
$O(n^3)$	cubic	Floyd's algorithm
$\Omega(2^n)$	exponential	algorithm THP

Figure 3.8 Common complexity classes of running algorithms

It is important to appreciate the differences among the complexity classes listed in Figure 3.8. One way to do this is to determine the running time of an algorithm which takes $f(n)$ steps to finish for an input of size n assuming that each step takes t units of time. Figure 3.9 shows the results for a computer that can perform 1 billion steps per

70 SESSION 3. Algorithms

second, i.e., $t = 1$ nanosecond. Running times that are less than a second are indicated as ‘ < 1 sec’; you may easily verify that these values range from a low of 5.64 nanoseconds to a high of 100 milliseconds (no more than the time it takes you to press the ENTER key on your keyboard).

$f(n)$	$n = 50$	$n = 1000$	$n = 10,000$	$n = 100,000$	$n = 1,000,000$
$\log_2 n$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
n	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
$n \log_2 n$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
n^2	< 1 sec	< 1 sec	< 1 sec	10.00 secs	16.67 min
n^3	< 1 sec	1.00 sec	16.67 min	11.57 days	31.69 yrs
2^n	13.03 days	3.40×10^{284} yrs	6.32×10^{2993} yrs	$3.17 \times 10^{30,086}$ yrs	$3.14 \times 10^{301,013}$ yrs

Figure 3.9 Comparing complexity classes: running times at 1 step/nanosecond

It is obvious from Figure 3.9 that the exponential complexity class 2^n is quite apart from the rest. However, even among the polynomial complexity classes the differences become dramatic as the size of the input increases.

Another way to illustrate the differences among the different complexity classes is to determine the size of the largest problem that can be solved in time T by an algorithm which takes $f(n)$ steps to finish assuming each step takes t units of time. Figure 3.10 shows the results for $t = 1$ nanosecond.

$f(n)$	$T = 1$ min	$T = 1$ hour	$T = 1$ day	$T = 1$ week	$T = 1$ year
n	6.00×10^{10}	3.60×10^{12}	8.64×10^{13}	6.05×10^{14}	3.16×10^{16}
$n \log_2 n$	1.94×10^9	9.86×10^{10}	2.11×10^{12}	1.38×10^{13}	6.42×10^{14}
n^2	2.45×10^5	1.90×10^6	9.30×10^6	2.46×10^7	1.78×10^8
n^3	3.91×10^3	1.53×10^4	4.42×10^4	8.46×10^4	3.16×10^5
2^n	35 (45)	41 (51)	46 (56)	49 (59)	54 (64)

Figure 3.10 Comparing complexity classes: largest problem size at 1 step/nanosecond

Note that for a given amount of time T , the size of the problem that can be solved decreases significantly as we climb the complexity scale. Note also that if the algorithm takes 2^n steps, increasing the allotted time from 1 minute to 1 year does not even double the size of the problem that can be solved. Similarly, increasing the computer speed a thousand times from a billion to a trillion steps per second increases the problem size that can be solved in time T by only 10, as indicated by the quantities enclosed in parenthesis. In the final analysis, it is not CPU speed, but rather algorithmic complexity, which determines how large a problem we can solve on a computer.

Summary

- An algorithm must be communicated to both humans and machines. Pseudocode is rich enough to capture the essence of an algorithm and bring one to an understanding of how an algorithm works. Deeper insight into an algorithm, its limitations, its idiosyncracies and the like, can be gained by transcribing it from pseudocode into a formal programming language and executing it on a machine.
- An algorithm requires both *time* and *space* for it to accomplish its task. The amount of time it uses depends on the *size* of the input to the algorithm.
- To *analyze an algorithm* is to determine the *rate* or *order of growth* of the time and space it utilizes as the size of the input increases. This order of growth, expressed as a function of the size of the input and evaluated in the limit, is called the *asymptotic time complexity* (or simply, the *running time*) and *asymptotic space complexity* (or simply, the *running space*) of the algorithm.
- Running time is often more crucial than running space. For one, time spent is time lost; but space can always be reused.
- The running time of an algorithm depends also on the *nature* of the input, in addition to its size. For an input of size n , the running time may be categorized as *best-case*, *worst-case* or *average-case*, depending on what input of size n is given.
- Three very important *asymptotic notations* that are used to describe the time and space complexities of running algorithms are the *O-notation*, *Ω -notation* and *Θ -notation*. Informally, the *O-notation* says ‘This algorithm’s performance can’t be any worse than this’; the *Ω -notation* says ‘This algorithm’s performance can’t be any better than this.’
- The *O-notation* is generally used to classify algorithms into *complexity classes*. The time complexity of a large number of algorithms fall into the following few classes: *constant*, *logarithmic*, *linear*, ‘ *n -log- n* ’, *quadratic*, *cubic* and *exponential*.

Exercises

1. Show that the average case running time of insertion sort is a quadratic function of the size of the input n .
2. An algorithm executes in $3 + 6 + 9 + \dots + 3n$ steps. What is the time complexity of the algorithm?
3. An algorithm executes in $1(2) + 2(3) + \dots + (n - 1)n$ steps. What is the time complexity of the algorithm?

72 SESSION 3. Algorithms

4. Find the time complexity of the following segment of EASY code:

```

for  $i \leftarrow 1$  to  $n$  do
  for  $j \leftarrow i$  to  $n$  do
    for  $k \leftarrow 1$  to  $j$  do
      statements which take  $O(1)$  time
    endfor
  endfor
endfor

```

5. Find the time complexity of the following segment of EASY code:

```

for  $s \leftarrow 1$  to  $n$  do
  for  $i \leftarrow 0$  to  $n - s$  do
    for  $r \leftarrow i + 1$  to  $i + s$  do
      statements which take  $O(1)$  time
    endfor
  endfor
endfor

```

6. Arrange the following in order of increasing complexity, for sufficiently large n .

$n \log n$, $\log \log n$, 10^n , $2^{\log n}$, $\log^2 n$, $\sqrt{\log n}$, $n^2 \log n$, n^3 , $5n^{\frac{3}{2}}$, $n \log^2 n$, $10n$, n^2 , 2^{2^n}

7. It is estimated that the sun will burn out in 5 billion years. What is the largest problem size, n , that can be solved on a computer which can perform 1 billion steps per second, before the sun grows cold, if the algorithm used requires

(a) n^2 steps? (b) n^3 steps? (c) 2^n steps? (d) 10^n steps? (e) $n!$ steps?

8. The number of combinations of n objects taken k at a time is given by the recursive formula

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad 0 \leq k \leq n$$

Write a recursive EASY procedure to find $\binom{n}{k}$.

9. The *powerset* $\mathcal{P}(S)$ of an n -set S , $n \geq 0$, is the set of all subsets of S . For example, given the set $S = \{a, b, c, d\}$, then $\mathcal{P}(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}, \{a, b, c, d\}\}$. Write a recursive EASY procedure to generate $\mathcal{P}(S)$ given S .
10. (a) Write an iterative EASY procedure to implement Euclid's algorithm.
 (b) Write a recursive EASY procedure to implement Euclid's algorithm.
 (c) Show that the time complexity of Euclid's algorithm is $O(\log n)$.
11. Write an EASY procedure to implement Algorithm G (Approximate graph coloring) in Session 1. What is the time complexity of your procedure?
12. An n th degree polynomial in x may be written in the form

$$p_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n$$

Devise, and implement in EASY, an $O(n)$ algorithm to evaluate $p_n(x)$, given x .

13. Devise and implement in EASY an $O(\log n)$ algorithm to evaluate x^n , $n \geq 0$.
14. Redo the table in Figure 3.9 for a computer that can perform 1 trillion steps per second.
15. Redo the table in Figure 3.10 for a computer that can perform 1 trillion steps per second.

Bibliographic Notes

A description of SPARKS, the progenitor of EASY, is given in HOROWITZ[1976], pp. 8–15. Procedure `BINARY_SEARCH`, our first example of an EASY procedure, is an implementation of Algorithm B (*Binary search*) given in KNUTH3[1998], p. 410. Procedure `INVERT` is an EASY implementation of the algorithm to invert in-place a singly-linked list given in KNUTH1[1997], p. 546.

The analysis of insertion sort given in section 3.2.1 follows that found in CORMEN[2001], pp. 23–27. KNUTH3[1998], pp. 80–102, gives a detailed analysis of the algorithm (which Knuth calls *straight insertion sort*), and discusses a number of variations on the general theme of sorting by insertion (binary insertion, two-way insertion, Shell’s diminishing increment sort, list insertion, etc.).

Standish[1994], pp. 213–219, gives an ‘intuition-building’ introduction to the O -notation and an insightful comparison of complexity classes.

NOTES

SESSION 4

Stacks

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Explain implementation issues pertaining to the array representation of a stack.
2. Explain implementation issues pertaining to the linked list representation of a stack.
3. Discuss some important problems in Computer Science whose solution requires the use of a stack.
4. Generalize the algorithm to convert arithmetic expressions from infix to postfix form to include expressions involving other operators (relational, logical, etc.).
5. Explain implementation issues pertaining to the representation of multiple stacks in a single array.
6. Describe memory reallocation strategies for stacks which coexist in a fixed region of memory.

READINGS HOROWITZ[1976], pp. 91–97; TREMBLAY[1976], pp. 191–207; TENENBAUM[1986], pp. 87–106; KNUTH1[1997], pp. 246–250; STANDISH[1980], pp. 28–39.

DISCUSSION

One of the simplest but nonetheless extremely important data structure used in computer algorithms is the stack. Stacks are utilized in such diverse tasks as list and tree traversals, evaluation of expressions, resolving procedure calls, and so on.

As an abstract data type, we can think of a stack as a linearly ordered set of data elements on which is imposed the discipline of ‘last-in, first-out’. The stack elements may be any entity, such as a token, an activation record, intermediate values in a computation, a node of a binary tree, a vertex or an edge of a graph, and so on.

The two basic operations on a stack are

1. insert (or *push*) an element at the top of the stack
2. delete (or *pop*) the top element of the stack

Note that these are the operations which maintain the LIFO discipline on the elements of a stack.

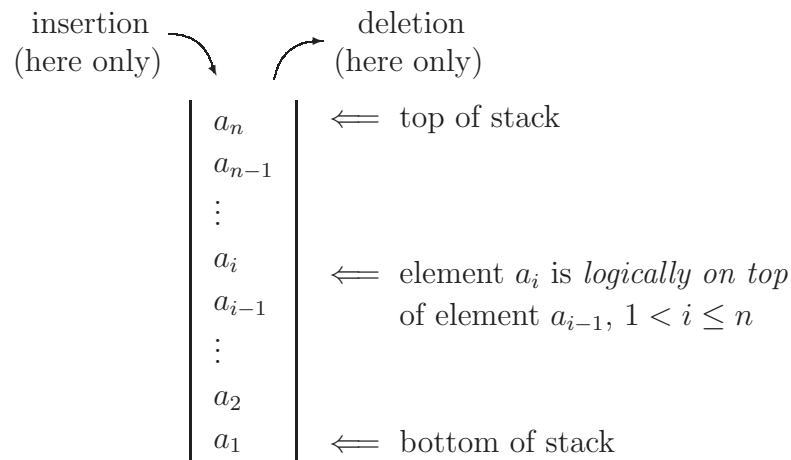


Figure 4.1 The stack as an abstract data type

The other auxiliary operations defined on a stack are:

3. initialize the stack
4. test if the stack is empty
5. test if the stack is full

Note that in the abstract, a stack may be empty, but it may not be full. There is nothing in the definition of a stack which disallows pushing yet one more element onto it. The state whereby a stack is full is a consequence of implementing the stack ADT on a machine with finite memory.

Occasionally we may want to examine the top element of a stack without removing it from the stack. Such an operation is allowable, but only as a sequence of a pop followed by a push operation. That is, pop the stack (this retrieves the top element and logically deletes it from the stack), examine the element, then push it back onto the stack.

There are two ways to implement the stack ADT as a concrete data structure, namely:

1. sequential implementation – the stack is stored in a one-dimensional array
2. linked implementation – the stack is represented as a linked linear list

4.1 Sequential implementation of a stack

Figure 4.2 shows the sequential representation of a stack in an array S of size n . The elements marked ‘x’ comprise the stack, with the top element pointed to by the variable top . Note that the array indices start at 1; thus $S(1)$ is the bottom element and $S(6)$ is the top element of the stack. Be sure to keep the distinction between the stack (of size top) and the array (of size n) in which the stack ‘resides’.

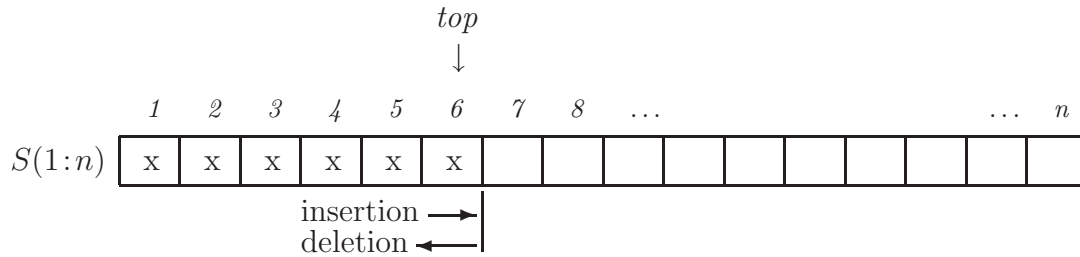


Figure 4.2 Array implementation of a stack

To insert an element, say x , onto the stack we set

$$\begin{aligned} top &\leftarrow top + 1 \\ S(top) &\leftarrow x \end{aligned}$$

If after an insertion top equals n , we say that the *stack is full*; all preallocated space for the stack is in use. An attempt to insert onto a full stack results in an *overflow condition*. Clearly such a condition will arise if, in some particular application, not enough space has been initially allocated for the stack. The simple solution then would be to allocate more and rerun the program. On the other hand, an overflow condition may also occur due to an error in the program; for instance, having a push operation inside an infinite loop. In any case, an implementation of the insert operation for a sequentially allocated stack must test for the occurrence of an overflow condition.

To delete the top element of the stack we set

$$\begin{aligned} x &\leftarrow S(top) \\ top &\leftarrow top - 1 \end{aligned}$$

The first statement retrieves the top element and stores it in x ; the second statement ‘logically’ deletes the element from the stack (‘physically’, the element is still in the array, a consequence of non-destructive readout). If after a deletion top equals 0, we say that the *stack is empty*. An attempt to delete from an empty stack results in an *underflow condition*. Whereas the condition whereby a stack becomes empty is a normal occurrence, an underflow condition is usually abnormal. Attempting to delete an element from an empty stack is often an indication that something is amiss. An implementation of the delete operation must invariably test for the occurrence an underflow condition.

Consistent with the condition $top = 0$ indicating an empty stack, we *initialize a stack* by setting the value of top to zero, thus

$$top \leftarrow 0$$

We will use the notation $\mathbb{S} = [S(1:n), top]$ to denote the *data structure* depicted in Figure 4.2. This notation simply means that this particular implementation, \mathbb{S} , of the stack ADT consists of two *components*, namely, an array S of size n in which the stack elements reside, and a pointer top to the top element of the stack. In the EASY procedures which implement the various stack operations which follow, we assume that a pointer to the structure \mathbb{S} is passed to the implementing procedure, allowing the procedure access to the components of the structure. Any change in any of these components is seen by the calling procedure.

4.1.1 Implementing the auxiliary operations

The following EASY procedures implement stack operations 3 and 4 listed above. We choose to embed the test for a full stack in the procedure which implements the insert operation, obviating the need for a separate procedure for operation 5.

```

1  procedure InitStack( $\mathbb{S}$ )
2     $top \leftarrow 0$ 
3  end InitStack

```

Procedure 4.1 Initializing a stack (array implementation)

```

1  procedure IsEmptyStack( $\mathbb{S}$ )
2    return( $top = 0$ )
3  end IsEmptyStack

```

Procedure 4.2 Testing if a stack is empty (array implementation)

Procedure IsEmptyStack is written as a function which returns **true** if $top = 0$, else **false**.

4.1.2 Implementing the insert operation

The EASY procedure PUSH implements the insert operation for a sequentially allocated stack $\mathbb{S} = [S(1:n), top]$ following the conventions stated above. The procedure receives a pointer to \mathbb{S} (line 1); the variable x contains the element to be pushed onto the stack.

```

1  procedure PUSH( $\mathbb{S}, x$ )
2    if  $top = n$  then call STACKOVERFLOW
3     $top \leftarrow top + 1$ 
4     $S(top) \leftarrow x$ 
5  end PUSH

```

Procedure 4.3 Insertion onto a stack (array implementation)

Line 2 tests if the stack is full; if it is, procedure `STACKOVERFLOW` is invoked to handle the occurrence of an overflow condition. It may take one of two possible courses of action:

1. Reallocate memory; if additional space becomes available, return control to `PUSH` via a **return** statement so that the insertion operation can be completed. (We will address the issue of memory reallocation in Section 4.4 in connection with the coexistence of multiple stacks in a fixed region of memory.)
2. Issue an error message and return control to the operating system via a **stop** statement. (For a single stack, this is the more likely option to take.)

If space is available the new element is pushed onto the stack (lines 3–4). Unless `STACKOVERFLOW` is invoked, the insert operation clearly takes constant time.

4.1.3 Implementing the delete operation

The `EASY` procedure `POP` implements the delete operation for a sequentially allocated stack $\mathbb{S} = [S(1:n), top]$ following the conventions stated above. `POP` receives pointers to \mathbb{S} and to x (line 1). Upon entry into `POP` the variable x is undefined; upon return to the calling program, it contains the deleted top element of the stack (line 3).

```

1  procedure POP( $\mathbb{S}, x$ )
2  if  $top = 0$  then call STACKUNDERFLOW
3      else [ $x \leftarrow S(top)$ ;  $top \leftarrow top - 1$ ]
4  end POP

```

Procedure 4.4 Deletion from a stack (array implementation)

Procedure `STACKUNDERFLOW` (line 2) handles the occurrence of an underflow condition. It is left uncoded; the action it takes depends on the particular application. Otherwise, the deleted element is retrieved in x . A successful delete operation clearly takes $O(1)$ time.

In the remainder of this section, we will consider how the above `EASY` procedures might be implemented in the Pascal and C languages.

4.1.4 A Pascal implementation of the array representation of a stack

The following Pascal program implements the data structure $\mathbb{S} = [S(1:n), top]$ and its associated procedures. Also included is a short program segment which illustrates how the procedures are invoked.

To implement the structure \mathbb{S} , we use a Pascal structured data type called a *record*, which consists of any number of components, each of a possibly different type. In the

present case we call the record type **Stack**; its components are **top** which is of type **integer**, and **Stack** which is an array of size **n**, where each element is of type **StackElemType**. We define **StackElemType** to be of type **integer**; other possible data types are **real**, **char**, **boolean**, an enumerated type, and so on, depending on the application. Alternatively, we could have externally defined **StackElemType**.

With this Pascal implementation of \mathbb{S} , we proceed to write the code for the different stack operations. It is instructive to compare these Pascal procedures with their counterparts in EASY.

```

program ArrayStack;
const
    n = 100;
type
    StackElemType = integer;
    Stack = record
        top: integer;
        Stack: array[1..n] of StackElemType
    end;

procedure InitStack(var S:Stack);
begin
    S.top := 0
end;

function IsEmptyStack(S:Stack):boolean;
begin
    IsEmptyStack := (S.top = 0)
end;

procedure StackOverflow(S:Stack);
begin
    writeln('Stack overflow detected. ');
    halt
end;

procedure StackUnderflow;
begin
    writeln('Stack underflow detected. ');
    halt
end;

procedure PUSH(var S:Stack; x:StackElemType);
begin
    with S do begin
        if top = n then
            StackOverflow
        else begin
            top := top + 1;
            Stack[top] := x
        end;
    end;
end;

```



```

procedure POP(var S:Stack; var x:StackElemType);
begin
    with S do begin
        if top = 0 then
            StackUnderflow
        else begin
            x := Stack[top];
            top := top - 1
        end;
    end;
end;
{Sample calling sequence}
var S:Stack; x:StackElemType; i:integer;
begin
    InitStack(S);
    for i := 1 to 5 do
        PUSH(S,100*i);
    while not IsEmptyStack(S) do
        begin
            POP(S,x);
            writeln(x)
        end;
end.

```

Program 4.1 A Pascal implementation of the array representation of a stack

When run, the preceeding program will generate the following output:

```

500
400
300
200
100

```

4.1.5 A C implementation of the array representation of a stack

The following C program implements the data structure $\mathbb{S} = [S(1:n), top]$ and its associated procedures. Also included is a short program segment which illustrates how the procedures are invoked.

To implement the structure \mathbb{S} , we use a C construct called a *structure*, which consists of any number of components called *structure elements*, each of a possibly different type. In the present case we call the structure **stack** and this name also serves as its type specifier. Its components are **top** which is of type **int**, and **Stack** which is an array of size **n**, where each element is of type **StackElemType**. We define **StackElemType** to be of type **int**; other possible data types are **char**, **float**, **double float**, a structure name, and so on, depending on the application.

Although we define the array to be of size 101, the stack can only have at most 100 elements. This is because we do not use the zeroth cell in order to be consistent with the boundary condition $top = 0$ indicating an empty stack. In cases where we cannot reliably predict how large the stack can get, we may use a dynamically allocated array instead.

Having chosen a particular C implementation of \mathbb{S} , we proceed to write the code for the different stack operations. It is instructive to compare these C procedures with the corresponding EASY and Pascal procedures.

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

#define n 101
typedef int StackElemType;

struct stack
{
    int top;
    StackElemType Stack[n];
};
typedef struct stack Stack;

void InitStack(Stack *S)
{
    S->top = 0;
}

int IsEmptyStack(Stack *S)
{
    return (S->top == 0);
}

void StackOverflow(void)
{
    printf("Stack overflow detected.\n");
    exit(1);
}

void StackUnderflow(void)
{
    printf("Stack underflow detected.\n");
    exit(1);
}

void PUSH(Stack *S, StackElemType x)
{
    if(S->top == n)
        StackOverflow();
    else {
        ++S->top;
        S->Stack[S->top] = x;
    }
}
```

```

void POP(Stack *S, StackElemType *x)
{
    if(S->top == 0)
        StackUnderflow();
    else {
        *x = S->Stack[S->top];
        --S->top; }
}

main()
{
    Stack S; StackElemType x; int i;
    clrscr();
    InitStack(&S);
    for(i=1; i<=5; i++) PUSH(&S,100*i);
    while(!IsEmptyStack(&S)) {
        POP(&S,&x);
        printf("%d\n",x); }
    return 0;
}

```

Program 4.2 A C implementation of the array representation of a stack

Program 4.2 produces the same output as Program 4.1.

4.2 Linked list implementation of a stack

The figure below shows a stack represented as a singly-linked linear list. The top element is x_1 and the bottom element is x_n . The pointer to the list is top , which is also the pointer to the top element of the stack. Note that the *LINK* field of the last node is set to Λ .

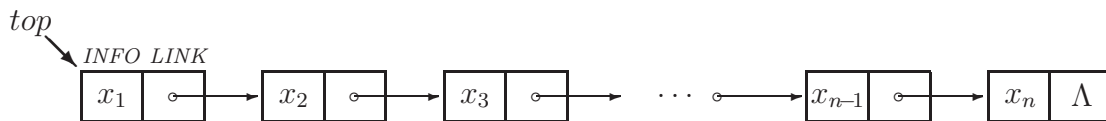


Figure 4.3 A stack represented as a linked linear list

To push a new element x onto the stack we obtain a node from the memory pool by invoking GETNODE, store x in the *INFO* field of the new node and then append the node at the head of the list.

```

call GETNODE( $\alpha$ )
 $INFO(\alpha) \leftarrow x$ 
 $LINK(\alpha) \leftarrow top$ 
 $top \leftarrow \alpha$ 

```

To pop the top element of the stack we retrieve the content of the *INFO* field of the first (or top) node of the list and store this in some variable, say x ; then we update top to point to the next node in the list.

```
 $x \leftarrow INFO(top)$ 
 $top \leftarrow LINK(top)$ 
```

Note that if these operations are performed on a one-element stack (or one-node list), then top becomes equal to Λ . Hence the relation $top = \Lambda$ means the stack is empty; consistent with this condition, we initialize the stack by setting

```
 $top \leftarrow \Lambda$ 
```

So that the deleted node does not end up as ‘garbage’, we should return it to the memory pool. This is readily accomplished by calling `RETNODE`. With these additional ‘housekeeping’ chores we obtain the following sequence of instructions for the `POP` operation.

```
 $\alpha \leftarrow top$ 
 $x \leftarrow INFO(top)$ 
 $top \leftarrow LINK(top)$ 
call RETNODE( $\alpha$ )
```

We will use the notation $\mathbb{S} = [(INFO, LINK), top]$ to denote the data structure depicted in Figure 4.3. This notation means that this particular implementation, \mathbb{S} , of the stack ADT consists of two components, namely, a linked linear list consisting of nodes, each with node structure $(INFO, LINK)$, and a pointer top to the top element of the stack. As with the array implementation of a stack, we assume that a pointer to the structure \mathbb{S} is passed to the procedures which perform the different stack operations, allowing access to the components of the structure. Any change in any of these components is seen by the calling procedure.

4.2.1 Implementing the auxiliary stack operations

The following `EASY` procedures implement operations 3 and 4 listed above for the stack ADT.

```
1  procedure InitStack( $\mathbb{S}$ )
2     $top \leftarrow \Lambda$ 
3  end InitStack
```

Procedure 4.5 Initializing a stack (linked-list implementation)

```
1  procedure IsEmptyStack( $\mathbb{S}$ )
2    return( $top = \Lambda$ )
3  end IsEmptyStack
```

Procedure 4.6 Testing if a stack is empty (linked list implementation)

Procedure `IsEmptyStack` is written as a function which returns **true** if $top = \Lambda$, else **false**.

4.2.2 Implementing the insert operation

The EASY procedure PUSH implements the insert operation for a stack $\mathbb{S} = [(INFO, LINK), top]$ following the conventions stated above. The procedure receives pointers to \mathbb{S} and x (line 1); the variable x contains the element to be pushed onto the stack.

```

1  procedure PUSH( $\mathbb{S}, x$ )
2  call GETNODE( $\alpha$ )
3   $INFO(\alpha) \leftarrow x$ 
4   $LINK(\alpha) \leftarrow top$ 
5   $top \leftarrow \alpha$ 
6  end PUSH

```

Procedure 4.7 Insertion onto a stack (linked list implementation)

With the linked list representation of a stack, there is no condition which indicates a ‘full stack’. However we get the equivalent of a ‘stack overflow’ if GETNODE fails to obtain a node from the memory pool. Recall from Session 1 that in such an eventuality GETNODE issues an error message and returns control to the runtime system via a **stop** statement. This means that if GETNODE returns to PUSH, the allocation *is* successful and insertion can proceed. Assuming that GETNODE takes constant time, the insert operation clearly takes $O(1)$ time as well.

4.2.3 Implementing the delete operation

The EASY procedure POP implements the delete operation for a stack $\mathbb{S} = [(INFO, LINK), top]$ following the conventions stated above. POP receives pointers to \mathbb{S} and x (line 1). Upon entry into POP the variable x is undefined; upon return to the calling program it contains the deleted top element of the stack (line 4).

```

1  procedure POP( $\mathbb{S}, x$ )
2  if  $top = \Lambda$  then call STACKUNDERFLOW
3      else [ $\alpha \leftarrow top$ 
4           $x \leftarrow INFO(top)$ 
5           $top \leftarrow LINK(top)$ 
6          call RETNODE( $\alpha$ )]
7  end POP

```

Procedure 4.8 Deletion from a stack (linked list implementation)

Line 2 tests if the stack is empty and, as before, invokes STACKUNDERFLOW to handle the occurrence of an underflow condition. Otherwise, a temporary pointer to the node to be deleted is retained in α (line 3) before top is updated to point to the new top node (line 5). Finally RETNODE is invoked with argument α (line 6) to return the deleted node to the memory pool. Assuming that RETNODE takes constant time, a successful delete operation clearly takes $O(1)$ time.

4.2.4 A Pascal implementation of the linked list representation of a stack

The following Pascal program implements the data structure $S = [(INFO, LINK), top]$ of Figure 4.3 and its associated procedures.

As with the array representation (see Program 4.1), we use the Pascal data type **record** to implement the linked list representation of a stack. To this end, we define a record type called **StackNode** with components **INFO** and **LINK** to represent a node in the linked list, and another record type called **Stack** with component **top**, which is a pointer to the record type **StackNode**. Together, **StackNode** and **Stack** represent the data structure S . The stack elements are assumed to be integers.

```

program LinkedStack;
type
  StackElemType = integer;
  StackPtr = ^StackNode;
  StackNode = record
    INFO: StackElemType;
    LINK: StackPtr
  end;
  Stack = record
    top: StackPtr
  end;

procedure InitStack(var S:Stack);
begin
  S.top :=  $\Lambda$ 
end;

function IsEmptyStack(S:Stack):boolean;
begin
  IsEmptyStack := (S.top =  $\Lambda$ )
end;

{See Program 4.1 for procedures StackOverflow and StackUnderflow.}

procedure PUSH(var S:Stack; x:StackElemType);
  var alpha:StackPtr;
begin
  New(alpha);
  if alpha = nil then
    StackOverflow
  else begin
    alpha^.INFO := x;
    alpha^.LINK := S.top;
    S.top := alpha
  end;
end;

```

```

procedure POP(var S:Stack; var x:StackElemType);
    var alpha:StackPtr;
begin
    if S.top = nil then
        StackUnderflow
    else begin
        x := S.top^.INFO;
        alpha := S.top;
        S.top := S.top^.LINK;
        Dispose(alpha)
    end;
end;

{Sample calling sequence}

    var S:Stack; x:StackElemType; i:integer;
begin
    InitStack(S);
    for i := 1 to 5 do
        PUSH(S,100*i);
    while not IsEmptyStack(S) do
        begin
            POP(S,x);
            writeln(x)
        end;
end.

```

Program 4.3 A Pascal implementation of the linked list representation of a stack

Note that the sample calling sequence given above is exactly the same calling sequence we used in Program 4.1 for the array implementation of a stack. How the stack is implemented, sequential or linked, is irrelevant to the calling program. Needless to say, Program 4.3 generates the same output as Program 4.1.

4.2.5 A C implementation of the linked list representation of a stack

The following C program implements the data structure $\mathbb{S} = [(INFO, LINK), top]$ of Figure 4.3 and its associated procedures.

As with the array representation (see Program 4.2), we use a C structure to implement the linked list representation of a stack. To this end, we define a structure called **stacknode** with structure elements (or components) **INFO** and **LINK** to represent a node in the linked list, and another structure **stack** with component **top**, which is a pointer to **stacknode**. Together, **stacknode** and **stack** represent the data structure \mathbb{S} . The stack elements are assumed to be integers.

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

```

```

typedef int StackElemType;
typedef struct stacknode Stacknode;
struct stacknode
{
    StackElemType INFO;
    StackNode *LINK;
};
struct stack
{
    StackNode *top;
};
typedef struct stack Stack;
void InitStack(Stack *S)
{
    S->top = NULL;
}
int IsEmptyStack(Stack *S)
{
    return (S->top == NULL);
}
/* See Program 4.2 for procedures StackOverflow and StackUnderflow. */
void PUSH(Stack *S, StackElemType x)
{
    StackNode *alpha;
    alpha = (StackNode *) malloc(sizeof(StackNode));
    if(alpha == NULL)
        StackOverflow();
    else {
        alpha->INFO = x;
        alpha->LINK = S->top;
        S->top = alpha; }
}
void POP(Stack *S, StackElemType *x)
{
    StackNode *alpha;
    if(S->top == NULL)
        StackUnderflow();
    else {
        alpha = S->top;
        *x = S->top->INFO;
        S->top = S->top->LINK;
        free(alpha); }
}
main() /* See Program 4.2 for a sample calling sequence. No changes needed. */

```

Program 4.4 A C implementation of the linked list representation of a stack

4.3 Applications of stacks

We will now consider two problems in which the stack plays a prominent role in the solution. In subsequent sessions, we will encounter more applications of stacks.

4.3.1 A simple pattern recognition problem

Let

$$\begin{aligned} \{a, b\}^* &= \text{the set of all strings over the alphabet } \{a, b\}, \text{ including the null string } \Lambda \\ &= \{\Lambda, a, b, aa, ab, ba, bb, abb, baa, abba, aaaa, bbaa, \dots\} \\ w &= \text{any string in } \{a, b\}^* \\ w^R &= \text{the reverse of } w \text{ (e.g., if } w = abb, w^R = bba) \\ P &= \{wcw^R\} \end{aligned}$$

Some of the strings in P are $c, aca, bcb, abbcbb, baacaab$, etc. The following strings are *not* in P : $abba, abaabcbabba, abacab, bbaacaabbb$.

Webster defines a *palindrome* as ‘a word, phrase, or sentence which reads the same forward and backward’. If we include in the definition any string with the same property, then the strings in P are palindromes, albeit of a restricted kind. We want to devise an algorithm which, given a string on the alphabet $\{a, b, c\}$, will determine whether or not the string is in P .

The algorithm to solve this simple pattern recognition problem consists essentially of two passes, or loops. In the first pass, we extract the characters from the first half of the string and push these onto a stack. In the second pass, we extract the characters from the second half of the string and compare these with the characters stored in the stack.

Algorithm InP. Given an input string on the alphabet $\{a, b, c\}$, algorithm returns **true** to indicate that the string is in P , and **false** otherwise.

1. [Process first half] Get next character, say *char* from input string. If *char* is ‘a’ or ‘b’, push *char* onto stack and repeat this step. If *char* is ‘c’, go to step 2. If *char* is the blank character, return **false** (no second half).
2. [Process second half] Get next character, say *char*, from input string. Pop top element of stack, say x . If $char = x$, repeat this step (OK so far). If $char \neq x$, return **false**. If *char* is the blank character and stack is empty, return **true**; else, return **false**.

The EASY function InP implements the algorithm as described. It invokes procedures InitStack(\mathbb{S}), IsEmptyStack(\mathbb{S}), PUSH(\mathbb{S}, x) and POP(\mathbb{S}, x) of the previous sections. How the stack \mathbb{S} is implemented (array or linked list) is immaterial to InP.

Note that the second loop is on the stack; the test ‘**while not** IsEmptyStack(\mathbb{S})’ in line 8 guarantees that an underflow condition will not occur.

```

1   procedure InP(string)
▷   Given an input string 'string' on the alphabet {a,b,c}, right padded with a blank, and
▷   a function NEXTCHAR(string) which returns the next symbol in 'string', procedure
▷   InP returns true if 'string' is in P and false, otherwise.
2   call InitStack( $\mathbb{S}$ )
3   char  $\leftarrow$  NEXTCHAR(string)
4   while char  $\neq$  ' ' do
5       if char = ' ' then return(false)
6       else [ call PUSH( $\mathbb{S}$ , char); char  $\leftarrow$  NEXTCHAR(string) ]
7   endwhile
8   while not IsEmptyStack( $\mathbb{S}$ ) do
9       char  $\leftarrow$  NEXTCHAR(string)
10      call POP( $\mathbb{S}$ , x)
11      if char  $\neq$  x then return(false)
12  endwhile
13  if NEXTCHAR(string)  $\neq$  ' ' then return(false)
14      else return(true)
15  end InP

```

Procedure 4.9 Recognizing simple patterns

4.3.2 Conversion of arithmetic expressions from infix to postfix form

This is an old problem in Computer Science. The problem has its roots in the fact that the form in which we have been taught to write arithmetic expressions since our preschool days (which is infix notation) is not the form suitable for the efficient evaluation of an expression by a computer (which is postfix notation). Thus the need to convert from infix to postfix notation.

To put the problem in proper perspective, let us define what we mean by the infix and postfix forms of an expression in terms of the way in which the expression is evaluated.

1. An expression is in infix notation if every subexpression to be evaluated is of the form operand-operator-operand. The subexpressions may or may not be enclosed in parentheses. Their order of evaluation depends on specified operator precedence and on the presence of parentheses with the expression normally scanned repeatedly from left to right during its evaluation.
2. An expression is in postfix notation if every subexpression to be evaluated is of the form operand-operand-operator. Subexpressions are never parenthesized. The expression is scanned only once from left to right and subexpressions of the form operand-operand-operator are evaluated in the order in which they are encountered. In postfix notation, operator precedence is no longer relevant.

We will assume, for ease of exposition, that the arithmetic expressions to be converted involve the five *binary* operators $+$, $-$, $*$, $/$ and $^$ only. The table below lists these operators in decreasing order of priority.

Operator	Priority	Property	Example
$^$	3	Right associative	$a \wedge b \wedge c \equiv a \wedge (b \wedge c)$
$*$ $/$	2	Left associative	$a / b / c \equiv (a / b) / c$
$+$ $-$	1	Left associative	$a - b - c \equiv (a - b) - c$

Figure 4.4 Order of precedence of the five binary arithmetic operators

Shown below are examples of arithmetic expressions in infix notation and the corresponding postfix expressions, indicating the order of evaluation of subexpressions in both notations. It is clear that it takes several left-to-right passes to evaluate the infix form of the expression, and only one left-to-right pass to evaluate the corresponding postfix form. In the latter case, operands are pushed onto a stack as they are encountered in the input postfix expression. Each time an operator is encountered in the input, the stack is popped two times to retrieve (and delete from the stack) its two operands; the indicated operation is performed and the result is pushed onto the stack. Then the left-to-right scan of the postfix expression continues. Simulate this process on any of the given postfix expressions and verify that indeed one pass suffices to evaluate the expression.

Infix expression	Postfix expression
1. $a + b - c/d + e \wedge f \wedge 2$ 	$a \ b \ + \ c \ d \ / \ - \ e \ f \ 2 \ \wedge \ \wedge \ +$
2. $a * (b + c/d) / e$ 	$a \ b \ c \ d \ / \ + \ * \ e \ /$
3. $a + (b \wedge (c + d - e) \wedge f) + g * h$ 	$a \ b \ c \ d \ + \ e \ - \ f \ \wedge \ \wedge \ + \ g \ h \ * \ +$

Figure 4.5 Evaluating infix and postfix expressions

To help us formulate an algorithm to convert an expression from infix to postfix notation, we take note of the following observations about the examples given above.

1. The postfix form contains no parentheses.
2. The order of the operands in both forms is the same, whether or not parentheses are present in the infix expression.
3. The order of the operators is altered in the postfix form. The reordering of the operators is a function of operator precedence and the presence of parentheses in the infix expression.

Formulating an algorithm to convert an expression from infix to postfix form

The conversion algorithm takes as input an infix expression and generates as output the corresponding postfix expression; we assume, without loss of generality, that both are strings. Also, for reasons that will become clear shortly, we require that the infix expression be enclosed in parentheses. Now imagine scanning the input infix string from left to right. We encounter four types of tokens: an operand, an operator, an opening parenthesis and a closing parenthesis. How is each one processed?

Figure 4.6 (see next page) gives a step by step account of the process by which we obtained the postfix form of the first of the three infix expressions in Figure 4.5, along with an explanation for the actions taken. Needless to say, the basis for these actions are simply the rules of arithmetic.

The input infix string is $(a + b - c/d + e^f^2)$; the output postfix string is initially null. We use a stack as temporary storage for operators and the '('; the stack is initially empty. We denote by $Pr(x)$ the precedence of operator x as given in Figure 4.4. (Study carefully the example given in Figure 4.6 before reading on.)

It is clear from Figure 4.6 that the crucial part of the conversion process is the handling of operators. We note that *all* operators are first stored in a stack; the question each time is 'Can we now push the in-coming operator onto the stack, or do we pop the stack first to delete the operator at the top of the stack before pushing the new one?' The answer lies in the order of precedence and the associativity properties of the operators as indicated in Figure 4.4. The actions we took in steps 5, 7, 9, 11 and 13, for the reasons given, can be cast in the form of rules as follows:

- (a) An in-coming operator with a lower precedence is never placed directly on top of an operator with a higher precedence; the latter must first be popped (and appended to the output). Conversely, an in-coming operator with a higher precedence is immediately pushed on top of an operator with a lower precedence.
- (b) When the in-coming operator and the operator at the top of the stack (which may be the same or a different operator) have the same precedence, the former is immediately pushed onto the stack if it is right associative; otherwise, the operator at the top of the stack is first popped (and appended to the output).

When the infix expression contains parenthesized subexpressions, we apply the same rules for each parenthesized subexpression. This is the reason why we require that the

Given infix expression: $(a + b - c/d + e^f 2)$

Step	Token	Remarks / Reason
1.	(Push '(' onto initially empty stack (stack is now '('). The '(' remains in the stack until the matching ')' is encountered.
2.	a	Append 'a' to initially null postfix string (string is now 'a'). An operand is immediately appended to the postfix string; this is because the order of the operands is the same in both infix and postfix forms of an expression.
3.	+	Push '+' onto stack (stack is now '('+'). An operator is not immediately appended to the postfix string; this is because (a) in postfix notation an operator appears after its operands, and (b) the operator may have a lower precedence than another operator that comes after it in the infix string
4.	b	Append 'b' to postfix string (string is now 'a b').
5.	-	Since $Pr(-) = Pr(+)$ and both operators are <i>left associative</i> , the addition is performed before the subtraction. Hence, we pop the stack to delete the '+' and then append it to the postfix string (string is now 'a b +'). Next we push the '-' onto the stack (stack is now '('-').
6.	c	Append c to postfix string (string is now 'a b + c').
7.	/	Since $Pr(/) > Pr(-)$ we push the '/' onto the stack; thus when these operators are unstacked and appended to the postfix string, the '/' will precede the '-' (stack is now '('-/').
8.	d	Append d to postfix string (string is now 'a b + c d').
9.	+	Since $Pr(+)<Pr(/)$, we unstack the '/' and append it to the postfix string. The top element of the stack is now '-'. Since $Pr(-) = Pr(+)$ and both operators are left associative, we unstack the '-' and append it to the postfix string (string is now 'a b + c d / -'). Next we push the '+' onto the stack (stack is now '('+').
10.	e	Append 'e' to postfix string (string is now 'a b + c d / - e').
11.	^	Since $Pr(^)>Pr(+)$ we push '^' onto the stack (stack is now '('+^^').
12.	f	Append 'f' to postfix string (string is now 'a b + c d / - e f').
13.	^	The token and the top element of the stack are both '^'; since '^' is <i>right associative</i> we push the '^' onto the stack (stack is now '('+^^^'). When the operators are unstacked and appended to the postfix string, the second '^' will precede the first '^'.
14.	2	Append '2' to postfix string (string is now 'a b + c d / - e f 2').
15.)	We pop the stack and append the deleted elements to the postfix string until the matching '(' becomes the top element (postfix string is now 'a b + c d / - e f 2 ^ ^ +'). Next we pop the stack to delete the '('; stack is now empty. Conversion is complete.

Figure 4.6 Converting an arithmetic expression from infix to postfix form

entire expression be enclosed in parentheses; the enclosing parentheses are simply treated as the outermost pair. The actions we took in steps 1, 3 and 15 in the above example can be cast in the form of the following additional rules:

- (c) An opening parenthesis is immediately pushed onto the stack (regardless of what is at the top of the stack).

- (d) An operator is immediately pushed onto the stack if the element at the top of the stack is an opening parenthesis.
- (e) When the in-coming token is a closing parenthesis, the stack is popped (and the deleted operators appended to the output) until the matching opening parenthesis becomes the top element. Then the stack is popped once more to delete the opening parenthesis. If at this point the stack is empty, then the conversion is done; otherwise, the processing continues.

The final rule applies to operands. For reasons already given (see step 2), it is

- (f) An operand immediately goes to the output.

Rules (a) through (d) are easily implemented by assigning two priority numbers to tokens (operators and the opening parenthesis) which enter the stack — an **in-coming priority**, say $icp()$ and an **in-stack priority**, say $isp()$ — as follows:

- (a) Let x and y be operators such that x has a higher precedence than y ; then we set $icp(x) > isp(y)$. For example: $icp('^') > isp('/')$, $icp('*') > isp('-')$.
- (b) Let x and y be operators with the same precedence (x may be the same as y); then we set $icp(x) > isp(y)$ if x and y are right associative, and we set $icp(x) < isp(y)$ if x and y are left associative. For example: $icp('^') > isp('^')$, $icp('+') < isp('-')$, $icp('*') < isp('*')$.
- (c) Let x be the operator with the lowest icp ; then we set $isp('(') < icp(x)$.

With this definition of the quantities $icp()$ and $isp()$, all the rules pertaining to tokens which are stored in the stack can be summarized into just one general prescription:

Let x be the in-coming token and let y be the token at the top of the stack. If $icp(x) < isp(y)$ pop the stack (and append the deleted operator to the output) and test again. Eventually, there will be a new top element y such that $icp(x) > isp(y)$; then push x onto the stack.

Figure 4.7 shows an assignment of $icp()$ and $isp()$ values for the five binary arithmetic operators according to the rules stated above. (Ignore for the moment the last column.)

Token, x	$icp(x)$	$isp(x)$	$rank$
operand	—	—	+1
+ or −	1	2	−1
* or /	3	4	−1
^	6	5	−1
(—	0	—

Figure 4.7 In-coming and in-stack priorities

Putting together all of the above we obtain the following algorithm. (Note: Postfix notation is also called ‘reverse Polish’ notation after the Polish logician Jan Lukasiewicz.)

Algorithm POLISH. Given an infix expression on the five binary arithmetic operators $+$, $-$, $*$, $/$ and $^$, POLISH generates the corresponding postfix form of the expression.

1. Get next token, say x .
2. If x is an operand, then output x
3. If x is the '(', then push x onto the stack.
4. If x is the ')', then output stack elements until an '(' is encountered. Pop stack once more to delete the '('. If now the stack is empty, the algorithm terminates.
5. If x is an operator, then while $icp(x) < isp(stack(top))$, output stack elements; else, if $icp(x) > isp(stack(top))$, then push x onto the stack.
6. Go to step 1.

Example 4.1. Convert $(A*(B+(C-D)/E^2)+F)$ to postfix form. (Note that the whole expression is parenthesized.)

In-coming symbol	Stack elements	Output	Remarks
((Push (onto stack
A	(A	Output operand
*	(*	A	$icp[*] > isp[(]$
((*(A	
B	(*(AB	
+	(*(+	AB	$icp[+] > isp[(]$
((*(+(AB	
C	(*(+(ABC	
-	(*(+(-	ABC	$icp[-] > isp[(]$
D	(*(+(-	ABCD	
)	(*(+	ABCD-	Pop stack until top element is (; pop again to delete (
/	(*(+/ E	ABCD-	$icp[/] > isp[+]$
^	(*(+/ 2	ABCD-E	$icp[^] > isp[*]$
)	(*(+/ +)	ABCD-E2	Pop stack until ...
+	(*(+/ +)	ABCD-E2^/+*	$icp[+] < isp[*]$
+	(*(+/ +)	ABCD-E2^/+*	$icp[+] > isp[(]$
F	(*(+/ +)	ABCD-E2^/+*F	
)		ABCD-E2^/+*F+	Pop stack until ... (top = 0, stop)

Figure 4.8 Algorithm POLISH in action

Determining whether a postfix expression is well-formed

As we have earlier indicated, the reason why we convert an infix expression to postfix form is so we can mechanically *evaluate* the expression in a most efficient way, that is, in a way which requires only one left-to-right pass through the expression and in which operator precedence no longer matters. Before we evaluate a postfix expression, it is appropriate to first determine whether it is *well-formed*. A well-formed postfix expression can be evaluated (see Figure 4.5); an expression that is not well-formed cannot be evaluated.

The following theorem indicates when a postfix expression is well-formed, but first let us define a few terms.

1. The degree of an operator is its number of operands.
2. The rank of an operand is 1. The rank of an operator is 1 minus its degree (see Figure 4.7). The rank of an arbitrary sequence of operands and operators is the sum of the ranks of the individual operands and operators.
3. Let ‘|’ denote the concatenation operator. If $z = x|y$ is a string, then x is the head of z . x is a proper head if y is not the null string.

Now the theorem:

THEOREM 4.1. A postfix expression is well-formed if and only if

- (a) the rank of every proper head is greater than or equal to 1, and
- (b) the rank of the entire expression is equal to 1

Example 4.2. For the generated postfix expression $ABCD - E2^{\wedge} / + *F +$ in Figure 4.8 the ranks of successive proper heads are 1, 2, 3, 4, 3, 4, 5, 3, 2, 1 and 2 and the rank of the entire expression is 1. The expression is well-formed and can be evaluated (verify).

Example 4.3. The infix expression $(AB++C)$ converts to the postfix expression $AB+C+$ (verify). The ranks of successive proper heads are 1, 2, 1 and 2 and the rank of the whole expression is 1; by theorem 4.1, the postfix expression is well-formed and it can be evaluated. For instance, if $A = 5$, $B = -3$ and $C = 7$, then $AB + C +$ evaluates to 9. However, the infix expression is clearly wrong. This example shows that a well-formed postfix expression, as generated by algorithm POLISH, is *not* a guarantee that the original infix expression is valid.

EASY implementation of algorithm POLISH

The EASY procedure POLISH implements algorithm POLISH to convert an arithmetic expression from infix to postfix form. It invokes procedures $\text{InitStack}(\mathbb{S})$, $\text{IsEmptyStack}(\mathbb{S})$, $\text{PUSH}(\mathbb{S}, x)$ and $\text{POP}(\mathbb{S}, x)$ of the previous sections. How the stack \mathbb{S} is implemented (array or linked list) is immaterial to POLISH.

POLISH invokes FINDRANK to determine the rank of the generated postfix expression. A value of 1 means the generated postfix expression is well-formed and can be evaluated; it does not necessarily mean that the original infix expression is correct.


```

1  procedure POLISH(infix, postfix, r)
  ▷ Given an input string 'infix' representing a parenthesized infix expression,
  ▷ POLISH generates the corresponding postfix expression in the string 'postfix'.
  ▷ POLISH uses the following procedures (in addition to the stack routines):
  ▷ EOS(string) – returns true if end of string is reached; else, false
  ▷ NEXTTOKEN(string) – returns next token in string
  ▷ IsOpnd(x) – returns true if x is an operand; else false
  ▷ ICP(x) – returns in-coming priority of token x
  ▷ ISP(x) – returns in-stack priority of token x
  ▷ RANK(x) – returns rank of token x
  ▷ POLISH returns the rank of the generated postfix expression in the parameter r.
2  call InitStack( $\mathbb{S}$ )
3  postfix  $\leftarrow$  "      ▷ null string
4  while not EOS(infix) do
5      x  $\leftarrow$  NEXTTOKEN(infix)
6      case
7          : IsOpnd(x) : postfix  $\leftarrow$  postfix | x      ▷ concatenation
8          : x = '(' : call PUSH( $\mathbb{S}$ , x)
9          : x = ')' : [ while not IsEmptyStack( $\mathbb{S}$ ) do
10                      call POP( $\mathbb{S}$ , xtop)
11                      if xtop  $\neq$  '(' then postfix  $\leftarrow$  postfix | xtop
12                      else if IsEmptyStack( $\mathbb{S}$ ) then
13                          [ call FINDRANK(postfix, r); return ]
14                      endwhile ]
15          : else : [ while not IsEmptyStack( $\mathbb{S}$ ) do
16                      call POP( $\mathbb{S}$ , xtop)
17                      if ICP(x) < ISP(xtop) then postfix  $\leftarrow$  postfix | xtop
18                      else [ call PUSH( $\mathbb{S}$ , xtop); call PUSH( $\mathbb{S}$ , x) ]
19                      endwhile ]
20      endcase
21  endwhile
22  r  $\leftarrow$  0
23  end POLISH

1  procedure FINDRANK(postfix, r)
2  r  $\leftarrow$  0
3  while not EOS(postfix) do
4      x  $\leftarrow$  NEXTTOKEN(postfix)
5      r  $\leftarrow$  r + RANK(x)
6      if r < 1 then return
7  endwhile
8  end FINDRANK

```

Procedure 4.10 Converting an infix expression to postfix form

Procedure POLISH can be extended to convert expressions involving other operators, e.g., unary arithmetic operators, relational operators, logical operators, and so on by expanding Figure 4.7 to include the *icp*() and *isp*() values for these operators according to specified priorities among the operators.

4.4 Sequential implementation of multiple stacks

It is not uncommon to have applications which require the concurrent use of two or more stacks. One way to implement multiple stacks in sequential memory is to allocate one array for each stack as shown in Figure 4.9. Each stack is a data structure unto itself, with its own components: $S1 = [S1(1 : n1), top1]$, $S2 = [S2(1 : n2), top2]$, With this implementation, procedures PUSH and POP of Section 4.1 can be used without modification. For example, to push an element, say x , onto stack 1 we simply write

call PUSH($S1, x$)

and to retrieve into y the top element of stack 3 we simply write

call POP($S3, y$)

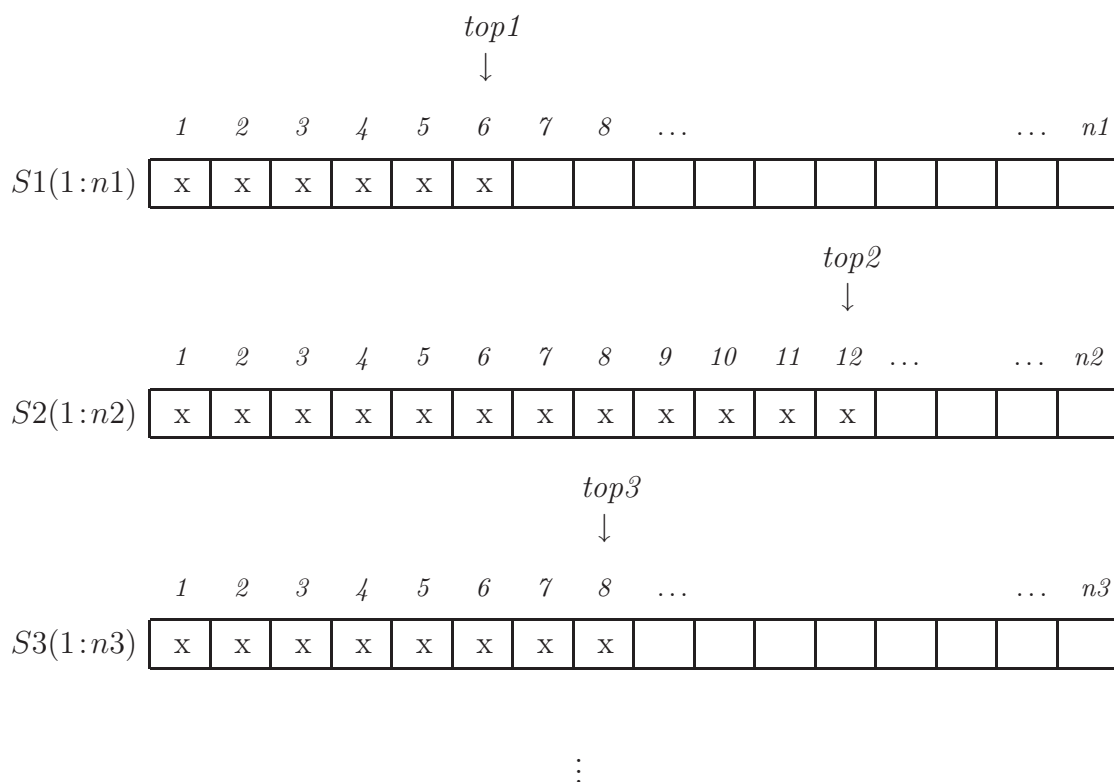


Figure 4.9 Implementation of multiple stacks in separate arrays

The problem with the above implementation is that if a particular stack overflows it cannot make use of free space that has been allocated to the other stacks (*kanya-kanya, walang bigayan*). A more efficient utilization of preallocated memory can be realized if we define only one array, say S of size n , and let all the stacks, say m of them, share in the use of S . In particular, this makes it possible to systematically reallocate available memory when overflow occurs in any one of the stacks.

Two cases are of interest: the case where $m = 2$ and the case where $m > 2$. The first case is easily handled; the second case requires a little more planning.

4.4.1 The coexistence of two stacks in a single array

The figure below shows two stacks coexisting in a common array S of size n . The stacks are arranged such that they grow toward each other, with their bottoms anchored at opposite ends of S . It is immediately clear that both stacks are full only if *all* n cells are in use; an attempt to insert onto any of the stacks then results in an overflow condition.

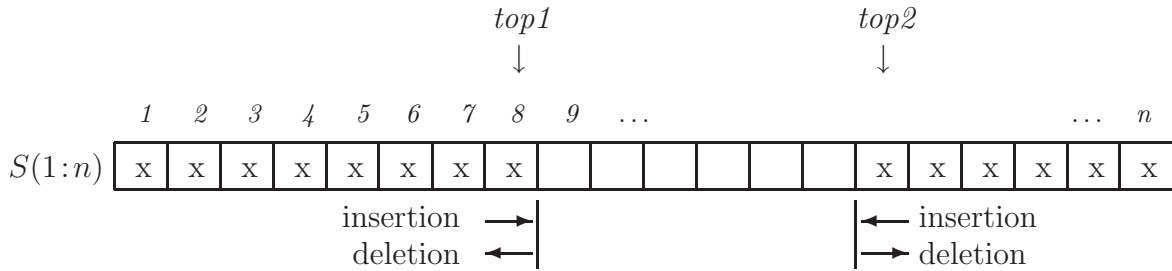


Figure 4.10 Coexistence of two stacks in a single array S

We initialize the stacks by setting $top1 \leftarrow 0$ and $top2 \leftarrow n + 1$. Consequently, the condition $top1 = 0$ means stack 1 is empty and the condition $top2 = n + 1$ means stack 2 is empty. The condition $top2 - top1 = 1$ means stacks 1 and 2 are full and an attempt to insert onto either stack will result in an overflow condition. It is obvious that when overflow occurs all cells $S(1)$ through $S(n)$ are in use. Thus the space allocated to S is utilized in an optimal way. The procedures for insertion and deletion are easily coded; these are left as exercises.

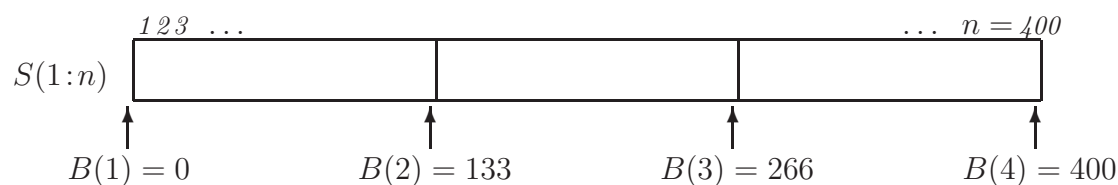
4.4.2 The coexistence of three or more stacks in a single array

When three or more stacks coexist in a common array S , it no longer follows that when overflow occurs in a particular stack that all n cells of S are in use (as it does in the case of two stacks coexisting in S). No matter how we initially allocate the available space among the stacks and no matter in what directions we make the stacks to grow, it may still happen that one stack runs out of space while others still have space to spare. To make optimal use of the space allocated to S we should allow for the free cells to be *reallocated* among the stacks so that, at the very least, the stack which overflows will get the space it needs. This can be accomplished by shifting stack boundaries.

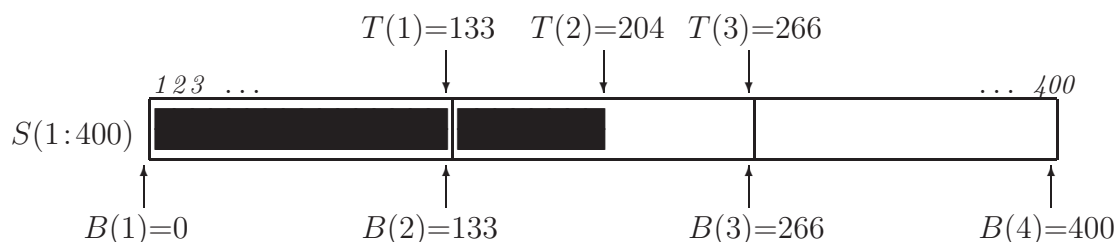
Figure 4.11 shows an array S of size $n = 400$ in which $m = 3$ stacks are to coexist. The 400 cells are initially allocated equally among the 3 stacks by defining the base pointers $B(i)$ according to the formulas

$$\begin{aligned} B(i) &= \lfloor n/m \rfloor (i - 1) & 1 \leq i \leq m \\ B(m + 1) &= n \end{aligned} \tag{4.1}$$

The base pointers indicate the boundaries, in terms of allocated space, among the stacks. Stack 1 is allocated cells 1 thru 133, stack 2 is allocated cells 134 thru 266 and stack 3 is allocated cells 267 thru 400. Thus we see from the figure that stacks 1 and 2 are allocated 133 cells each and stack 3 gets 134 cells.

**Figure 4.11** Establishing initial stack boundaries

To initialize the stacks, we set $T(i) \leftarrow B(i)$, $1 \leq i \leq m$. Thus the condition $T(i) = B(i)$ means that stack i is empty. You may easily verify that the condition $T(i) = B(i + 1)$ indicates that stack i is full. The figure below shows the three possible states of a stack: stack 1 is full, stack 2 has free cells and stack 3 is empty.

**Figure 4.12** Coexistence of three stacks in a single array S

An attempt to insert onto a full stack, such as stack 1 in the figure above, results in an overflow condition; an attempt to delete from an empty stack, such as stack 3 in the figure, results in an underflow condition. Procedures to implement the insert and delete operations must test for the occurrence of these conditions. Specifically, when an overflow condition obtains, memory should be reallocated to make room for the insertion.

We will use the notation $\text{MS} = [S(1:n), T(1:m), B(1:m+1), m, n]$ to denote the data structure consisting of an array S of size n in which m stacks coexist (such as the one shown in Figure 4.12). The EASY procedures MPUSH and MPOP implement the insert and delete operations for the data structure MS . They are essentially the same as the corresponding procedures for a sequentially allocated single stack.

```

1  procedure MPUSH( $\text{MS}, i, x$ )
2  if  $T(i) = B(i + 1)$  then call MSTACKOVERFLOW( $\text{MS}, i$ )
3   $T(i) \leftarrow T(i) + 1$ 
4   $S(T(i)) \leftarrow x$ 
5  end MPUSH

```

Procedure 4.11 Insertion onto stack i in MS

If stack i is full (line 2), procedure MSTACKOVERFLOW will attempt to reallocate memory to make insertion onto stack i possible. If the attempt is successful, MSTACKOVERFLOW returns control to MPUSH via a **return** statement for completion of the insert operation (lines 3–4). If reallocation is unsuccessful (because all n cells are in use), MSTACKOVERFLOW returns control to the operating system via a **stop** statement.

```

1  procedure MPOP(MS, i, x)
2  if  $T(i) = B(i)$  then call MSTACKUNDERFLOW
3      else  $[x \leftarrow S(T(i)); T(i) \leftarrow T(i) - 1]$ 
4  end MPOP

```

Procedure 4.12 Deletion from stack i in MS

Line 2 tests for underflow in stack i and calls MSTACKUNDERFLOW if such a condition obtains. Otherwise, the top element of stack i is retrieved into x and deleted from the stack (line 3).

4.4.3 Reallocating memory at stack overflow

We will now consider the problem of reallocating memory when a particular stack, say stack i , overflows. One simple procedure to obtain a free cell for stack i is to scan the stacks above stack i , *addresswise*, for the nearest stack with available cells, say stack k . Shifting stack $i + 1$ through stack k one cell upwards will make one cell available to stack i . The following segment of EASY code will perform this task (verify):

```

▷ Shift stack elements one cell upward
  for  $j \leftarrow T(k)$  to  $T(i) + 1$  by  $-1$  do
     $S(j + 1) \leftarrow S(j)$ 
  endfor
▷ Adjust top and base pointers
  for  $j \leftarrow i + 1$  to  $k$  do
     $T(j) \leftarrow T(j) + 1$ 
     $B(j) \leftarrow B(j) + 1$ 
  endfor

```

If all the stacks above stack i are full, then we scan the stacks below for the nearest stack with free space, say stack k . Shifting stacks $k + 1$ through i one cell downward will make one cell available to stack i . The following segment of EASY code will perform this task (verify).

```

▷ Shift stack elements one cell downward
  for  $j \leftarrow B(k + 1)$  to  $T(i)$  do
     $S(j - 1) \leftarrow S(j)$ 
  endfor
▷ Adjust top and base pointers
  for  $j \leftarrow k + 1$  to  $i$  do
     $T(j) \leftarrow T(j) - 1$ 
     $B(j) \leftarrow B(j) - 1$ 
  endfor

```

If all the stacks below stack i are also full, then all cells are in use and reallocation is not possible. `MSTACKOVERFLOW` may then simply issue a message to this effect and terminate execution of the program.

The problem with this *unit-shift technique* for obtaining a free cell for an overflowed stack is that successive insertions onto such a stack will require repetitive calls to `MSTACKOVERFLOW`. An alternative approach is to reallocate *all* of free memory at overflow such that each stack will receive its share of free cells in proportion to its *need* for space, where need is determined by some appropriate measure. This is the idea behind *Garwick's algorithm*.

Garwick's technique for reallocating memory at stack overflow may be summarized as follows:

1. Strip all the stacks of unused cells and consider all of these unused cells as comprising the available or free space.
2. Reallocate one to ten percent of the available space equally among the stacks.
3. Reallocate the remaining available space among the stacks in proportion to need for space as measured by *recent growth*.

For any stack j , recent growth is measured as the difference $T(j) - OLD T(j)$, where $OLD T(j)$ is the value of $T(j)$ at the end of the last reallocation. It is obvious that a negative difference means that stack j actually decreased in size since last reallocation and a positive difference means that stack j increased in size since last reallocation. The bigger the (positive) difference is, the bigger will be the stack's share of the available memory.

Knuth's implementation of Garwick's algorithm fixes at 10% of the available space the portion to be distributed equally among the stacks, with the remaining 90% apportioned according to recent growth. Standish suggests that in addition to recent growth, stack size (or cumulative growth) can also be used as a measure of need in distributing the remaining 90%. The idea is the bigger the stack the more space it needs.

We now consider an example to illustrate Garwick's algorithm before we implement it as procedure `MSTACKOVERFLOW`. Figure 4.13 shows four stacks coexisting in an array of size 400. An attempt to insert an element onto stack 2 using `MPUSH` results in an overflow condition and memory is reallocated using Garwick's technique.

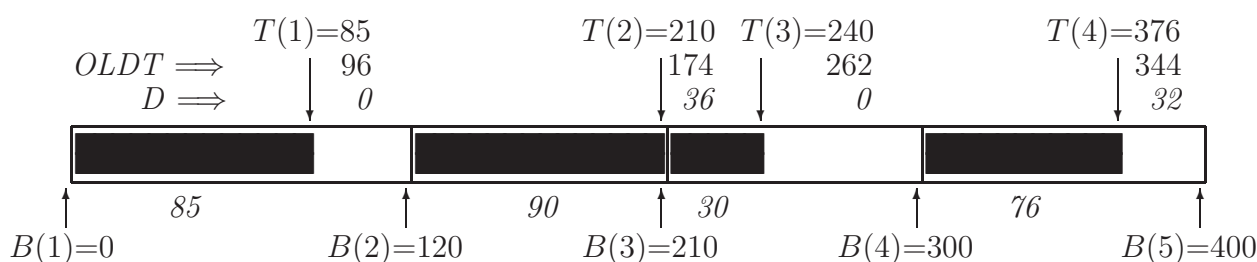


Figure 4.13 Configuration of the stacks at overflow condition

Example 4.4. Garwick's algorithm

1. [*Gather statistics on stack usage.*] The stack sizes $T(j) - B(j)$ and the differences $T(j) - OLDT(j)$ are calculated and indicated Figure 4.13. Note that a negative difference is replaced by zero.

$$\begin{aligned} freecells &= 400 - (85 + 90 + 1 + 30 + 76) = 118 \\ incr &= 36 + 1 + 32 = 69 \end{aligned}$$

The +1 in the above computations accounts for the cell that the overflowed stack is currently in need of. This cell is assumed to be already given to the overflowed stack and already in use.

2. [*Calculate allocation factors.*]

$$\begin{aligned} a &= (0.10 \times 118)/4 = 2.95 \\ b &= (0.90 \times 118)/69 = 1.54 \end{aligned}$$

The quantity a represents the number of cells that each stack will get from the 10% of available space allotted for equal distribution. The quantity b represents the number of cells that a stack will get per unit increase in stack usage from the remaining 90% of free space. For example, stack 2 will get, theoretically, $2.95 + 37 \times 1.54$ cells in all, but stack 1 will get 2.95 cells only. We say 'theoretically' because a stack may get an integral number of cells only; how the fractional portion is handled is described in step 3.

3. [*Compute new base addresses.*] Memory reallocation involves essentially determining the new stack boundaries, $NEWB(j)$, $j = 2, 3, \dots, m$. These new boundaries are found by allocating to each stack the number of cells it is currently using plus its share of the free cells as measured by the parameters a and b . An important aspect of this step is to see to it that the fractional portions are evenly distributed among the stacks. To this end, we define two real variables s and t , as follows:

$$\begin{aligned} s &= \text{free space theoretically allocated to stacks } 1, 2, \dots, j-1 \\ t &= \text{free space theoretically allocated to stacks } 1, 2, \dots, j \end{aligned}$$

Then, we calculate the actual number of (whole) free cells allocated to stack j as the difference

$$\lfloor t \rfloor - \lfloor s \rfloor$$

The following computations should help clarify the procedure.

$$\begin{aligned} \text{Stack 1: } NEWB(1) &= 0; \quad s = 0 \\ \text{Stack 2: } t &= 0 + 2.95 + 0(1.54) = 2.95 \\ NEWB(2) &= 0 + 85 + \lfloor 2.95 \rfloor - \lfloor 0 \rfloor = 87; \quad s = 2.95 \\ \text{Stack 3: } t &= 2.95 + 2.95 + (36 + 1)(1.54) = 62.88 \\ NEWB(3) &= 87 + 91 + \lfloor 62.88 \rfloor - \lfloor 2.95 \rfloor = 238; \quad s = 62.88 \\ \text{Stack 4: } t &= 62.88 + 2.95 + 0(1.54) = 65.83 \\ NEWB(4) &= 238 + 30 + \lfloor 65.83 \rfloor - \lfloor 62.88 \rfloor = 271 \end{aligned}$$

As a check, we calculate $NEWB(5)$ to see if it is 400.

$$t = 65.83 + 2.95 + 32(1.54) = 118.06$$

$$NEWB(5) = 271 + 76 + \lfloor 118.06 \rfloor - \lfloor 65.83 \rfloor = 400 \quad (\text{O.K.})$$

All nodes are accounted for.

4. [*Shift stacks to their new boundaries.*] Careful consideration should be given to this step. It is important that no data item is lost (for instance, overwritten) when the stacks are moved. To this end, stacks which are to be shifted down are processed from left to right (i.e., by increasing stack number), while stacks which are to be shifted up are processed from right to left. The same method applies to the individual data items. Data items that are moved down are processed in the order of increasing addresses (indices), and data items that are moved up are processed in the order of decreasing addresses. In this manner, no useful data is overwritten. (Verify.)

Figure 4.14 shows the stacks upon termination of Garwick's algorithm. Note that stacks 1 and 3 now have fewer free cells, while stacks 2 and 4 have more free cells. This is because stacks 1 and 3 exhibited negative growth, while stacks 2 and 4 exhibited positive growth, since memory was last reallocated. There is now more than enough room in stack 2 for the insert operation to proceed. This current state of the stacks is recorded in the updated arrays T and B ; note that $OLDT$ has also been updated in preparation for the next memory reallocation.

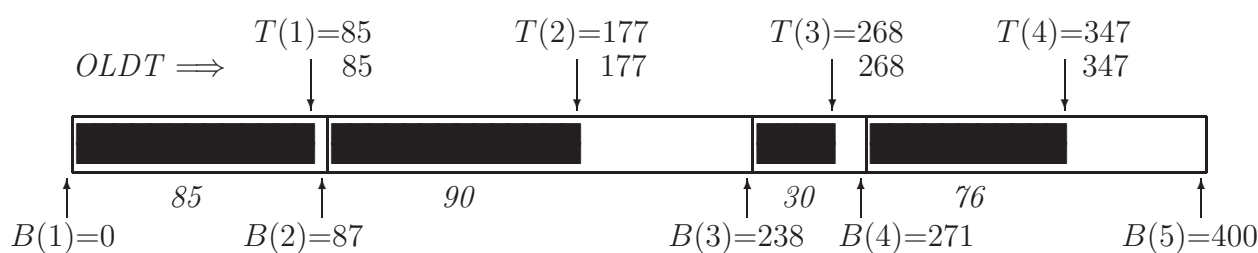


Figure 4.14 Configuration of the stacks after memory reallocation

EASY implementation of Garwick's algorithm

Procedure **MSTACKOVERFLOW** follows Knuth's implementation of Garwick's algorithm. Incorporating Standish's suggestion that stack size be also used as a reallocation parameter is straightforward and is left as an exercise.

The input to **MSTACKOVERFLOW** is a pointer to the data structure $MS = [S(1:n), T(1:m), B(1:m+1), OLD T(1:m), m, n]$ and the tag i of the stack which overflowed. Note that we have added $OLD T$ to the components of MS . All the changes brought about by **MSTACKOVERFLOW** to these components of MS are of course seen by the calling procedure **MPUSH**.


```

1  procedure MSTACKOVERFLOW(MS, i)
2  array NEWB(1:m), D(1:m)
3   $\triangleright$  Gather relevant information on stack usage
4  freecells  $\leftarrow$  n; incr  $\leftarrow$  0
5   $T(i) \leftarrow T(i) + 1$   $\triangleright$  preempt one free cell for stack i
6  for j  $\leftarrow$  1 to m do  $\triangleright$  find recent growth for each stack
7      freecells  $\leftarrow$  freecells - (T(j) - B(j))
8      if T(j) > OLD T(j) then [D(j)  $\leftarrow$  T(j) - OLD T(j); incr  $\leftarrow$  incr + D(j)]
9      else D(j)  $\leftarrow$  0
10 endfor
11 if freecells < 0 then [output 'No more available space'; stop]
12  $\triangleright$  Calculate allocation factors
13 a  $\leftarrow$  (0.10  $\times$  freecells)/m
14 b  $\leftarrow$  (0.90  $\times$  freecells)/incr
15  $\triangleright$  Compute new base addresses
16 NEWB(1)  $\leftarrow$  B(1); s  $\leftarrow$  0
17 for j  $\leftarrow$  2 to m do
18     t  $\leftarrow$  s + a + b  $\times$  D(j - 1)
19     NEWB(j)  $\leftarrow$  NEWB(j - 1) + T(j - 1) - B(j - 1) +  $\lfloor t \rfloor - \lfloor s \rfloor$ 
20     s  $\leftarrow$  t
21 endfor
22  $\triangleright$  Revert top pointer of stack i to true value
23 T(i)  $\leftarrow$  T(i) - 1
24  $\triangleright$  Shift stacks whose elements are to be moved down
25 for j  $\leftarrow$  2 to m do
26     if NEWB(j) < B(j) then [d  $\leftarrow$  B(j) - NEWB(j)
27     for k  $\leftarrow$  B(j) + 1 to T(j) do
28         S(k - d)  $\leftarrow$  S(k)
29     endfor
30     B(j)  $\leftarrow$  NEWB(j); T(j)  $\leftarrow$  T(j) - d]
31 endfor
32  $\triangleright$  Shift stacks whose elements are to be moved up
33 for j  $\leftarrow$  m to 2 by -1 do
34     if NEWB(j) > B(j) then [d  $\leftarrow$  NEWB(j) - B(j)
35     for k  $\leftarrow$  T(j) to B(j) + 1 by -1 do
36         S(k + d)  $\leftarrow$  S(k)
37     endfor
38     B(j)  $\leftarrow$  NEWB(j); T(j)  $\leftarrow$  T(j) + d]
39 endfor
40  $\triangleright$  Update OLD T array in preparation for next memory reallocation
41 OLD T  $\leftarrow$  T
42 end MSTACKOVERFLOW

```

Procedure 4.13 Garwick's algorithm

A few comments about Garwick's technique are in order at this point (see KNUTH1[1997], pp. 250–251).

1. If we know beforehand which stack will be largest, then we should make this the first stack since stack 1 is never moved during memory reallocation. Likewise, we can initially allocate more space to this stack, rather than giving equal space to every stack.
2. When a program gets close to utilizing all of the preallocated space in S , it will call `MSTACKOVERFLOW` more often, and, to make matters worse, the procedure now takes more time to execute since more data items need to be shifted around. It is also quite likely that such a program will eventually need more space than what is available, causing `MSTACKOVERFLOW` to terminate execution anyway. To avoid such futile effort at memory reallocation as memory runs out, we can modify `MSTACKOVERFLOW` such that it issues the **stop** command when *freecells* becomes less than a specified minimum value, say *minfree*, which the user can specify.
3. While `MSTACKOVERFLOW` is written specifically for stacks, Garwick's technique can also be applied to reallocate space for other types of sequentially allocated data structures, such as queues and sequential tables, or even for combinations of these different types of structures coexisting in a fixed region of memory.

Summary

- The stack is an extraordinarily simple ADT. The *last-in first-out* discipline imposed on its elements allow constant time implementations of its two basic operations, viz., *push* and *pop*.
- Despite its simplicity, the stack is an extremely useful ADT. Some important applications are: (a) converting arithmetic expressions from infix to postfix form for efficient evaluation, (b) evaluating expressions in postfix form, and (c) resolving recursive calls in recursive programs.
- In applications which require the use of several stacks, making them coexist in and share a common area of contiguous memory results in efficient utilization of space, since this allows for memory to be reallocated when one stack overflows.
- Garwick's algorithm allows for memory reallocation in multiple stacks based on a stack's need for space as measured by *recent growth*. Another possible measure as suggested by Standish is *cumulative growth* or *stack size*.

Exercises

1. Find all the permutations of the integers 1, 2, 3 and 4 obtainable with a stack, assuming that they are pushed onto the stack in the given order.

2. Using a programming language of your choice, transcribe procedure InP into a running program. Test your program using the following strings: (a) *abbabcbabba* (b) *abbabbabba* (c) *abbabcbabb* (d) *abbaacbabba* (e) *abbabcbabbaa*

3. Using Algorithm POLISH, find the postfix form of the following infix expressions:

- (a) $A + B * C / (D * E)$ (d) $A + X * (B + X * (C + X * D))$
 (b) $B ^ D ^ 2 / C ^ E$ (e) $A / (B + C) ^ D * E / (F - G) * H$
 (c) $A * (B * (6 + D - E) / 8) - G ^ 3$ (f) $((A - B) * ((C + D) ^ 2 + F) / G ^ (1/2))$

4. Another algorithm for converting an infix expression to postfix form is (see HOROWITZ[1976], p. 95):

- (a) Fully parenthesize the infix expression.
 (b) Move each operator so that it replaces its corresponding right parenthesis.
 (c) Delete all left parentheses.

For example, the infix expression $A * B + C / D ^ 2 * E - F$ when fully parenthesized becomes

$$(((A * B) + ((C / (D ^ 2)) * E)) - F)$$

Each subexpression of the form operator-operand (for unary operators) or operand-operator-operand (for binary operators) is parenthesized. The arrows point from an operator to its corresponding right parenthesis in the parenthesized subexpression. If now we move the operators to where the arrows point and then delete all parentheses, we obtain the postfix form of the expression: $AB * CD2 ^ / E * + F -$.

Apply the above algorithm to the infix expressions in Item 3 and verify that you get exactly the same results generated by Algorithm POLISH.

5. What is one major disadvantage of the algorithm in Item 4?
6. Write an EASY procedure EVALUATE(*postfix*) to evaluate a postfix expression on the five binary operators $+$, $-$, $*$, $/$ and $^$.
7. Using a language of your choice, transcribe procedure POLISH (Procedure 4.10) into a running program. Test your program using the infix expressions in Item 3.
8. Using a language of your choice, transcribe procedure EVALUATE in Item 6 into a running program. Test your program using as input the output of your program in Item 7. Assume: $A = 5$, $B = -7$, $C = 8$, $D = 3$, $E = 4$, $F = -8$, $G = 6$, $H = 9$, $X = 10$.
9. Extend the table of *icp* and *isp* values given in Figure 4.7 to handle all the operators listed below.

\wedge , unary $-$, unary $+$, not	6	right associative
$*$, $/$	5	left associative
$+$, $-$	4	left associative
$<$, \nless , \leq , $=$, \neq , \geq , \nless , $>$	3	left associative
and	2	left associative
or	1	left associative

10. Write EASY procedures to perform the push and pop operations for two stacks coexisting in a single array as shown in Figure 4.10.
11. Five stacks coexist in a vector of size 600. The state of the computations is defined by $OLDT(1:4) = (170, 261, 360, 535)$, $T(1:4) = (140, 310, 400, 510)$ and $B(1:5) = (0, 181, 310, 450, 600)$ and an insertion operation is attempted onto stack 2.
 - (a) Draw a diagram showing the current state of the stacks.
 - (b) Perform Garwick's algorithm, as implemented in procedure `MSTACKOVERFLOW` (Procedure 4.13), to reallocate memory and draw a diagram showing the state of the stacks after reallocation is done.
12. Redo the computations in Item 11 but incorporate Standish's suggestion to use *cummulative growth* (or *stack size*), in addition to *recent growth*, as a measure of the need for space. Specifically, reallocate the free cells as follows: 10% equally among all the stacks, 45% according to recent growth, and 45% according to cummulative growth.
13. Modify procedure `MSTACKOVERFLOW` to incorporate Standish's suggestion as in Item 12.

Bibliographic Notes

The pattern recognition problem in Section 4.3 is adapted from TREMBLAY[1976], where the problem is one of determining whether a string belongs to a 'language' (the set $\{w c w^R\}$) which is generated by a particular 'grammar'. We have recast the problem in terms of palindromes, albeit of a restricted kind. The student may wish to extend the set P to include all palindromes (e.g., 'Able was I ere I saw Elba') and to generalize procedure `InP` accordingly.

The problem of converting infix expressions to postfix form is an old one in CS and a classical application of stacks. An extended treatment of the topic is found in TENENBAUM[1986], HOROWITZ[1976] and TREMBLAY[1976]. Theorem 4.1 is from the last cited reference.

Garwick's algorithm, implemented as procedure `MSTACKOVERFLOW` in Section 4.4 is given as Algorithm G (*Reallocate sequential tables*) and Algorithm R (*Relocate sequential tables*) in KNUTH1[1997], pp. 248–250, and as Algorithm 2.3 (*Reallocate space for sequential stacks*) and Algorithm 2.4 (*Move stacks to new location*) in STANDISH[1980], pp. 34–36. Knuth remarks that a theoretical analysis of Garwick's algorithm is difficult. Standish gives a thorough discussion of the algorithm and presents *simulation* results

which indicate how the algorithm behaved under various conditions of stack growth and memory utilization. Garwick's reallocation algorithm was devised at a time (the mid 60's of the last century) when computer memory was a scarce commodity. Cheap memory chips notwithstanding, the idea of allocating resources in proportion to actual need is still relevant today. (And not only among stacks.)

NOTES

SESSION 5

Queues and Deques

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Explain implementation issues pertaining to the sequential representation of a queue or a deque using a one-dimensional array.
2. Explain implementation issues pertaining to the linked representation of a queue or a deque using a linked linear list.
3. Discuss some important problems in Computer Science whose solution requires the use of a queue or a deque.
4. Assess the suitability of using a queue or a deque to solve a given problem on the computer.

READINGS KNUTH1[1997], pp. 238–245, 254–268; HOROWITZ[1976], pp. 77–86.

DISCUSSION

The queue, which is next of kin to the stack, is another simple but important data structure used in computer algorithms. Queues find applications in such diverse tasks as job scheduling in operating systems, computer simulations, tree and graph traversals, and so on. In this session, we will use a queue to solve the so-called *topological sorting problem*.

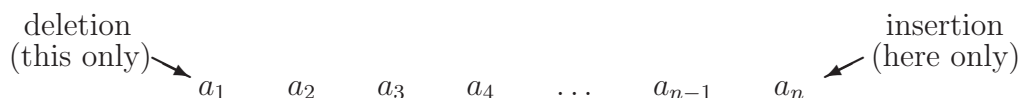


Figure 5.1 The queue ADT

In the abstract, we can think of a queue as a linearly ordered set of elements on which is imposed the discipline of ‘first-in, first-out’. As with the stack, the elements of a queue may be any entity such as a task awaiting execution, a node of a tree, an edge or vertex of a graph, etc.

112 SESSION 5. Queues and Deques

The two basic operations defined on a queue are

1. insert (or *enqueue*) an element at the *rear* of the queue
2. delete (or *dequeue*) the *front* element of the queue

Note that these are the operations which maintain the FIFO discipline on the elements of a queue. The other auxiliary operations defined on a queue are:

3. initialize the queue
4. test if the queue is empty
5. test if the queue is full

As with a stack, a queue in the abstract may be empty, but it may not be full. It should always be possible to insert yet one more element into a queue. The condition whereby a queue is full is a consequence of implementing the queue ADT on a machine with finite memory.

In certain applications, the elements of a queue may be reordered according to certain priorities assigned to the elements independently of their order of arrival. For instance, the rule may be ‘largest-in, first-out’ instead of ‘first-in, first-out’. Such a queue is called a *priority queue*. We will consider priority queues in Session 7.

As with the stack, there are two ways to implement the queue ADT as a concrete data structure, namely:

1. sequential implementation – the queue is stored in a one-dimensional array
2. linked implementation – the queue is represented as a linked linear list

There are two common variations on the sequential representation of a queue using a one-dimensional array. The second variation is commonly called a *circular queue*; for want of a name, let us call the first variation a *straight queue*.

5.1 Sequential implementation of a straight queue

Figure 5.2 shows the sequential representation of a queue in an array Q of size n . The elements marked ‘x’ comprise the queue. We adopt the convention whereby the variable *front* points to the cell immediately *preceding* the cell which contains the front element of the queue, and the variable *rear* points to the cell which contains the rear element of the queue. Be sure to keep the distinction between the queue (of size $rear - front$) and the array (of size n) in which the queue ‘resides’.

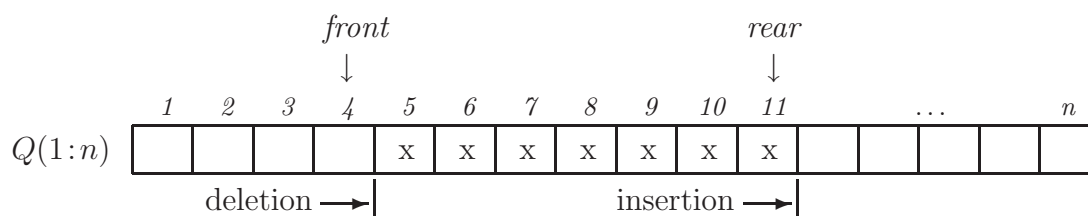


Figure 5.2 Sequential implementation of a straight queue

To insert an element, say x , into the queue we set

$$\begin{aligned} rear &\leftarrow rear + 1 \\ Q(rear) &\leftarrow x \end{aligned}$$

and to delete the front element from the queue and store it in x we set

$$\begin{aligned} front &\leftarrow front + 1 \\ x &\leftarrow Q(front) \end{aligned}$$

Note that as insertions and deletions are done, the queue moves rearward. If after an insertion $rear$ equals n , we get a *full queue*, and an insertion operation results in an *overflow condition* although there may still be empty cells in the forward portion of Q . If $front \neq 0$, moving the elements of the queue forward creates free space at the rear allowing the insertion to proceed (this is called *repacking* memory). An implementation of the insert operation for a sequentially allocated queue must test for the occurrence of an overflow condition and take corrective action if such a condition obtains.

If after a deletion $front$ catches up with $rear$, i.e., if $front = rear$, then we get an *empty queue*. An attempt to delete from an empty queue results in an *underflow condition*. An implementation of the delete operation must test for the occurrence of an underflow condition. What action is taken when such a condition arises usually depends on the application at hand.

Consistent with the condition $front = rear$ indicating an empty queue, we *initialize* a queue by setting

$$front \leftarrow rear \leftarrow 0$$

Setting $rear$ to 0 means that n insertions can be performed before we get a full queue. For this reason, we should re-initialize the queue after a deletion empties it.

5.1.1 Implementing the insert operation

The EASY procedure ENQUEUE implements the insert operation for a sequentially allocated queue represented by the data structure $Q = [Q(1:n), front, rear]$ following the conventions stated above. A pointer to the structure Q is passed to ENQUEUE (line 1) allowing access to its components, viz., the array Q of size n in which the queue resides, and the pointers $front$ and $rear$ as defined above. The variable x contains the element to be inserted into the queue. Line 2 tests for overflow; if an overflow condition does not exist, or if it does but the call to MOVE_QUEUE succeeds in repacking memory, then the insertion proceeds (lines 3 and 4) and a return is made (line 5) to the calling program. Unless MOVE_QUEUE is invoked, insertion takes $O(1)$ time.

```

1  procedure ENQUEUE( $Q, x$ )
2  if  $rear = n$  then call MOVE_QUEUE( $Q$ )
3   $rear \leftarrow rear + 1$ 
4   $Q(rear) \leftarrow x$ 
5  end ENQUEUE

```

Procedure 5.1 Insertion into a straight queue (array implementation)

```

1  procedure MOVE_QUEUE(Q)
2  if  $front = 0$  then [output 'No more available space'; stop]
3      else [for  $i \leftarrow front + 1$  to  $n$  do
4           $Q(i - front) \leftarrow Q(i)$ 
5      endfor
6           $rear \leftarrow rear - front$ 
7           $front \leftarrow 0$  ]
8  end MOVE_QUEUE

```

Procedure 5.2 Repacking memory at overflow

The condition $front = 0$, coupled with the condition $rear = n$ which triggered the call to MOVE_QUEUE, means that all pre-allocated cells in Q are in use and insertion cannot proceed. Consequently, a message to this effect is issued, execution is terminated and control is returned to the runtime system via the **stop** statement (line 2). Otherwise, the queue elements are shifted downward (lines 3 – 5) in $O(n)$ time; the amount of shift is the value of $front$, as you may easily verify. Finally, the pointers to the queue are updated (lines 6 and 7) and a return is made to ENQUEUE (line 8).

5.1.2 Implementing the delete operation

The EASY procedure DEQUEUE implements the delete operation for a sequentially allocated queue $Q = [Q(1:n), front, rear]$ following the conventions stated above. The deleted element is stored in the variable x .

```

1  procedure DEQUEUE(Q,  $x$ )
2  if  $front = rear$  then call QUEUEUNDERFLOW
3      else [ $front \leftarrow front + 1$ 
4           $x \leftarrow Q(front)$ 
5          if  $front = rear$  then  $front \leftarrow rear \leftarrow 0$  ]
6  end DEQUEUE

```

Procedure 5.3 Deletion from a straight queue (array implementation)

Procedure QUEUEUNDERFLOW handles the occurrence of an underflow condition (line 2). In certain applications such a condition is not necessarily abnormal. For instance, in operating systems which use a queue to implement input/output buffers, a process which attempts to delete an item from an empty buffer may simply enter a wait state until another process inserts an item into the buffer, at which point the suspended process resumes and completes its operation normally. Procedure QUEUEUNDERFLOW is coded subject to considerations such as these.

Otherwise, if the queue is not empty, then the front element is deleted from the queue and is retrieved in x (lines 3 and 4). If now the queue is empty, it is re-initialized (line 5) to prevent it from becoming full too soon. Deletion clearly takes $O(1)$ time.

5.2 Sequential implementation of a circular queue

The need to move the queue elements forward to make room at the rear by invoking `MOVE_QUEUE` can be eliminated altogether if we consider the cells to be arranged implicitly in a circle to form a *circular queue*. In a manner analogous to that of the straight queue, we adopt the convention whereby *front* points to the cell which immediately precedes, going clockwise, the cell which contains the front element of the queue, while *rear* points to the cell which contains the rear element of the queue.

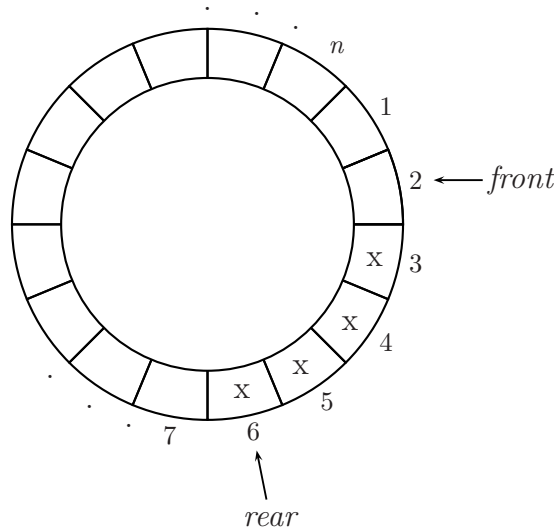


Figure 5.3 A sequentially allocated circular queue

To insert an element, say x , into the queue we proceed as follows:

```

if  $rear = n$  then  $rear \leftarrow 1$ 
      else  $rear \leftarrow rear + 1$ 
 $Q(rear) \leftarrow x$ 

```

We obtain more concise code if we use the **mod** function in lieu of the **if** statement:

```

 $rear \leftarrow rear \bmod n + 1$ 
 $Q(rear) \leftarrow x$ 

```

Similarly, to delete the front element of the queue we perform the steps

```

if  $front = n$  then  $front \leftarrow 1$ 
      else  $front \leftarrow front + 1$ 
 $x \leftarrow Q(front)$ 

```

or, more concisely

```

 $front \leftarrow front \bmod n + 1$ 
 $x \leftarrow Q(front)$ 

```

Note that as insertions and deletions are done, the queue moves in a clockwise direction. If after a deletion $front$ catches up with $rear$, i.e., if $front = rear$, then we get an empty queue. Consistent with this condition we initialize a circular queue by setting

$$front \leftarrow rear \leftarrow 1$$

The choice of the constant 1 is arbitrary; we could have chosen any number from 1 thru n .

If after an insertion $rear$ catches up with $front$, a condition also indicated by $front = rear$, then all cells are in use and we get a full queue. In order to avoid having the same relation signify two different conditions, we will not allow $rear$ to catch up with $front$ by considering the queue to be full when exactly one free cell remains. Thus a full queue is indicated by the relation $front = rear \bmod n + 1$.

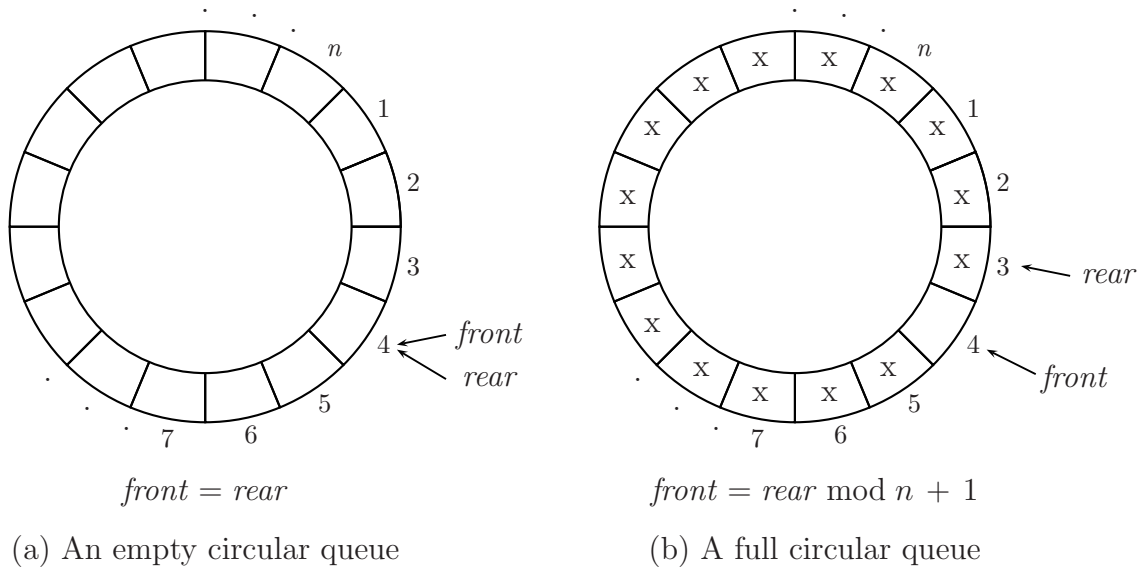


Figure 5.4 Boundary conditions for a circular queue

5.2.1 Implementing the insert and delete operations for a circular queue

The EASY procedure ENQUEUE implements the insert operation for a sequentially allocated circular queue represented by the same data structure $Q = [Q(1:n), front, rear]$ following the conventions stated above. No attempt at recovery is made if an overflow condition occurs; instead, an error message is issued outright and execution is terminated (line 2). Otherwise, if there are available cells in Q , the insertion is completed (line 3). The procedure clearly takes $O(1)$ time.

```

1  procedure ENQUEUE( $Q, x$ )
2  if  $front = rear \bmod n + 1$  then [output 'No more available cells'; stop]
3  else [ $rear \leftarrow rear \bmod n + 1$ ;  $Q(rear) \leftarrow x$ ]
4  end ENQUEUE

```

Procedure 5.4 Insertion into a circular queue (array implementation)

5.2 Sequential implementation of a circular queue 117

The EASY procedure DEQUEUE implements the delete operation for a sequentially allocated circular queue $\mathbb{Q} = [Q(1:n), front, rear]$ following the conventions stated above. The deleted element is retrieved in x . Note that we need not re-initialize the queue should the current deletion make it empty. Deletion clearly takes constant time.

```
1  procedure DEQUEUE( $\mathbb{Q}, x$ )
2  if  $front = rear$  then call QUEUEUNDERFLOW
3      else [ $front \leftarrow front \bmod n + 1$ ;  $x \leftarrow Q(front)$  ]
4  end DEQUEUE
```

Procedure 5.5 Deletion from a circular queue (array implementation)

We close this section by comparing the two sequential implementations of a queue in an array of fixed size n . For the straight queue (see Figure 5.2):

1. Updating $front$ or $rear$ simply requires adding 1 to the current value.
2. Since the queue moves towards the rear of the array as insertions and deletions are performed, an overflow condition may occur even though there may still be unused cells in the forward portion of the array, requiring memory repacking through a call to MOVE_QUEUE.
3. The queue is re-initialized after a deletion operation empties it.

For the circular queue (see Figure 5.3):

1. Updating $front$ or $rear$ requires the use of the **if** statement or the **mod** function to handle the wrap-around from cell n to cell 1.
2. Overflow occurs only if all cells (except one, as required by our definition of a full queue) are in use.
3. There is no need to re-initialize the queue after a deletion empties it.

The conventions we have adopted for the pointers $front$ and $rear$ as shown in Figures 5.2 and 5.3 are from KNUTH1[1997], pp. 244–245; these are by no means the only possible choices. For instance, we could have defined $front$ such that it points to the cell which actually contains the front element of the queue. Or we could have maintained only one pointer, say $front$, and kept a count of the number of elements in the queue, say $count$; we could then compute the value of $rear$ knowing $front$ and $count$. In any event, we should see to it that the *boundary conditions* which indicate whether the queue is empty or full (as shown, for instance, in Figure 5.4) are consistent with the conventions we choose. Likewise, the way we initialize the queue should be consistent with the condition which signifies that it is empty.

5.2.2 A C implementation of the array representation of a circular queue

Let us now consider how we might implement the data structure $\mathbb{Q} = [Q(1:n), front, rear]$ in C along with its associated procedures. As with the stack of the previous session, we will use a C structure to represent \mathbb{Q} . We will call the structure **queue** and this name serves as its type specifier. Its elements are **front** and **rear** which are of type **int**, and an array **Queue[n]** of size n , where each element is of type **QueueElemType**; what **QueueElemType** is depends on the current application. To do away with **MOVE_QUEUE**, we will use the circular array representation of \mathbb{Q} ; this means that the queue can have at most $n - 1$ elements. Since array indexing in C starts at 0, we initialize the queue by setting **front** and **rear** to 0 (actually, any integer between 0 and $n - 1$ will do).

To update the pointers **front** and **rear** we will use the **mod** function instead of the **if** statement to handle the wraparound from $n - 1$ to 0. Again, since indexing in C starts at 0, we need to replace the expressions $front \bmod n + 1$ and $rear \bmod n + 1$ to $(front + 1) \bmod n$ and $(rear + 1) \bmod n$, respectively, when these pointers are updated. The **mod** function, as defined by Eq.(2.1) of Session 2, is equivalent to the modulus division operator **%** in C when *both arguments are positive integers*, which is the case in our present usage of the function. For all other cases (e.g., negative or real arguments), the two are *not* in general equivalent or **%** may not be used.

Program 5.1 formalizes the particular implementation of \mathbb{Q} as described. A sample calling procedure is given to illustrate how the various queue routines are invoked. The output is: 100 200 300 400 500 (verify).

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

#define n 101
typedef int QueueElemType;

struct queue
{
    int front;
    int rear;
    QueueElemType Queue[n];
};
typedef struct queue Queue;

void InitQueue(Queue *);
int IsEmptyQueue(Queue *);
void ENQUEUE(Queue *, QueueElemType);
void DEQUEUE(Queue *, QueueElemType *);
void QueueOverflow(void);
void QueueUnderflow(void);
```

```

main()
{
    Queue Q; QueueElemType x; int i;
    clrscr();
    InitQueue(&Q);
    for(i=1; i<=5; i++) ENQUEUE(&Q,100*i);
    while(!IsEmpty(&Q)) {
        DEQUEUE(&Q,&x);
        printf("%d ",x);
    }
    return 0;
}

void InitQueue(Queue *Q)
{
    Q->front = 0; Q->rear = 0;
}

int IsEmptyQueue(Queue *Q)
{
    return(Q->front == Q->rear);
}

void QueueOverflow(void)
{
    printf(" Queue overflow detected.\n"); exit(1);
}

void QueueUnderflow(void)
{
    printf(" Queue underflow detected.\n"); exit(1);
}

void ENQUEUE(Queue *Q, QueueElemType x)
{
    if(Q->front == (Q->rear + 1)%n) QueueOverflow();
    else {
        Q->rear = (Q->rear + 1)%n;
        Q->Queue[Q->rear] = x;
    }
}

void DEQUEUE(Queue *Q, QueueElemType *x)
{
    if(Q->front == Q->rear) QueueUnderflow();
    else {
        Q->front = (Q->front + 1)%n;
        *x = Q->Queue[Q->front];
    }
}

```

Program 5.1 A C implementation of the array representation of a circular queue

5.3 Linked list implementation of a queue

Figure 5.5 depicts a queue implemented as a linked linear list. The front element of the queue is x_1 and the rear element is x_n . The pointer variable *front* points to the node which contains the front element of the queue while the pointer variable *rear* points to the node which contains the rear element of the queue. Note that the *LINK* field of the rear node is set to Λ . Two external pointers to the queue are used so that both the enqueue and dequeue operations can be done in $O(1)$ time.

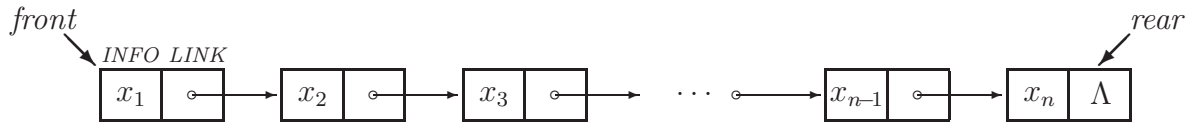


Figure 5.5 A queue represented as a linked linear list

The relation $front = \Lambda$ means that the queue is empty. Note that this boundary condition for an empty queue is not arbitrarily defined; we will see shortly that it arises when we perform a deletion operation on a queue with only one element and represented as in Figure 5.5. Consistent with this boundary condition indicating an empty queue, we initialize the queue by setting

$$front \leftarrow \Lambda$$

5.3.1 Implementing the insert operation

The EASY procedure ENQUEUE implements the insert operation for a linked queue represented by the data structure $\mathbb{Q} = [(INFO, LINK), front, rear]$ following the conventions indicated in Figure 5.5.

```

1  procedure ENQUEUE( $\mathbb{Q}, x$ )
2  call GETNODE( $\alpha$ )
3   $INFO(\alpha) \leftarrow x$ 
4   $LINK(\alpha) \leftarrow \Lambda$ 
5  if  $front = \Lambda$  then  $front \leftarrow rear \leftarrow \alpha$ 
6      else [ $LINK(rear) \leftarrow \alpha$ ;  $rear \leftarrow \alpha$ ]
7  end ENQUEUE

```

Procedure 5.6 Insertion into a queue (linked list implementation)

A pointer to the structure \mathbb{Q} is passed to ENQUEUE allowing access to all its components (line 1). To insert the new element, x , into the queue we first get a node from the memory pool (line 2). Recall that if GETNODE is unable to get a node from the pool it issues a message to this effect and terminates execution; otherwise, it returns control to ENQUEUE with α pointing to the new node. The new element is stored in the *INFO* field of the new node (line 3) and the *LINK* field is set to Λ (line 4) since this node becomes the last node in the list. Finally the new node is inserted into the list. Two cases are considered, namely, insertion into an empty queue (line 5) and insertion into an existing queue (line 6). The time complexity of the procedure is clearly $O(1)$.

5.3.2 Implementing the delete operation

The EASY procedure `DEQUEUE` implements the delete operation for a linked queue represented by the data structure $\mathbb{Q} = [(INFO, LINK), front, rear]$ following the conventions indicated in Figure 5.5.

```

1  procedure DEQUEUE( $\mathbb{Q}, x$ )
2  if  $front = \Lambda$  then call QUEUEUNDERFLOW
3      else [ $x \leftarrow INFO(front)$ 
4           $\alpha \leftarrow front$ 
5           $front \leftarrow LINK(front)$ 
6          call RETNODE( $\alpha$ )]
7  end DEQUEUE

```

Procedure 5.7 Deletion from a queue (linked list implementation)

As with the previous implementations of the delete operation, an underflow condition is handled by invoking `QUEUEUNDERFLOW` (line 2); otherwise, the front element is retrieved in x (line 3). A temporary pointer to the front node is created (line 4) before $front$ is updated to point to the next node in the list (line 5). Finally the old front node is deleted and returned to the memory pool (line 6). The delete operation clearly takes constant time.

If a queue has only one element, both $front$ and $rear$ will point to the node which contains this element. If now we perform a delete operation by invoking `DEQUEUE`, line 5 will set $front$ to null; thus the condition $front = \Lambda$ indicates that the queue is empty. Consistent with this, we initialize the queue by setting $front \leftarrow \Lambda$, as we have earlier pointed out.

5.3.3 A C implementation of the linked list representation of a queue

To implement the data structure $\mathbb{Q} = [(INFO, LINK), front, rear]$ of Figure 5.5 and its associated procedures in C, we define a structure called `queuenode` with structure elements `INFO`, which is of type `QueueElemType`, and `LINK`, which is a pointer to `queuenode`, to represent a node in the linked list; and another structure `queue` with components `front` and `rear`, which are pointers to `queuenode`. Together, `queuenode` and `queue` represent the data structure \mathbb{Q} .

Program 5.2 formalizes this particular implementation of \mathbb{Q} as described. The same `main` procedure in Program 5.1 may be used to test the present routines; needless to say, we get the same output. It is important to note that insofar as the calling procedure is concerned, it is immaterial how \mathbb{Q} is implemented; this is because both the array and linked implementations present the same interface to the calling procedure.

122 SESSION 5. Queues and Deques

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef int QueueElemType;
typedef struct queuenode QueueNode;
struct queuenode
{
    QueueElemType INFO;
    QueueNode * LINK;
};
struct queue
{
    QueueNode * front;
    QueueNode * rear;
};
typedef struct queue Queue;

void InitQueue(Queue *);
int IsEmptyQueue(Queue *);
void ENQUEUE(Queue *, QueueElemType);
void DEQUEUE(Queue *, QueueElemType *);
void QueueOverflow(void);
void QueueUnderflow(void);

main()
{
    Queue Q; QueueElemType x; int i;
    clrscr();
    InitQueue(&Q);
    for(i=1; i<=5; i++) ENQUEUE(&Q,100*i);
    while(!IsEmptyQueue(&Q))
    {
        DEQUEUE(&Q,&x);
        printf ("%d ",x);
    }
    return 0;
}

void InitQueue(Queue *Q)
{
    Q->front = NULL;
}

int IsEmpty(Queue *Q)
{
    return(Q->front == NULL);
}

```

```
void QueueOverflow(void)
{
    printf("Unable to get a node from the memory pool.\n");
    exit(1);
}
```

```
void QueueUnderflow(void)
{
    printf("Queue underflow detected.\n");
    exit(1);
}
```

```
void ENQUEUE(Queue *Q, QueueElemType x)
{
    QueueNode * alpha;
    alpha = (QueueNode *) malloc(sizeof(QueueNode));
    if(alpha == NULL) QueueOverflow();
    else
    {
        alpha->INFO = x;
        alpha->LINK = NULL;
        if(Q->front == NULL)
        {
            Q->front = alpha;
            Q->rear = alpha;
        }
        else
        {
            Q->rear->LINK = alpha;
            Q->rear = alpha;
        }
    }
}
```

```
void DEQUEUE(Queue *Q, QueueElemType *x)
{
    QueueNode * alpha;
    if (Q->front == NULL) QueueUnderflow();
    else
    {
        *x = Q->front->INFO;
        alpha = Q->front;
        Q->front = Q->front->LINK;
        free(alpha);
    }
}
```

Program 5.2 A C implementation of the linked list representation of a queue

5.4 Application of queues: topological sorting

If you consult most books on Data Structures, you will find the *topological sorting problem* discussed in a chapter on graphs, rather than in a chapter on queues. We include the topic at this point because the algorithm to solve the problem as given in KNUTH1[1997], pp. 261–268, contains a number of features that are particularly relevant to what we have studied thus far. These include:

1. the use of both sequential and linked allocation in representing a data structure — these two ways of organizing memory are not mutually exclusive; we can very well have a data structure that is partly contiguous and partly linked, depending primarily on the type of operations we perform on the structure
2. the use of a linked queue embedded in an array — dynamic variables and pointers are not the only vehicles for implementing the linked design; as pointed out in section 1.4 of Session 1, arrays and cursors can be used as well
3. the use of some simple techniques to reduce both the time and space requirements (if not complexity) of an algorithm — utilizing preallocated memory for different purposes and avoiding unnecessary or multiple ‘passes’ through a structure (e.g., a list or an array) are ways to accomplish this

Knuth’s solution to the topological sorting problem demonstrates in a concrete way how the ideas enumerated above can be implemented in a straightforward way.

The topological sorting problem

Topological sorting, as a process, is applied to the elements of a set, say S , on which *partial order* is defined. Recall from Session 2 that a partial order is a binary relation R on S that is *reflexive*, *antisymmetric* and *transitive*. As a particular example, we saw that the relation ‘ x exactly divides y ’ on the set of positive integers is a partial order. For purposes of the present discussion, let us denote the relation R by the symbol ‘ \preceq ’ (read: ‘precedes or equals’), satisfying the above properties for any elements x , y and z in S , thus:

1. Reflexivity: $x \preceq x$
2. Antisymmetry: if $x \preceq y$ and $y \preceq x$, then $x = y$.
3. Transitivity: if $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

If $x \preceq y$ and $x \neq y$, then we write $x \prec y$ (read: ‘ x precedes y ’). An equivalent set of properties to define partial order for this case is

1. Irreflexivity: $x \not\prec x$
2. Asymmetry: if $x \prec y$ then $y \not\prec x$
3. Transitivity: if $x \prec y$ and $y \prec z$, then $x \prec z$.

5.4 Application of queues: topological sorting 125

Other familiar examples of partial order from mathematics are the relations ' \leq ' on real numbers and ' \subseteq ' on sets. Another example of partial order is shown in the figure below (although you may not have thought of it in this light before):

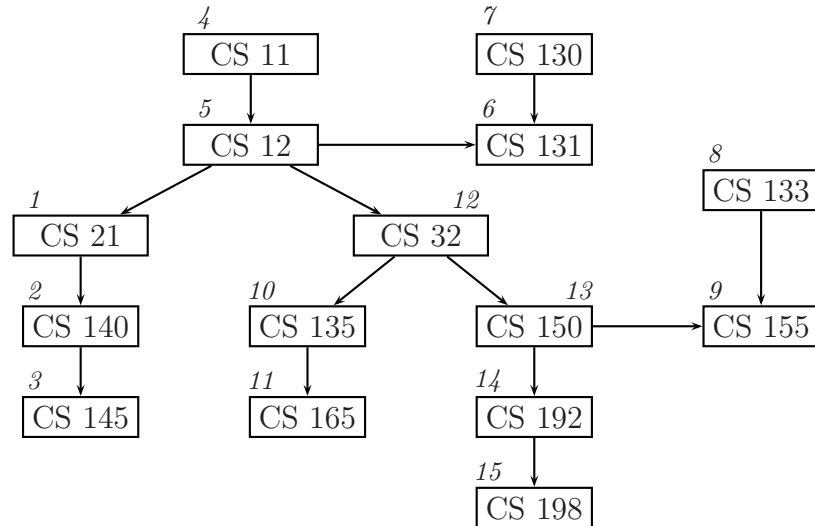


Figure 5.6 A partial ordering of 15 objects

Figure 5.6 shows the prerequisite structure among the CS courses in the BSCS curriculum (Mathematics prerequisites are not shown). This structure can be represented as a partial ordering: $\text{CS } 11 \prec \text{CS } 12$, $\text{CS } 12 \prec \text{CS } 32$, $\text{CS } 12 \prec \text{CS } 21$, and so on. The relation $\text{CS } 11 \prec \text{CS } 12$, means, of course, that you must take and pass CS 11 before you can take CS 12.

To make the discussion more general, but utilizing the figure above as the vehicle for exposition, let us imagine the figure as a partial ordering of n objects labeled 1 through n . In the relation $j \prec k$, denoted by an arrow from object j to object k , we say that j is a **direct predecessor** of k and that k is a **direct successor** of j . *The topological sorting problem is to arrange the objects in a linear sequence such that no object appears in the sequence before its direct predecessor(s).*

It is immediately clear that there is, in general, more than one linear arrangement of the objects satisfying this sorting rule. Three such sequences are given below for the partial order shown in Figure 5.6. [Note: If we consider these labels as representing the CS courses so labeled, all the sequences shown are valid but not necessarily followed in practice; this is because non-CS prerequisites have been ignored in the partial order of Figure 5.6. For instance, it is possible, but most unlikely, for a CS student to be taking 4 (CS 11), 7 (CS 130) and 8 (CS 133) together in the same semester.]

4	7	8	5	6	12	1	13	10	2	9	14	11	3	15
4	5	1	12	10	13	2	3	11	14	7	6	8	9	15
4	5	12	1	10	2	13	8	11	14	7	3	9	15	6

Figure 5.7 Topologically sorted sequences for the partial order of Figure 5.6

Developing an algorithm to solve the topological sorting problem

We consider, in turn, the three main components of the desired solution, namely: the form of the input, the form of the output and the algorithm proper.

Input: To communicate to the computer the partial order defined on the n objects, we can input pairs of numbers of the form j, k for each relation $j \prec k$. For instance, the following input describes the partial order shown in Figure 5.6:

1,2; 2,3; 4,5; 5,1; 5,12; 5,6; 7,6; 8,9; 10,11; 12,10; 12,13; 13,14; 13,9; 14,15.

Note that each pair j, k corresponds to an arrow in the figure. The number pairs can be input in any order.

Output: The output is simply a linear sequence of the objects, represented by their integer labels, such that no object appears in the sequence before its direct predecessors.

Algorithm proper: According to the rule for topological sorting, an object can be placed in the output only after its direct predecessor(s) have already been placed in the output. One simple way to satisfy this requirement is to keep, for each object, a *count of direct predecessors*. An object whose count is zero (because it has no direct predecessor) or whose count drops to zero (because its direct predecessors have already been placed in the output) is ready for output. Once such an object is placed in the output, its direct successors are located so that their count of direct predecessors, in turn, can be decreased by 1. Locating the direct successors of a particular object can be efficiently accomplished by maintaining, for each object, a *list of direct successors*.

The count of direct predecessors for each object can be stored in an array of size n , say, $COUNT(1 : n)$. The list of direct successors for each object, on the other hand, can be represented as a singly-linked list, where each node has structure, say, $[SUCC, NEXT]$. The pointers to the n lists can be maintained in another array of size n , say, $LIST(1 : n)$. Thus we use both sequential and linked allocation techniques as needed.

Initially, we set the $COUNT$ array to zero and the $LIST$ array to null. Subsequently, when an input pair of the form j, k arrives, we update the count of direct predecessors of object k , thus

$$COUNT(k) \leftarrow COUNT(k) + 1$$

and we add object k to the list of direct successors of object j . Since it is not necessary to keep the nodes which comprise the list of direct successors of object j arranged in some specific order, we insert the node for object k at the *head* of the list, thus

```

call GETNODE( $\alpha$ )
 $SUCC(\alpha) \leftarrow k$ 
 $NEXT(\alpha) \leftarrow LIST(j)$ 
 $LIST(j) \leftarrow \alpha$ 

```

Note that the insertion takes constant time; had we appended the new node at the end of the list, the insertion would have taken $O(l)$ time, where l is the length of the list.

5.4 Application of queues: topological sorting 127

Figure 5.8 below shows the *COUNT* and *LIST* arrays and the corresponding linked lists generated by the input 1,2; 2,3; 4,5; 5,1; 5,12; 5,6; 7,6; 8,9; 10,11; 12,10; 12,13; 13,14; 13,9 and 14,15 for the partial order shown in Figure 5.6. For instance, we see that $COUNT(6) = 2$ since object 6 has two direct predecessors, namely, objects 5 and 7. Similarly, we see that the list of direct successors of object 5 consists of the nodes for objects 6, 12 and 1, since object 5 is a direct predecessor of these objects. That the objects appear in the list in the order shown (6,12,1) is a consequence of the order in which the pertinent relations have been input ($\dots, 5, 1; 5, 12; 5, 6, \dots$) and the way in which new nodes are inserted into a list. Figure 5.8 depicts graphically the internal representation in computer memory of the partial order defined on the 15 objects in Figure 5.6 such that topological sorting can be carried out in a most efficient manner.

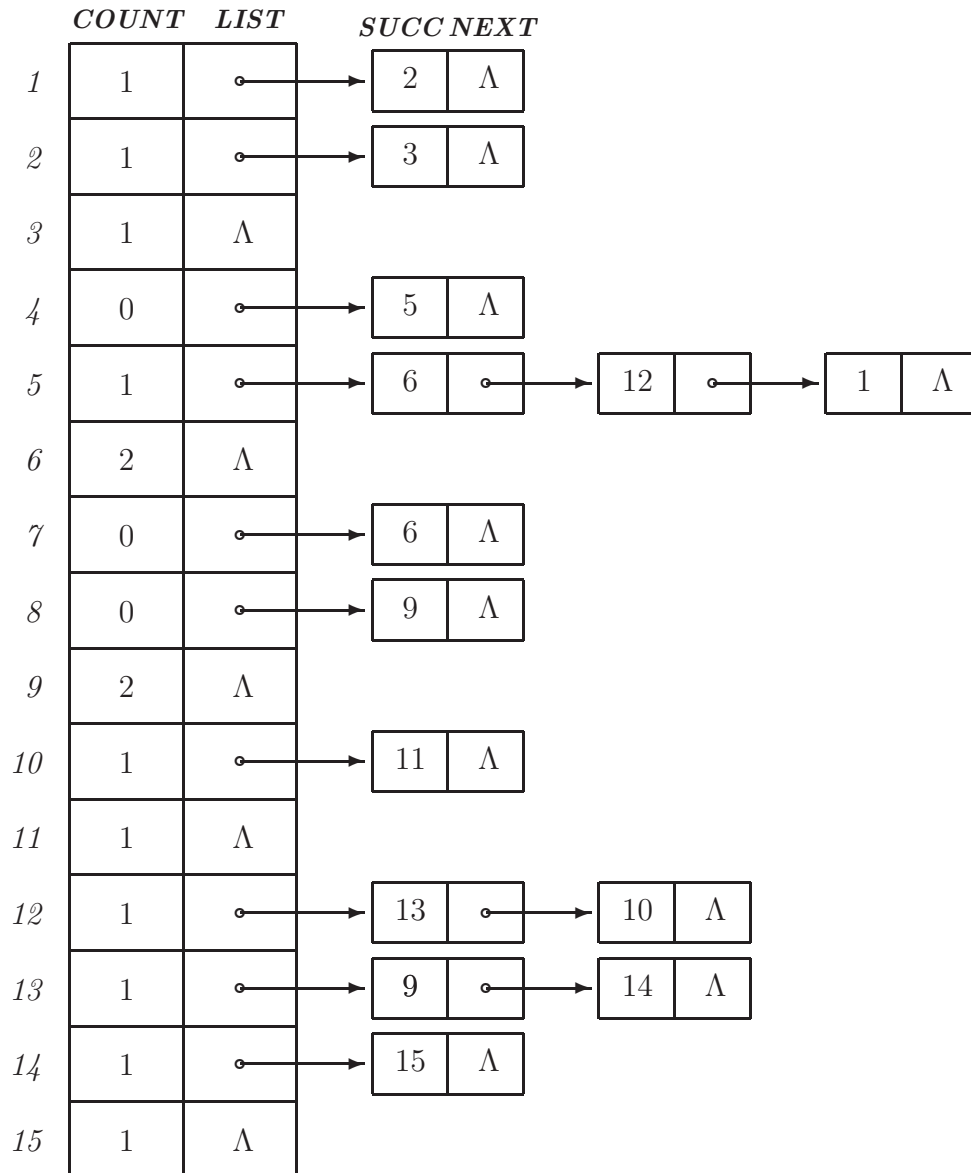


Figure 5.8 Internal representation of the partial order of Figure 5.6

128 SESSION 5. Queues and Deques

To generate the output, which is a linear ordering of the objects such that no object appears in the sequence before its direct predecessors, we proceed as follows:

1. Look for an object, say object k , whose count of direct predecessors is zero, i.e., $COUNT(k) = 0$, and put k in the output. If no such object can be found, stop.
2. Scan the list of direct successors of object k , and decrement the count of each such successor by 1. Go back to Step 1.

For the partial order represented in Figure 5.8, a straightforward application of this procedure requires two passes through the $COUNT$ array. During the first pass, the following objects will be placed in the output: 4, 5, 7, 8, 12, 13, 14 and 15. During the second pass, the remaining objects will be placed in the output: 1, 2, 3, 6, 9, 10 and 11. [Verify.] To avoid having to go through the $COUNT$ array repeatedly as we look for objects with a count of zero, we will constitute all such objects into a *linked queue*. Initially, the queue will consist of objects with no direct predecessors (there will always be at least one such object if the input is indeed a partial order). Subsequently, each time that the count of direct predecessors of an object drops to zero, the object is inserted into the queue, ready for output. To avoid having to allocate additional space for the queue, we will overstore the queue in the $COUNT$ array. Since the count of each object, say j , in the queue *is* zero, we now reuse $COUNT(j)$ as a *link field*, appropriately renamed $QLINK(j)$, such that

$$\begin{aligned} QLINK(j) &= k && \text{if } k \text{ is the next object in the queue} \\ &= 0 && \text{if } j \text{ is the rear element in the queue} \end{aligned}$$

The following segment of EASY code initializes the queue by scanning the $COUNT$ array once and linking together, in the $COUNT/QLINK$ array itself, all objects with a count of zero. What we have here is essentially a cursor implementation of a linked queue.

```
front ← 0
for  $k \leftarrow 1$  to  $n$  do
    if  $COUNT(k) = 0$  then if  $front = 0$  then  $front \leftarrow rear \leftarrow k$ 
    else [  $QLINK(rear) \leftarrow k$ ;  $rear \leftarrow k$  ]
endfor
```

Figure 5.9 shows the initial output queue embedded in the $COUNT/QLINK$ array. The entries in boldface are the links; they have replaced the old entries which are counts of zero. The front element is object 4 as indicated by the cursor $front$; $QLINK(4) = 7$ means that the next element in the queue is object 7. By the same token, $QLINK(7) = 8$ means that object 8 comes next. Finally, $QLINK(8) = 0$ means that object 8 is the rear element, as indicated by the cursor $rear$.

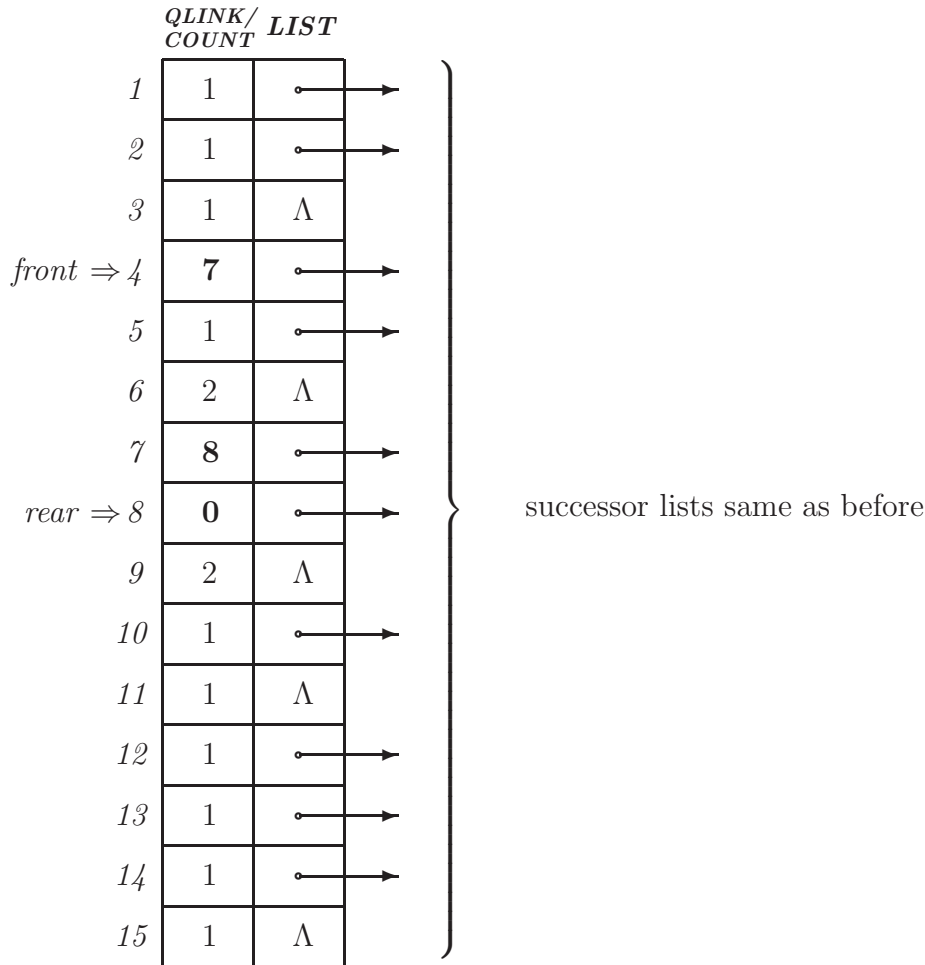


Figure 5.9 The initial output queue embedded in the *COUNT/QLINK* array

With the initial output queue constituted, the desired output can now be generated as follows:

1. If the queue is empty, stop. Otherwise, dequeue front element, say object j , and place it in the output.
2. Scan the list of direct successors of object j and decrement the count of direct predecessors of each successor by 1. Enqueue every successor whose count drops to zero. Go back to Step 1.

If the input to the algorithm is correct, i.e., if the input relations satisfy partial order, then the algorithm terminates when the queue is empty with all n objects placed in the output. If, on the other hand, partial order is violated in that there are objects which constitute one or more loops (for instance, $1 \prec 2$; $2 \prec 3$; $3 \prec 4$; $4 \prec 1$), then the algorithm still terminates, but objects comprising a loop will not be placed in the output.

The algorithm described above to solve the topological sorting problem is given as Algorithm T in KNUTH1[1997], p. 265. It is implemented as procedure `TOPOLOGICAL_SORT` below.

130 SESSION 5. Queues and Deques

▷ Given a partial ordering of n objects labeled 1 thru n , procedure generates a
 ▷ topological ordering of the n objects

```

1  procedure TOPOLOGICAL_SORT( $n$ )
2  array  $COUNT(1:n)$ ,  $LIST(1:n)$ 
3  node( $SUCC, NEXT$ )
4  equivalence  $COUNT, QLINK$ 
  
```

▷ Initializations

```

5  for  $k \leftarrow 1$  to  $n$  do
6     $COUNT(k) \leftarrow 0$ 
7     $LIST(k) \leftarrow \Lambda$ 
8  endfor
9   $m \leftarrow n$     ▷  $m$  is the number of objects still to be output
  
```

▷ Read in data and generate count of direct predecessors and lists of direct successors.
 ▷ The pair 0,0 terminates the input.

```

10 input  $j, k$ 
11 while  $j \neq 0$  do
12    $COUNT(k) \leftarrow COUNT(k) + 1$ 
13   call GETNODE( $\alpha$ )
14    $SUCC(\alpha) \leftarrow k$ 
15    $NEXT(\alpha) \leftarrow LIST(j)$ 
16    $LIST(j) \leftarrow \alpha$ 
17   input  $j, k$ 
18 endwhile
  
```

▷ Initialize output queue

```

19  $front \leftarrow 0$ 
20 for  $k \leftarrow 1$  to  $n$  do
21   if  $COUNT(k) = 0$  then if  $front = 0$  then  $front \leftarrow rear \leftarrow k$ 
22   else [ $QLINK(rear) \leftarrow k$ ;  $rear \leftarrow k$ ]
23 endfor
  
```

▷ Output objects

```

24 while  $front \neq 0$  do
25   output  $front$ 
26    $m \leftarrow m - 1$ 
27    $\alpha \leftarrow LIST(front)$ 
28   while  $\alpha \neq \Lambda$  do
29      $k \leftarrow SUCC(\alpha)$ 
30      $COUNT(k) \leftarrow COUNT(k) - 1$ 
31     if  $COUNT(k) = 0$  then [ $QLINK(rear) \leftarrow k$ ;  $rear \leftarrow k$ ]
32      $\alpha \leftarrow NEXT(\alpha)$ 
33   endwhile
34    $front \leftarrow QLINK(front)$ 
35 endwhile
  
```

▷ Check for presence of loops

```

36 if  $m > 0$  then output 'Some objects comprise a loop'
37 end TOPOLOGICAL_SORT
  
```

Procedure 5.8 Topological sorting

Both sequential allocation (line 2) and linked allocation (line 3) are used to implement the algorithm. Line 4 indicates that the array *COUNT* may also be addressed as *QLINK*, i.e., *COUNT*(*i*) and *QLINK*(*i*), $1 \leq i \leq n$, refer to the same memory cell. Which name we use depends on what the value stored in the cell signifies. We will see shortly how this works out. The array *LIST* is an array of pointers to *n* linked lists composed of nodes with node structure (*SUCC*, *NEXT*); initially all the lists are null (line 7).

The input pair *j, k* in line 10 means that object *j* precedes object *k*, where *j* and *k* are integer labels from 1 through *n*. We assume that the pair 0,0 terminates the input (line 11). For each input *j, k* line 12 updates the count of direct predecessors of object *k* to include object *j*, and lines 13 thru 16 update the list of direct successors of object *j* to include object *k*. For reasons already pointed out, the node for object *k* is appended at the head of the list of object *j*.

With the input phase complete, objects with no predecessors are constituted into a linked queue in lines 19 thru 23, ready for output. You will find similar lines of code (lines 5 and 6) in Procedure 5.6 in section 2 but with one important difference. In this instance, we are using an array (the *COUNT/QLINK* array) and array indices (cursors), instead of dynamic variables and pointers, to realize a linked queue. Note that the *COUNT* array is scanned only once in setting up the queue.

The actual output is generated in lines 24 thru 35, which implement the two-step procedure outlined above. Note that the output loop is on the queue, terminating when the queue becomes empty, indicated by the condition *front* = 0. Line 36 tests whether all *n* objects have been placed in the output; if not, some objects comprise a loop or loops, and a message to this effect is issued. KNUTH1[1997], p. 271 (Exercise 23) and p. 548 (answer to Exercise 23), gives an algorithm for identifying objects which constitute a loop; you may want to incorporate this in procedure *TOPOLOGICAL_SORT*.

For the partial order defined in Figure 5.6 (also Figure 5.8), the output produced by procedure *TOPOLOGICAL_SORT* is the first of the three topological orderings shown in Figure 5.7, as you may easily verify.

Analysis of procedure *TOPOLOGICAL_SORT*

Let *e* be the number of relations of the form $j \prec k$ which define a partial order on the given set of *n* objects. For instance, this corresponds to the number of arrows for the partial order shown in Figure 5.6, and also to the number of nodes which comprise the lists of direct successors in Figure 5.8.

Looking now at Procedure 5.8, it is clear that the initialization segments (lines 5–8 and 19–23) both take $O(n)$ time. The input phase (lines 10–18) takes $O(e)$ time; had we appended a new node at the end of a list instead of at the head, this would take $O(e^2)$ time instead. Finally, during the output phase (lines 24–35), each of the *n* objects is placed in turn in the output and its list of direct successors scanned once, thus taking $O(n + e)$ time. Note that if we did not use a queue, but made multiple passes through the *COUNT* array instead, this phase would take $O(n^2)$ time.

Thus the time complexity of procedure *TOPOLOGICAL_SORT* is $O(n + e)$, i.e., it is linear in the size of the problem as defined by the number of objects, *n*, and the number of input relations, *e*.

5.5 The deque as an abstract data type

A deque (short for *double-ended queue* and pronounced ‘deck’) is a linearly ordered list of elements in which insertions and deletions are allowed at both the *left* and *right* ends of the list, and here only.

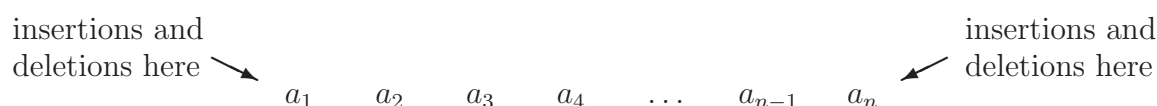


Figure 5.10 The deque ADT

The four basic operations defined on a deque are

1. insert an element at the left end of the deque
2. insert an element at the right end of the deque
3. delete the leftmost element of the deque
4. delete the rightmost element of the deque

If insertions are allowed only at one end we obtain an *input-restricted deque*; if deletions are allowed only at one end we obtain an *output-restricted deque*. These restrictions are not mutually exclusive. Depending on which of the four operations are disallowed, a deque may function either as a stack or as a queue. The other auxiliary operations defined on a deque are:

5. initialize the deque
6. test if the deque is empty
7. test if the deque is full

As with a stack or a queue, a deque in the abstract may be empty, but it may not be full. It should always be possible to insert yet one more element into a deque. The state whereby a deque is full is a consequence of implementing the deque ADT on a machine with finite memory.

As with the stack and the queue, there are two common ways to implement the deque ADT as a concrete data structure, namely:

1. sequential implementation — the deque is stored in a one-dimensional array
2. linked implementation — the deque is represented as a linked linear list

Akin to the queue, there are two common variations on the sequential representation of a deque using a one-dimensional array — straight or circular.

5.6 Sequential implementation of a straight deque

Figure 5.11 shows the sequential representation of a deque in an array D of size n . The elements marked 'x' comprise the deque. We adopt the convention whereby the variable l points to the cell which contains the leftmost element of the deque, and the variable r points to the cell which contains the rightmost element of the deque. Be sure to keep the distinction between the deque (of size $r - l + 1$) and the array (of size n) in which the deque resides.

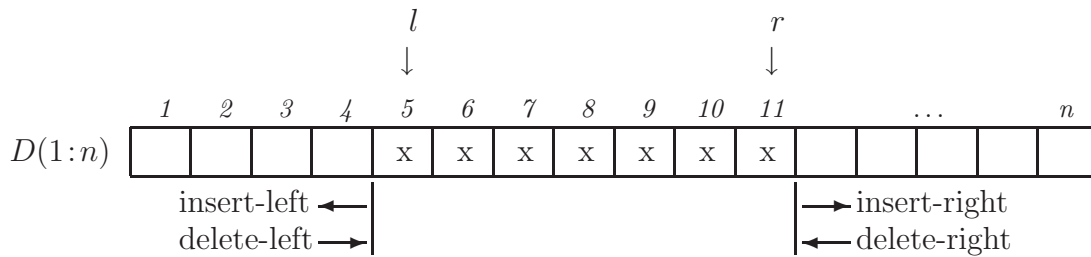


Figure 5.11 Sequential representation of a deque

With the conventions depicted in Figure 5.11, the four basic operations on a deque are implemented, thus:

1. Insert element x at left end of deque
 $l \leftarrow l - 1$
 $D(l) \leftarrow x$
2. Insert element x at right end of deque
 $r \leftarrow r + 1$
 $D(r) \leftarrow x$
3. Delete leftmost element of deque and retrieve in x
 $x \leftarrow D(l)$
 $l \leftarrow l + 1$
4. Delete rightmost element of deque and retrieve in x
 $x \leftarrow D(r)$
 $r \leftarrow r - 1$

Consider a deque with only one element, i.e., $l = r$. If now we perform a delete-left or a delete-right operation, then we get an *empty deque* indicated by the condition $l > r$. Assuming that insertions and deletions are equally likely to occur at either the left or right end of the deque, and consistent with the condition $l > r$ indicating an empty deque, we initialize a deque as follows:

$$r \leftarrow \lfloor n/2 \rfloor$$

$$l \leftarrow r + 1$$

With the deque initialized as indicated, it takes $\lfloor n/2 \rfloor$ insert-right operations before the deque becomes 'full' with $r = n$, or $\lfloor n/2 \rfloor$ insert-left operations before the deque becomes 'full' with $l = 1$. An overflow condition occurs if an insert-right operation is attempted and $r = n$ or an insert-left operation is attempted and $l = 1$. In either case, if there are still available cells at the other end of the deque, the deque is moved such that it occupies the middle $r - l + 1$ cells of the array D .

5.6.1 Implementing the insert operation

The EASY procedure INSERT_DEQUE implements both the insert-left and insert-right operations, while procedure DELETE_DEQUE implements both the delete-left and delete-right operations for the deque $\mathbb{D} = [D(1:n), l, r]$. Which operation is intended is indicated by setting the boolean parameter *left* to **true** if the insertion/deletion is to be at the left end of the deque, and to **false** otherwise.

```

1  procedure INSERT_DEQUE( $\mathbb{D}$ , left, x)
2  if (left and  $l = 1$ ) or (not left and  $r = n$ ) then call MOVE_DEQUE( $\mathbb{D}$ , left)
3  if left then [  $l \leftarrow l - 1$ ;  $D(l) \leftarrow x$  ]
4      else [  $r \leftarrow r + 1$ ;  $D(r) \leftarrow x$  ]
5  end INSERT_DEQUE

```

Procedure 5.9 Insertion into a straight deque (array implementation)

A pointer to the structure $\mathbb{D} = [D(1:n), l, r]$ is passed to INSERT_DEQUE (line 1) allowing access to its components, viz., the array D of size n in which the deque resides, and the pointers l and r as defined above. Line 2 tests for overflow; if an overflow condition does not exist or if it exists but the call to MOVE_DEQUE succeeds in repacking memory, the insertion proceeds (line 3 or 4) and a return is made (line 5) to the calling program. Unless MOVE_DEQUE is invoked, the insertion takes $O(1)$ time.

```

1  procedure MOVE_DEQUE( $\mathbb{D}$ , left)
2  if  $l = 1$  and  $r = n$  then [ output ‘No more available cells’; stop ]
3  if left then [  $s \leftarrow \lceil (n - r)/2 \rceil$ 
4      for  $i \leftarrow r$  to  $l$  by  $-1$  do
5           $D(i + s) \leftarrow D(i)$ 
6      endfor
7       $r \leftarrow r + s$ ;  $l \leftarrow l + s$  ]
8  else [  $s \leftarrow \lceil (l - 1)/2 \rceil$ 
9      for  $i \leftarrow l$  to  $r$  do
10          $D(i - s) \leftarrow D(i)$ 
11     endfor
12      $r \leftarrow r - s$ ;  $l \leftarrow l - s$  ]
13 end MOVE_DEQUE

```

Procedure 5.10 Repacking memory at deque overflow

Procedure MOVE_DEQUE attempts to repack memory if possible. If it finds that all pre-allocated cells in D are already in use, it issues a message to this effect, terminates execution and returns control to the runtime system via the **stop** statement (line 2). Otherwise, it moves the deque such that it occupies the middle $r - l + 1$ cells of D by shifting the elements by the amount s as found in line 3 or 8. If the overflow occurred at the left end of the deque, the elements are shifted upward by *decreasing* addresses (lines 4 – 6); otherwise, if the overflow occurred at the right end, the elements are shifted downward by *increasing* indices (lines 9 – 11). Finally, the pointers l and r are updated (line 7 or 12), and a return is made to INSERT_DEQUE. Note that in the worst case $n - 1$ elements may have to be moved; thus the time complexity of MOVE_DEQUE is $O(n)$.

5.6.2 Implementing the delete operation

```

1  procedure DELETE_DEQUE( $\mathbb{D}$ ,  $left$ ,  $x$ )
2  if  $l > r$  then call DEQUEUEUNDERFLOW
3      else [ if  $left$  then [  $x \leftarrow D(l)$ ;  $l \leftarrow l + 1$  ]
4              else [  $x \leftarrow D(r)$ ;  $r \leftarrow r - 1$  ] ]
5  if  $l > r$  then [  $r \leftarrow \lfloor n/2 \rfloor$ ;  $l \leftarrow r + 1$  ]
6  end DELETE_DEQUE

```

Procedure 5.11 Deletion from a straight deque (array implementation)

Line 2 tests for underflow and calls DEQUEUEUNDERFLOW if such a condition obtains; what DEQUEUEUNDERFLOW does is left unspecified. Otherwise, the desired element (leftmost or rightmost) is retrieved in x and logically deleted from the deque by updating the appropriate pointer (line 3 or 4). The deque is re-initialized (line 5) if this deletion operation emptied the deque. A successful delete operation clearly takes constant time.

5.7 Sequential implementation of a circular deque

In a manner akin to the circular queue, we can consider the elements of the deque to be arranged implicitly in a circle to form a *circular deque*, as depicted in Figure 5.12. The deque is stored in cell l clockwise through cell r of the array D . As with the circular queue, we can handle the transition from cell n to cell 1, and vice versa, by using the **if** statement or the **mod** function.

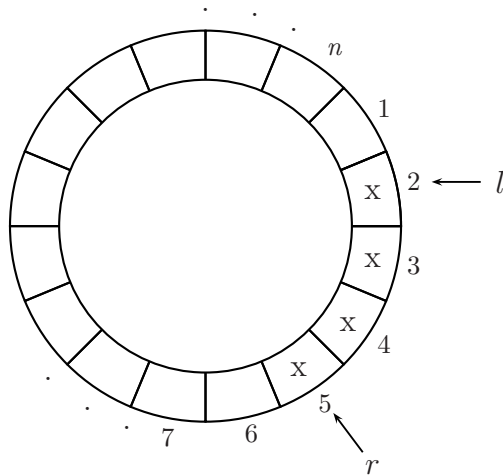


Figure 5.12 A sequentially allocated circular deque

With the conventions depicted in Figure 5.12 and using the **mod** function we can implement the four basic deque operations, thus:

1. Insert element x at left end of deque

$$l \leftarrow n - (1 - l) \bmod n$$

$$D(l) \leftarrow x$$

2. Insert element x at right end of deque

$$r \leftarrow r \bmod n + 1$$

$$D(r) \leftarrow x$$

3. Delete leftmost element of deque and retrieve in x

$$x \leftarrow D(l)$$

$$l \leftarrow l \bmod n + 1$$

4. Delete rightmost element of deque and retrieve in x

$$x \leftarrow D(r)$$

$$r \leftarrow n - (1 - r) \bmod n$$

As with the non-circular implementation (Figure 5.11), the relation $l = r$ means the deque has only one element. A delete-left or a delete-right operation empties the deque, and this is indicated by the relation $l = r \bmod n + 1$. Consistent with this, we initialize the deque such that

$$r \leftarrow i$$

$$l \leftarrow r \bmod n + 1$$

where i is any integer from 1 through n . However, the same relation $l = r \bmod n + 1$ also indicates the condition whereby all the cells are occupied, as you may easily verify. As with the circular queue, we avoid having the same relation signify two different conditions by considering the deque to be full when all but one cell are in use; then the relation $l = (r + 1) \bmod n + 1$ indicates that the deque is full.

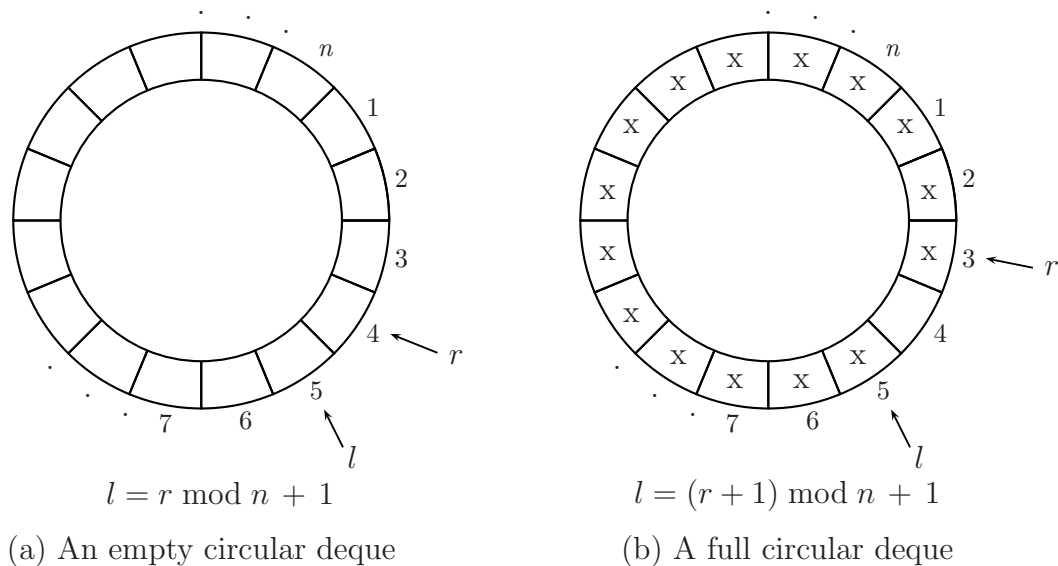


Figure 5.13 Boundary conditions for a circular deque

5.7.1 Implementing the insert and delete operations for a circular deque

The following EASY procedures implement the insert and delete operations for a circular deque $\mathbb{D} = [D(1:n), l, r]$ following the conventions indicated in Figures 5.12 and 5.13. In both cases a pointer to the structure $\mathbb{D} = [D(1:n), l, r]$ is passed to the procedure, thus allowing the procedure to gain access to the structure components. The boolean parameter *left* is set to **true** if the insertion is to be at the left end of the deque; otherwise, it is set to **false**.

```

1  procedure INSERT_DEQUE( $\mathbb{D}$ , left,  $x$ )
2  if  $l = (r + 1) \bmod n + 1$  then [ output ‘No more available cells’; stop ]
3  if left then [  $l \leftarrow n - (1 - l) \bmod n$ ;  $D(l) \leftarrow x$  ]
4      else [  $r \leftarrow r \bmod n + 1$ ;  $D(r) \leftarrow x$  ]
5  end INSERT_DEQUE

```

Procedure 5.12 Insertion into a circular deque (array implementation)

The parameter x (line 1) contains the element to be inserted into the deque. Line 2 tests for overflow. No attempt at recovery is made if such a condition exists; instead an error message is issued, execution is terminated and control is returned to the runtime system. Otherwise, the insertion proceeds (line 3 or 4) and a return is made to the calling program (line 5). The procedure clearly takes $O(1)$ time to execute.

```

1  procedure DELETE_DEQUE( $\mathbb{D}$ , left,  $x$ )
2  if  $l = r \bmod n + 1$  then call DEQUEUEUNDERFLOW
3      else [ if left then [  $x \leftarrow D(l)$ ;  $l \leftarrow l \bmod n + 1$  ]
4              else [  $x \leftarrow D(r)$ ;  $r \leftarrow n - (1 - r) \bmod n$  ]
5  end DELETE_DEQUE

```

Procedure 5.13 Deletion from a circular deque (array implementation)

Line 2 tests for underflow and calls DEQUEUEUNDERFLOW if such a condition obtains; what DEQUEUEUNDERFLOW does is left unspecified. Otherwise, the desired element (leftmost or rightmost) is retrieved in x and is logically deleted from the deque by updating the appropriate pointer (line 3 or 4). A successful delete operation clearly takes $O(1)$ time.

We have considered two variations on the array implementation of a deque. As always, there are tradeoffs. With the straight implementation, updating the variables l and r requires nothing more than addition or subtraction. However, an overflow condition may be triggered during an insert operation even though there is still available space; moving the deque elements to allow the insertion to proceed takes $O(n)$ time. With the circular implementation, overflow occurs only if all preallocated space is used up, which is reason enough to terminate execution. However, updating the variables l and r is more expensive in that it requires the use of the **if** statement or the **mod** function.

5.8 Linked list implementation of a deque

Figure 5.14 depicts a deque implemented as a singly-linked linear list. The leftmost element of the deque is x_1 and the rightmost element is x_n . The pointer variable l points to the node which contains the leftmost element, and the pointer variable r points to the node which contains the rightmost element. Note that the *LINK* field of the rightmost node is set to Λ . We will see in a moment that with these conventions the first three basic operations on a deque (insert-left, insert-right and delete-left) take $O(1)$ time; however, the delete-right operation takes $O(n)$ time.

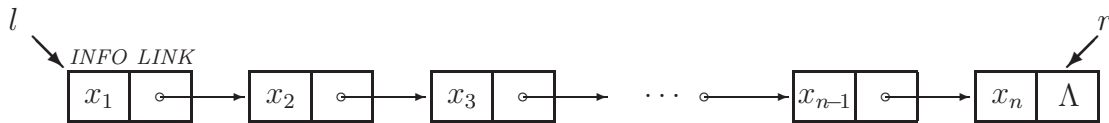


Figure 5.14 A deque represented as a singly-linked linear list

The condition $l = \Lambda$ means that the deque is empty; this is a consequence of the way in which we implement the delete operation, as we will see shortly. Consistent with this condition, we initialize the deque by setting

$$l \leftarrow \Lambda$$

5.8.1 Implementing the insert and delete operations

The following EASY procedures implement the insert and delete operations for a linked deque represented by the data structure $\mathbb{D} = [(INFO, LINK), l, r]$ according to the conventions shown in Figure 5.14. As with the earlier implementations of these operations, a pointer to the structure \mathbb{D} is passed to the procedure thereby allowing access to the structure components. The parameter *left* is set to **true** if insertion/deletion is to be at the left end of the deque; otherwise, *left* is set to **false**.

```

1  procedure INSERT_DEQUEUE( $\mathbb{D}$ , left,  $x$ )
2  call GETNODE( $\alpha$ )
3   $INFO(\alpha) \leftarrow x$ 
4  case
5    :  $l = \Lambda$  : [  $l \leftarrow r \leftarrow \alpha$ ;  $LINK(\alpha) \leftarrow \Lambda$  ]
6    : left   : [  $LINK(\alpha) \leftarrow l$ ;  $l \leftarrow \alpha$  ]
7    : else   : [  $LINK(r) \leftarrow \alpha$ ;  $r \leftarrow \alpha$ ;  $LINK(r) \leftarrow \Lambda$  ]
8  endcase
9  end INSERT_DEQUEUE

```

Procedure 5.14 Insertion into a deque (linked-list implementation)

To insert a new element x into the deque we first get a node from the memory pool (line 2). If GETNODE is unable to get a node it issues a message to this effect and terminates execution; otherwise, it returns control to INSERT_DEQUEUE with α pointing to the new node. The element to be inserted into the deque is stored in the *INFO* field of the new node (line 3) and the new node is subsequently appended to the deque (lines 4–8). Three cases are considered: insertion into an initially empty deque (line 5), at the left end (line 6) or at the right end (line 7) of the deque. The time complexity of the procedure is clearly $O(1)$.

```

1  procedure DELETE_DEQUEUE( $\mathbb{D}$ ,  $left$ ,  $x$ )
2  if  $l = \Lambda$  then call DEQUEUEUNDERFLOW
3      else [ if  $left$  or  $l = r$  then [  $\alpha \leftarrow l$ ;  $l \leftarrow LINK(l)$  ]
4          else [  $\alpha \leftarrow l$ 
5              while  $LINK(\alpha) \neq r$  do
6                   $\alpha \leftarrow LINK(\alpha)$ 
7              endwhile
8               $r \leftrightarrow \alpha \quad \triangleright$  swap pointers
9               $LINK(r) \leftarrow \Lambda$  ]
10       $x \leftarrow INFO(\alpha)$ 
11      call RETNODE( $\alpha$ ) ]
12 end DELETE_DEQUEUE

```

Procedure 5.15 Deletion from a deque (linked-list implementation)

As with all the previous implementations of the delete operation, an underflow condition is handled by calling procedure DEQUEUEUNDERFLOW. Otherwise, the leftmost or rightmost element, as prescribed by the parameter $left$, is retrieved and stored in x (line 10). If the deque has only one element, i.e., if $l = r$, a delete-left or a delete-right operation results in an empty deque. To have the same relation signify this condition, deletion from a one-element deque is implemented as a delete-left, regardless of the value of $left$ (line 3). Thus the relation $l = \Lambda$ (whatever r may be) means the deque is empty. It is clear that a delete-left operation takes $O(1)$ time; on the other hand, a delete-right operation (lines 4–9) takes $O(n)$ time, where n is the number of elements in the deque. As a final step, the deleted node is returned to the memory pool (line 11).

Summary

- The queue, next of kin to the stack, is another simple yet extremely useful ADT. The *first-in-first-out* discipline imposed on its elements allow constant time implementations of its two basic operations, viz., *enqueue* and *dequeue*.
- Queues are often used in computer simulations of systems in which the governing philosophy is first-come-first-serve; such systems abound in the real world. Queues are also used in implementing certain traversal algorithms for trees and graphs.

- The use of modular arithmetic yields an elegant sequential implementation of the queue ADT, with the queue moving in cyclic fashion as elements are enqueued and dequeued.
- The linked implementation of the queue ADT may utilize either pointers and linked nodes or cursors and parallel arrays. Knuth's solution to the topological sorting problem makes use of a linked queue embedded in a sequential array.
- Deques are not commonly used in practice; nonetheless, implementing the deque ADT is an interesting academic exercise.

Exercises

1. Find all the permutations of the integers 1, 2, 3 and 4 obtainable with a queue, assuming that they are inserted into the queue in the given order.
2. A circular queue is represented sequentially in an array $Q(1:n)$ according to the conventions shown in Figure 5.3. Derive a *single* formula in terms of *front*, *rear* and n for the number of elements in the queue.
3. Show how to represent a singly-linked queue with only *one external* pointer to the queue such that the enqueue and dequeue operations can be done in constant time.
4. Procedure TOPOLOGICAL_SORT is presented with the following input: 7,6; 6,8; 8,2; 3,9; 1,9; 4,1; 5,4; 2,1; 5,2; 0,0.
 - (a) Show the tables and lists generated by the input.
 - (b) Show the resulting output.
 - (c) Show the tables and lists after the procedure terminates.
5. Using a language of your choice, transcribe procedure TOPOLOGICAL_SORT (Procedure 5.8) into a running program. Test your program using the example given in section 5.4.
6. Using your program in Item 5, solve the topological sorting problem defined by the following input: 1,9; 1,10; 1,11; 2,1; 2,3; 2,10; 2,14; 3,4; 5,4; 6,5; 7,6; 8,7; 8,11; 8,12; 9,11; 10,3; 11,12; 11,14; 12,13; 13,4; 13,6; 14,4; 0,0.
7. An output-restricted deque is a deque in which insertions are allowed at both ends of the deque but deletions are allowed only at one end. Show how to represent a singly-linked output-restricted deque with only *one external* pointer to the deque such that insertions and deletions can be done in constant time.
8. Write EASY procedures to perform the insert and delete operations on the output-restricted deque in Item 7.
9. Using a language of your choice, transcribe procedures INSERT_DEQUE and DELETE_DEQUE (Procedures 5.12 and 5.13) for a sequentially allocated circular deque into running programs.

10. Using a language of your choice, transcribe procedures INSERT_DEQUE and DELETE_DEQUE (Procedures 5.14 and 5.15) for a singly-linked deque into running programs.

Bibliographic Notes

Procedure TOPOLOGICAL_SORT is an EASY implementation of Algorithm T (*Topological sort*) given in KNUTH1[1997], p. 265. Knuth was the first to devise a linear time algorithm to solve the topological sorting problem.

NOTES

SESSION 6

Binary Trees

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define a binary tree and explain related concepts.
2. State important properties of binary trees.
3. Enumerate the nodes of any given binary tree in preorder, inorder and postorder.
4. Design algorithms which center on the idea of traversing a binary tree, e.g., constructing a binary tree given its nodes in preorder and inorder, traversing a binary tree in level order, and so on, and implement such algorithms in EASY.
5. Describe inorder threading of a binary tree and explain the effect of threading on binary tree traversals.
6. Devise and implement algorithms pertaining to threaded binary trees.
7. Describe binary tree traversal using link inverson and Siklóssy's technique.

READINGS KNUTH1[1997], pp. 311–312, 318–330; STANDISH[1980], pp. 51–66, 74–83; HOROWITZ[1976], pp. 223–244.

DISCUSSION

Thus far we have considered only linear data structures, viz., stacks, queues and deques, which are represented in the memory of a computer either by a one-dimensional array or a singly-linked linear list. We will now consider a very important type of *nonlinear* data structure — the **binary tree**.

The binary tree is utilized in a wide variety of applications, including searching (binary search trees, AVL trees, red-black trees), sorting (heaps in heapsort), efficient encoding of strings (Huffman coding tree), representation of algebraic expressions involving binary operators (expression trees), and so on. Binary trees are used to implement priority queues, decision tables, symbol tables, and the like. Algorithms for traversing binary trees underlie other procedures (e.g., for marking, copying, etc.) for other structures such as generalized lists and multilinked structures. Applications of binary trees extend beyond the CS domain. In the next session we will use a binary tree to model an irrigation network to solve a water resources engineering problem.

6.1 Definitions and related concepts

We begin our study of binary trees appropriately by defining what a binary tree is. We adopt the following definition from KNUTH1[1997], p. 312:

DEFINITION 6.1. A binary tree is a finite set of nodes which is either empty, or consists of a root and two disjoint binary trees called the left and right subtrees of the root.

Note that the definition is recursive: a binary tree is defined in terms of a binary tree. This recursive nature of binary trees will be manifested again and again in many algorithms on binary trees which, as we will see shortly, are often cast in a recursive way.

In the discussion which follows, we will refer to the nodes of a binary tree by their *label*, as in ‘node A’, or by their *address*, as in ‘node α ’ or ‘node i ’, where α or i is the name of the pointer to the node.

Shown below are six different binary trees.

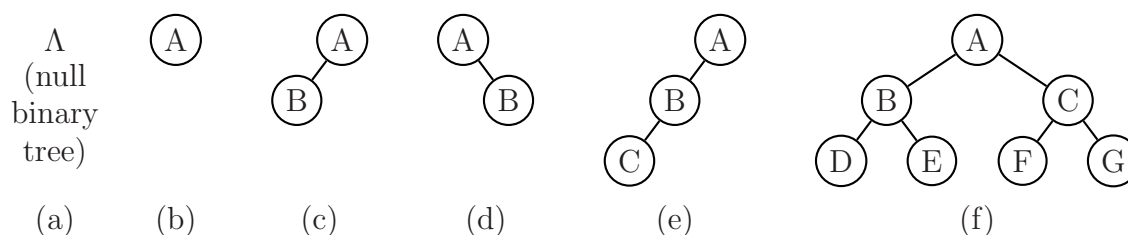


Figure 6.1 Six different binary trees

Example (a) depicts the omnipresent *null* or *empty* binary tree. Every binary tree contains the null binary tree. (In the recursive algorithms on binary trees, this is the base case.) In examples (b) to (f), node A is the root of the binary tree. In (b), both the left and right subtrees of the root are null; while in (c) and (d), the right and left subtrees, respectively, are null. For the binary tree in (e), all right subtrees are null; while for that in (f), each node either has two subtrees or none at all. Later, we will assign names to these and other types of binary trees.

Consider now the binary tree shown in Figure 6.2. Node A is the root of the binary tree. The left and right subtrees of node A are the binary trees rooted at nodes L and T, respectively. In turn, the left and right subtrees of node L are the binary trees rooted at G and I, respectively. And so on. Unlike natural trees, the root of a binary tree is at the top and the tree ‘grows’ downward.

The number of *non-null* subtrees of a node is called its *degree*. Thus, nodes A, L, T and G have degree 2; node H has degree 1; and nodes I, S, O, R and M have degree 0. A node with degree zero is called a *terminal node* or a *leaf*. A line joining two nodes is called a *branch*. The degree of a node is also the number of branches emanating from it.

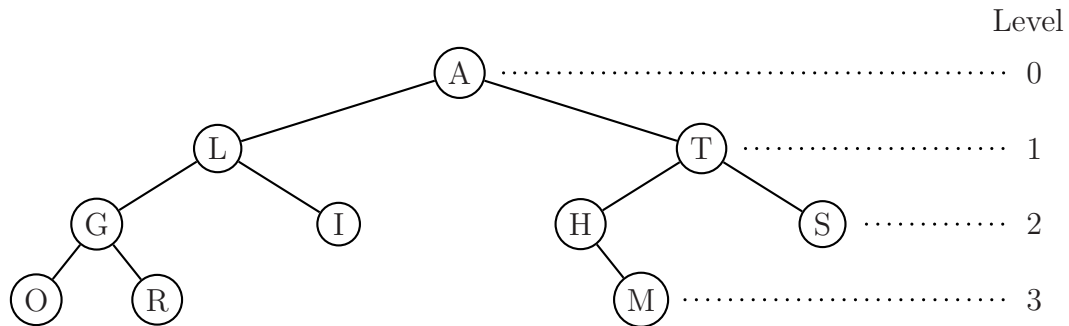


Figure 6.2 A typical binary tree

The *level* of a node is defined by specifying that the root of the binary tree is at level 0, the roots of its subtrees are at level 1, and so on. The *height* of a binary tree is the level of the bottommost node(s); thus the height of the binary tree in Figure 6.2 is 3.

A sequence of branches emanating from some node, say node X, and terminating in some other node, say node Y, is a *path* from node X to node Y, and this path is unique to X and Y. The *length of a path* is the number of branches in it; thus the length of the path from A to R is 3. The height of a binary tree may also be defined as the length of the *longest* path from the root to a leaf.

In addition to *botanical* terms (tree, root, leaf, branch) and *spatial* terms (left, right, top, bottom), we will also use *familial* terms (father, son, brother, descendant, ancestor) in describing binary trees and operations on them. In describing algorithms on binary trees we will freely mix metaphors involving these three classes of terms, as in the phrase *left son of the root*; only in CS will such a phrase make sense. [Note: We choose masculine over feminine or neutral terms for only one reason: ‘son’ is shorter than either ‘daughter’ or ‘child’.]

In the binary tree of Figure 6.2, node A is the *father* of nodes L and T; nodes L and T are the *sons* of node A. Nodes L and T are, of course, *brothers*. Nodes G, I, O and R are the *descendants* of node L; nodes A, T and H are the *ancestors* of node M.

Some operations on binary trees

Unlike the stack and the queue for which only a few operations are defined, there is for the binary tree a whole repertoire of possible operations. Some operations pertain to the entire binary tree, while others pertain to individual nodes only. Some operations are generic in that they apply to all types of binary trees, others are specific to a certain species only, for instance, an AVL tree. Here is a list of some common operations; others will be introduced later in their proper setting.

1. Test if a binary tree is empty.
2. Construct (or ‘grow’) a binary tree.
3. Find the size (number of nodes) of a binary tree.
4. Find the height of a binary tree (height is often more important than size).

146 SESSION 6. Binary Trees

5. Make a duplicate copy of a binary tree.
6. Test if two binary trees are equivalent. (Is one an exact replica of the other?)
7. Traverse a binary tree.
8. Find the leftmost and/or the rightmost node in a binary tree.
9. Insert a node into a binary tree.
10. Delete a node from a binary tree.
11. Replace a subtree in a binary tree by another subtree.
12. Given a node, find another node in the binary tree which bears a special relation to the given node.
13. And so on, and so forth.

Some properties of binary trees

1. The *maximum* number of nodes at level i of a binary tree is 2^i , $i \geq 0$.
2. The *maximum* number of nodes in a binary tree of height h is

$$n_{FBT} = \sum_{i=0}^h 2^i = 1 + 2 + 4 + 8 + \dots + 2^h = 2^{h+1} - 1 \quad (6.1)$$

This result follows from Eq.(2.29) of Session 2.

3. Let n_0 be the number of terminal nodes and n_2 be the number of nodes of degree 2 in a binary tree; then

$$n_0 = n_2 + 1 \quad (6.2)$$

Proof: Let n be the total number of nodes, n_1 be the number of nodes of degree 1 and n_B be the number of branches. Then:

- (a) $n = n_0 + n_1 + n_2$ (obviously)
 $= n_B + 1$ (since each node, except the root, has a branch leading to it)
- (b) $= n_1 + 2n_2 + 1$ (since a node of degree 1 has one branch emanating from it,
 and a node of degree 2 has two)

Subtracting (b) from (a) and rearranging terms, we obtain $n_0 = n_2 + 1$. (Q.E.D.)

4. The number of distinct binary trees on n unlabelled nodes is

$$b_n = \frac{1}{n+1} \binom{2n}{n} = \frac{1}{n+1} \frac{2n(2n-1)(2n-2) \cdots (n+1)}{n(n-1)(n-2) \cdots 1} \quad (6.3)$$

$$\approx \frac{4^n}{n^{\frac{3}{2}} \sqrt{\pi}} + O(4^n n^{-\frac{5}{2}}) \quad (6.4)$$

The derivation of Eq.(6.3) is beyond the scope of this book. The interested student may consult STANDISH[1980], pp. 54–57 or KNUTH1[1997], pp. 388–389 where a complete derivation is given. Figure 6.3 lists values of b_n for the first few values of n . Note how fast b_n increases with increasing n .

n	1	2	3	4	5	6	7	8	9	10
b_n	1	2	5	14	42	132	429	1430	4862	16796

Figure 6.3 Number of distinct binary trees on n unlabelled nodes

A nomenclature for binary trees

Certain types of binary trees are used in the analysis of the time complexity of a number of searching algorithms and also in the implementation of a particular sorting technique. Unfortunately, there is no agreement among various authors as to the names given to these binary trees. To add to the confusion, the same name is sometimes used to refer to two different types of binary tree. For our purposes, we will adopt, and consistently use, the following nomenclature for binary trees.

1. A *right (left)-skewed* binary tree is a binary tree in which every node has no left (right) subtree. The height of a left or right-skewed binary tree on n nodes is $n - 1$; this is the *maximum* possible height of a binary tree on n nodes.



Figure 6.4 Left- and right-skewed binary trees

2. A *strictly binary* tree is a binary tree in which every node has either two subtrees or none at all. The number of nodes in a strictly binary tree is always odd (you can readily see this by starting with one node and then constructing larger and larger strictly binary trees). Using Eq.(6.2), the following equations are easily established for a strictly binary tree: $n_0 = (n + 1)/2$ and $n_2 = (n - 1)/2$ (verify).

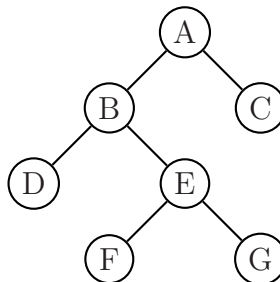


Figure 6.5 A strictly binary tree

3. A *full* binary tree is a strictly binary tree in which all terminal nodes lie at the bottommost level. Alternatively, a full binary tree may be defined as a binary tree with the full complement of nodes (1, 2, 4, 8, 16, ...) at *every* level. The maximum number of nodes given by Eq.(6.1) is attained in a full binary tree. The height of a full binary tree on n nodes is $\log_2(n+1) - 1$ or, more concisely, $\lfloor \log_2 n \rfloor$ (verify).

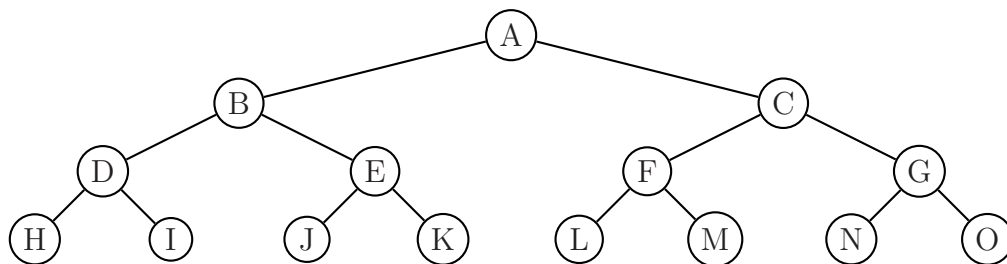


Figure 6.6 A full binary tree

4. A *complete* binary tree is a binary tree with the full complement of nodes at every level except possibly the bottommost, say level l ; in addition, the leaves at level l lie in the leftmost positions of l . In a complete binary tree leaves lie on at most two levels, viz., l and $l - 1$. If we start with a full binary tree and delete leaves right-to-left, bottom-up, the resulting trees will be complete binary trees. The height of a complete binary tree on n nodes is $\lfloor \log_2 n \rfloor$ (verify); this is the *minimum* possible height of a binary tree on n nodes.

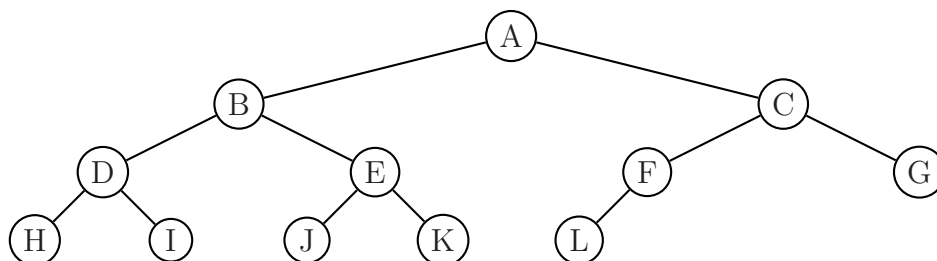


Figure 6.7 A complete binary tree

5. An *extended* binary tree is a binary tree in which null sons are indicated explicitly as square nodes. The circular nodes are called *internal nodes* and the square nodes are called *external nodes*.

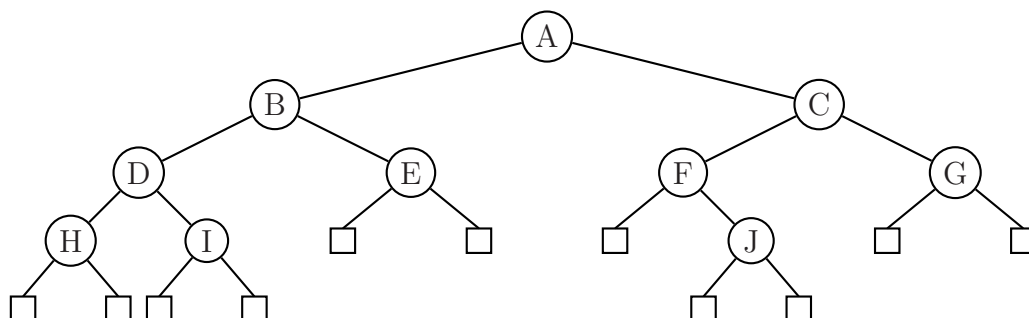


Figure 6.8 An extended binary tree

6.2 Binary tree traversals

Since a binary tree is structurally nonlinear, a sequential processing of the information contained in its nodes requires a procedure that *visits* the nodes of the binary tree in a linear fashion such that each node is visited exactly once. To visit a node means to perform the computations, not necessarily numerical, local to the node. Such a process is called a *traversal*.

In a binary tree, there are three distinct and disjoint entities: a root (which is any node), its left subtree (which may be null) and its right subtree (which may be null). Let us denote these entities as N, L and R, respectively. The order in which we process these entities, recursively, defines a particular traversal method. There are six different sequences: NLR, NRL, LNR, LRN, RNL and RLN. If we insist that L should always precede R, then we are left with three traversal orders: NLR-recursive or *preorder*, LNR-recursive or *inorder*, and LRN-recursive or *postorder*. The prefixes *pre*, *in* and *post* specify the position of the root N in the order of processing. The three traversal methods are defined as follows (see KNUTH1[1997], p. 319):

DEFINITION 6.2. If the binary tree is null consider the traversal as ‘done’; else

Preorder traversal

- Visit the root.
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder.

Inorder traversal

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

Postorder traversal

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root.

Note that each of the above definitions is recursive. This is to be expected since a binary tree is itself a recursively defined structure. If, in the above definitions, we interchange ‘left’ and ‘right’, then we obtain three new traversal orders which we may call *converse preorder*, *converse inorder* and *converse postorder*.

Two other important traversal methods applied primarily on *complete* binary trees are

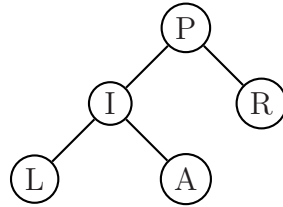
Level order traversal — visit the nodes top-to-bottom, left-to-right
and

Reverse level order traversal — visit the nodes bottom-up, right-to-left

We will find applications for these last two traversal methods in Session 7 in connection with *heaps* and in Session 13 in connection with *multiplicative binary search*. A generalization of level order traversal, called *breadth first search*, is the basis of a number of extremely important algorithms on graphs (more of this in Session 9).

150 SESSION 6. Binary Trees

Example 6.1. For the binary tree shown below, preorder, inorder and postorder traversal are performed as indicated.



1. Preorder traversal

Visit the root (**P**)

Traverse left subtree of P in preorder, i.e., subtree rooted at I

Visit the root (**I**)

Traverse left subtree of I in preorder, i.e., subtree rooted at L

Visit the root (**L**)

Traverse left subtree of L in preorder; subtree is null, hence 'done'

Traverse right subtree of L in preorder; subtree is null, hence 'done'

Traverse right subtree of I in preorder, i.e., subtree rooted at A

Visit the root (**A**)

Traverse left subtree of A in preorder; subtree is null, hence 'done'

Traverse right subtree of A in preorder; subtree is null, hence 'done'

Traverse right subtree of P in preorder, i.e., subtree rooted at R

Visit the root (**R**)

Traverse left subtree of R in preorder; subtree is null, hence 'done'

Traverse right subtree of R in preorder; subtree is null, hence 'done'

Preorder sequence is: **P I L A R**

2. Inorder traversal

Traverse left subtree of P in inorder, i.e., subtree rooted at I

Traverse left subtree of I in inorder, i.e., subtree rooted at L

Traverse left subtree of L in inorder; subtree is null, hence 'done'

Visit the root (**L**)

Traverse right subtree of L in inorder; subtree is null, hence 'done'

Visit the root (**I**)

Traverse right subtree of I in inorder, i.e., subtree rooted at A

Traverse left subtree of A in inorder; subtree is null, hence 'done'

Visit the root (**A**)

Traverse right subtree of A in inorder; subtree is null, hence 'done'

Visit the root (**P**)

Traverse right subtree of P in inorder, i.e., subtree rooted at R

Traverse left subtree of R in inorder; subtree is null, hence 'done'

Visit the root (**R**)

Traverse right subtree of R in inorder; subtree is null, hence 'done'

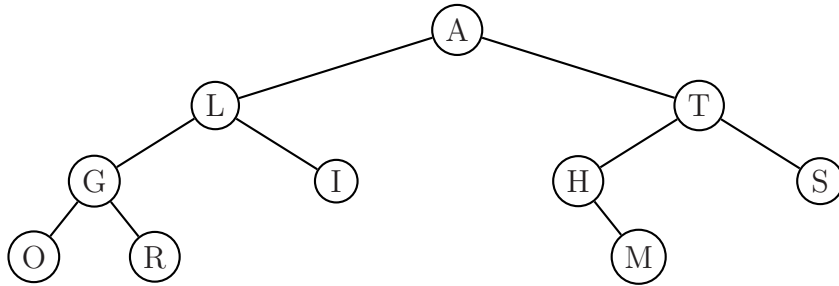
Inorder sequence is: **L I A P R**

3. Postorder traversal

Traverse left subtree of P in postorder, i.e., subtree rooted at I
 Traverse left subtree of I in postorder, i.e., subtree rooted at L
 Traverse left subtree of L in postorder; subtree is null, hence 'done'
 Traverse right subtree of L in postorder; subtree is null, hence 'done'
 Visit the root (**L**)
 Traverse right subtree of I in postorder, i.e., subtree rooted at A
 Traverse left subtree of A in postorder; subtree is null, hence 'done'
 Traverse right subtree of A in postorder; subtree is null, hence 'done'
 Visit the root (**A**)
 Visit the root (**I**)
 Traverse right subtree of P in postorder, i.e., subtree rooted at R
 Traverse left subtree of R in postorder; subtree is null, hence 'done'
 Traverse right subtree of R in postorder; subtree is null, hence 'done'
 Visit the root (**R**)
 Visit the root (**P**)

Postorder sequence is: **L A I R P**

Example 6.2. The preorder, inorder and postorder sequences for the binary tree shown below are as indicated (verify).



Preorder sequence: **A L G O R I T H M S**

Inorder sequence: **O G R L I A H M T S**

Postorder sequence: **O R G I L M H S T A**

The converse preorder, converse inorder and converse postorder sequences for same binary tree are as indicated (verify).

Converse preorder sequence: **A T S H M L I G R O**

Converse inorder sequence: **S T M H A I L R G O**

Converse postorder sequence: **S M H T I R O G L A**

152 SESSION 6. Binary Trees

Example 6.3. The binary tree shown below is the *expression tree* for the expression $A * (B + (C - D)/E^2) + F$. In this case the expression tree is a binary tree because all the operators used are binary operators. Expressions involving operators other than binary, for instance unary operators, are represented using *ordered trees*; more of this in Session 8. In an expression tree all the terminal nodes (leaves) contain operands (variables or constants), and all the internal nodes contain operators. Every non-null subtree represents a subexpression. Note that the expression tree in Figure 6.9 does not contain parentheses although the infix expression it represents does; this is because the order in which subexpressions are evaluated is implied by the structure of the tree. To evaluate an expression tree such as Figure 6.9, we apply the operator at the root to the values of its left and right subtrees; these values are obtained by applying the same process, recursively, to successively smaller subtrees. Using parentheses to indicate the successive subexpressions evaluated in this way we obtain $(A * (B + ((C - D)/(E^2)))) + F$, which is the same as the given expression, albeit overly parenthesized.

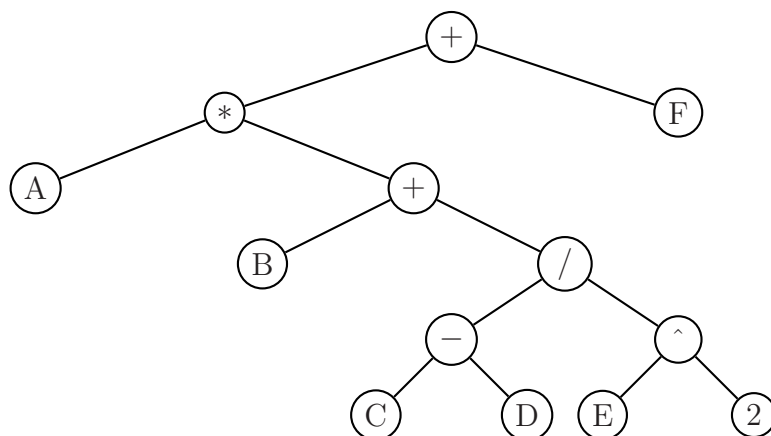


Figure 6.9 Expression tree for $A * (B + (C - D)/E^2) + F$

Traversing the binary tree in Figure 6.9 yields

Preorder sequence: $+ * A + B / - C D ^ E 2 F$

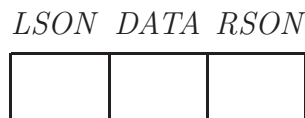
Inorder sequence: $A * B + C - D / E ^ 2 + F$

Postorder sequence: $A B C D - E 2 ^ / + * F +$

The preorder and postorder sequences are the prefix and postfix forms of the given infix expression (see Figure 4.8 of Session 4 for a similar result using algorithm POLISH). However, the inorder sequence is not equivalent to the given infix expression. This is because the given expression uses parentheses to override the predefined priorities among the five binary operators, but these parentheses are not present (because they are no longer relevant) in the expression tree. In general, a straight inorder traversal of a binary expression tree will not yield the correct infix expression; however, an algorithm based on inorder traversal which supplies the appropriate parentheses will.

6.3 Binary tree representation in computer memory

Thus far we have been looking at a binary tree in the abstract, independent of a specific representation. However, to implement the traversal and other algorithms on binary trees on a computer we need to represent a binary tree internally in computer memory. The most natural way to do this is to use the linked design. To this end, we will use a node with node structure



where the *LSON* and *RSON* fields contain pointers to the left and right subtrees of the node, respectively, and the *DATA* field contains the data local to the node. This representation mimics the way we draw a binary tree on paper. Using this representation, the binary tree of Figure 6.2 is depicted, thus:

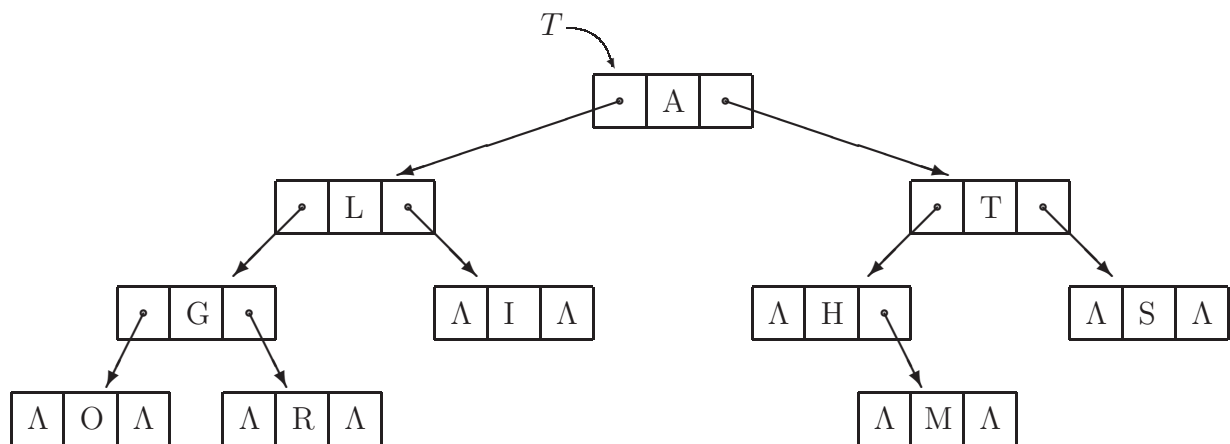


Figure 6.10 Linked representation of a binary tree

The pointer variable T points to the root node and, effectively, to the entire binary tree. Note that from the root node all nodes of the binary tree are accessible via the *LSON* and *RSON* links. In general, we will refer to a binary tree by the pointer to its root node, e.g., the binary tree of Figure 6.10 will be called binary tree T . The condition $T' = \Lambda$ means that the binary tree T' is empty.

For any given node this linked representation of a binary tree answers the question ‘Who are your sons?’ but not the question ‘Who is your father?’. We ask the first question as we climb down a binary tree (from the root to the leaves!), and we ask the second question when we climb up a binary tree. As we will see shortly, during traversals we will be climbing down and up a binary tree. In the absence of a *FATHER* link, we need to devise ways to locate fathers. One simple way to do this is to use a stack; other techniques, such as using *threads*, *link inversion*, etc. will also be discussed in this session.

Linked representation is by no means the only way to represent binary trees. In Session 7, we will consider an elegant sequential representation scheme for complete binary trees.

6.4 Implementing the traversal algorithms

We will now consider the implementation of the traversal algorithms as EASY procedures using the linked representation of a binary tree as depicted in Figure 6.10. The recursive versions of the procedures follow directly from the recursive definitions of the traversal orders (see DEFINITION 6.2); the non-recursive versions, though less concise, give further insight into the traversal process, which is of great value in understanding and developing algorithms on binary trees.

We will use the notation $\mathbb{B} = [(LSON, DATA, RSON), T]$ to denote the *data structure* such as that shown in Figure 6.10. This notation means that this particular implementation, \mathbb{B} , of a binary tree consists of two components, namely, a collection of linked nodes, each with node structure $(LSON, DATA, RSON)$, which comprise the binary tree and a pointer T to the root node of the binary tree. The pointer T is passed to the procedures which implement the various algorithms on a binary tree; this gives the implementing procedure access to the root node of the binary tree, and to the rest of its nodes via the $LSON$ and $RSON$ fields in each node.

6.4.1 Recursive implementation of preorder and postorder traversal

```

1  procedure PREORDER( $T$ )
2  if  $T \neq \Lambda$  then [ call VISIT( $T$ )
3                      call PREORDER( $LSON(T)$ )
4                      call PREORDER( $RSON(T)$ ) ]
5  end PREORDER

```

Procedure 6.1 Preorder traversal (recursive version)

```

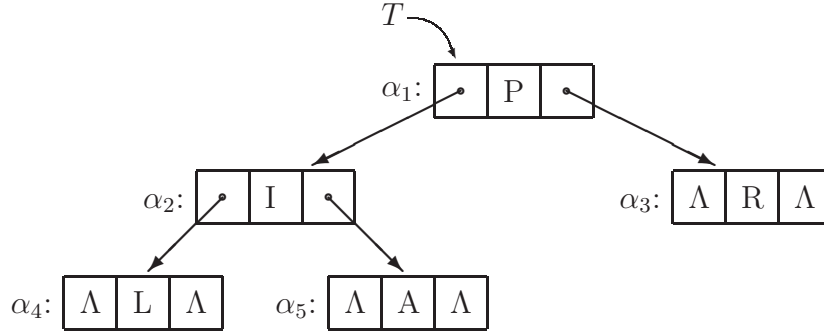
1  procedure POSTORDER( $T$ )
2  if  $T \neq \Lambda$  then [ call POSTORDER( $LSON(T)$ )
3                      call POSTORDER( $RSON(T)$ )
4                      call VISIT( $T$ ) ]
5  end POSTORDER

```

Procedure 6.2 Postorder traversal (recursive version)

Note that the procedures are essentially a straightforward transcription of the definitions of preorder and postorder traversal into EASY. The same applies to inorder traversal. In each of these procedures, VISIT is assumed to perform the computations local to the node pointed to by T .

Since this is our first encounter with recursive procedures on binary trees, it would be instructive to trace the actions of, say, procedure PREORDER when applied to a sample binary tree, such as the one shown below. We assume that VISIT in this instance simply outputs the label of the node that is visited.



call PREORDER(α_1)	$\triangleright \alpha_1 = T$
call VISIT(α_1)	\triangleright output 'P'
call PREORDER(α_2)	$\triangleright \alpha_2 = LSON(\alpha_1)$
call VISIT(α_2)	\triangleright output 'I'
call PREORDER(α_4)	$\triangleright \alpha_4 = LSON(\alpha_2)$
call VISIT(α_4)	\triangleright output 'L'
call PREORDER(Λ)	$\triangleright \Lambda = LSON(\alpha_4)$
call PREORDER(Λ)	$\triangleright \Lambda = RSON(\alpha_4)$
call PREORDER(α_5)	$\triangleright \alpha_5 = RSON(\alpha_2)$
call VISIT(α_5)	\triangleright output 'A'
call PREORDER(Λ)	$\triangleright \Lambda = LSON(\alpha_5)$
call PREORDER(Λ)	$\triangleright \Lambda = RSON(\alpha_5)$
call PREORDER(α_3)	$\triangleright \alpha_3 = RSON(\alpha_1)$
call VISIT(α_3)	\triangleright output 'R'
call PREORDER(Λ)	$\triangleright \Lambda = LSON(\alpha_3)$
call PREORDER(Λ)	$\triangleright \Lambda = RSON(\alpha_3)$

Figure 6.11 A trace of recursive procedure PREORDER

6.4.2 Iterative implementation of preorder and postorder traversal

Imagine traversing the binary tree in Figure 6.11 in preorder non-recursively. In contrast to the recursive implementation in which the runtime system does all the ‘housekeeping chores’, this time we need to do all the work. For instance, we need to maintain a ‘roving pointer’, say α , which will point from node to node in the course of the traversal. Initially α points to the root node, i.e., $\alpha = T$.

In preorder, we start by visiting the root node, i.e., node P; let us assume that to visit a node is simply to output its label, so we output ‘P’. Next we traverse the left subtree of node P in preorder, i.e., the subtree rooted at node I. To locate node I we simply set $\alpha \leftarrow LSON(\alpha)$; now α points to node I which we now visit, so we output ‘I’. Next we traverse the left subtree of node I in preorder, i.e., the subtree rooted at node L. Once again we locate node L by setting $\alpha \leftarrow LSON(\alpha)$. We visit node L by outputting ‘L’ and then we traverse its left subtree. This time setting $\alpha \leftarrow LSON(\alpha)$ sets α to Λ , so we do nothing and consider the traversal of the left subtree of node L as ‘done’. To complete the traversal of the subtree rooted at node L, we should now traverse L’s right subtree. But we have a problem: α now points to the null left son of node L, so how do we find node L itself? We need the address of node L so we can locate and traverse its right subtree.

We can rephrase the question more succinctly: How do we climb up a binary tree? Our representation of a binary tree in computer memory, as shown in Figure 6.10, allows us to descend (leftward via a *LSON* link and rightward via a *RSO*N link), but not to ascend a binary tree. In this session, we will look at four different ways by which we can climb up a binary tree, namely: (a) by using a stack (b) by using threads (c) by inverting links, and (d) by using the *exclusive or* operator.

We start with the stack. Note that when the runtime system executes the recursive procedure *PREORDER* it uses the runtime stack for precisely this purpose. To illustrate the use of a stack in a non-recursive setting, let us turn once more to the task of traversing the binary tree shown in Figure 6.11. We proceed exactly as already described above but with one additional step: after we visit a node but before we traverse its left subtree, we push its address (the current value of the pointer α) onto a stack. In effect we keep in the stack a record of the addresses of the nodes which we have visited so far and whose left subtrees we are yet in the process of traversing, so that later on, these nodes can be located and their right subtrees traversed in turn. Thus, in our present example, after traversing the left subtree of node L (by doing nothing because the subtree is null), we find the address of node L, α_4 , at the top of the stack. By popping the stack and retrieving the top element into α we have α pointing back to node L; effectively, we have climbed up the tree. By setting $\alpha \leftarrow RSON(\alpha)$ we locate the right subtree of node L and proceed to traverse it in preorder. And so on, yielding the preorder sequence ‘PILAR’.

Procedure *PREORDER* below formalizes in 14 lines of *EASY* code what we have described above in so many words. It utilizes an explicit stack \mathbb{S} and calls on procedures *InitStack*(\mathbb{S}), *IsEmptyStack*(\mathbb{S}), *PUSH*(\mathbb{S}, x) and *POP*(\mathbb{S}, x) of Session 4. How \mathbb{S} is implemented (array or linked) is irrelevant to *PREORDER*.

```

1  procedure PREORDER( $T$ )
  ▷ Traverses in preorder a binary tree represented using the data structure
  ▷  $[(LSO$ N,  $DATA$ ,  $RSO$ N),  $T$ ].  $T$  points to the root node of the binary tree.
  ▷ VISIT performs the computations local to a node.
2  call InitStack( $\mathbb{S}$ )
3   $\alpha \leftarrow T$ 
4  loop
5    while  $\alpha \neq \Lambda$  do
6      call VISIT( $\alpha$ )
7      call PUSH( $\mathbb{S}, \alpha$ )
8       $\alpha \leftarrow LSON(\alpha)$ 
9    endwhile
10   if IsEmptyStack( $\mathbb{S}$ ) then return
11   else [ call POP( $\mathbb{S}, \alpha$ );  $\alpha \leftarrow RSON(\alpha)$  ]
12 forever
13 end PREORDER
```

Procedure 6.3 Preorder traversal (iterative version)

The simplest way to understand how Procedure 6.3 works is to trace its actions, using pencil and paper, on a sample binary tree. Do that now and verify that indeed it visits the nodes in preorder and that the traversal is complete once the stack is empty in line 10. For the binary tree in Figure 6.11, verify that the stack will successively have the following states: $\text{empty} \Rightarrow (\alpha_1) \Rightarrow (\alpha_1, \alpha_2) \Rightarrow (\alpha_1, \alpha_2, \alpha_4) \Rightarrow (\alpha_1, \alpha_2) \Rightarrow (\alpha_1) \Rightarrow (\alpha_1, \alpha_5) \Rightarrow (\alpha_1) \Rightarrow \text{empty} \Rightarrow (\alpha_3) \Rightarrow \text{empty}$ (algorithm terminates).

If you followed our suggestion, you would have noticed that each node in the binary tree is encountered twice: first, as a father to its left son (at which point the node is visited and then stacked), and subsequently as a father to its right son (when the node is unstacked after traversal of its left subtree is completed and traversal of its right subtree commences). Assuming that VISIT takes constant time, the time complexity of Procedure 6.3 is therefore $O(n)$, where n is the number of nodes in the binary tree.

By transferring the **call** VISIT(α) statement in Procedure 6.3 from its position immediately before the call to PUSH to the position immediately after the call to POP, we obtain the iterative version of *inorder* traversal. Perform this simple change, call the new procedure INORDER, trace its actions on your sample binary tree and verify that the procedure does visit the nodes in inorder and thereafter terminates. Unlike the case of inorder traversal, the iterative procedure to implement *postorder* traversal requires more than just a strategic repositioning of the call to VISIT, as the following observations indicate.

In preorder and inorder traversal, a node is encountered twice: (a) on going down leftward as its left subtree is traversed, at which point it is visited and placed in the stack (if preorder) or simply placed in the stack (if inorder), and (b) on going up from the left after its left subtree has been traversed, at which point it is unstacked and its right subtree traversed (if preorder) or unstacked, visited and its right subtree traversed (if inorder). In postorder traversal, each node is encountered thrice: (a) on going down leftward as its left subtree is traversed, at which point it is stacked; (b) on going up from the left after its left subtree has been traversed, at which point it is unstacked (so that its right subtree can be located) and then placed in the stack anew (so it can be visited later); and (c) on going up from the right after its right subtree has been traversed, at which point it is unstacked the second time and finally visited. In short, for preorder and inorder traversal a node is stacked-unstacked once, whereas for postorder traversal a node is stacked-unstacked twice. Thus, for postorder traversal, we must have a way of distinguishing between the first and second instances that a node is placed in the stack, say, by using a *tag*.

The EASY procedure POSTORDER given below uses the ‘−’ and ‘+’ signs as tags. A node whose address is, say α , is placed in the stack for the first time as $-\alpha$, and its left subtree traversed. When on ascent the node is encountered at the top of the stack, the ‘−’ tag indicates that the node’s left subtree has just been traversed; the node is then placed in the stack for the second time as $+\alpha$ and its right subtree traversed. When on ascent the node is again encountered at the top of the stack, the ‘+’ tag indicates that the node’s right subtree has already been traversed and that the node can now be visited. Thereafter, the node leaves the stack.

```

1  procedure POSTORDER(T)
  ▷ Traverses in postorder a binary tree represented using the data structure
  ▷  $[(LSON, DATA, RSON), T]$ . T points to the root node of the binary tree.
  ▷ VISIT performs the computations local to a node.
2  call InitStack( $\mathbb{S}$ )
3   $\alpha \leftarrow T$ 
4  1: while  $\alpha \neq \Lambda$  do
5      call PUSH( $\mathbb{S}, -\alpha$ )
6       $\alpha \leftarrow LSON(\alpha)$ 
7  endwhile
8  2: if IsEmptyStack( $\mathbb{S}$ ) then return
9      else [ call POP( $\mathbb{S}, \alpha$ )
10             if  $\alpha < 0$  then [  $\alpha \leftarrow -\alpha$ 
11                                 call PUSH( $\mathbb{S}, \alpha$ )
12                                  $\alpha \leftarrow RSON(\alpha)$ 
13                                 go to 1 ]
14             else [ call VISIT( $\alpha$ )
15                     go to 2 ] ]
16 end POSTORDER

```

Procedure 6.4 Postorder traversal (iterative version)

In Procedure 6.4, the stack elements are ‘signed addresses’ in which the tag and the node’s address are treated as a single algebraic entity as, for instance, in lines 5 and 10. If we use the array implementation of the linked design in which addresses are actually indices into an array, then this approach is valid (the program will compile). However, if we use dynamic variables in which addresses are actual memory locations, this technique may not be legal. In such a case, the two should be considered as distinct components of a single unit, e.g., as structure elements of a C structure.

A C implementation of Procedure 6.4

To convert Procedure 6.4 into a C program, we will implement the node structure $(LSON, DATA, RSON)$ as a C structure of type, say `BinaryTreeNode`, with structure elements `LSON`, `DATA` and `RSON`. We define `LSON` and `RSON` to be pointers to `BinaryTreeNode`, and `DATA` to be of type, say, `char`.

We will use the stack routines given in Program 4.2 (array implementation of \mathbb{S}) or Program 4.4 (linked list implementation of \mathbb{S}) from Session 4, without any modification, except for `StackElemType`. Since, in this case, we need to store in the stack an address and a tag, we will define each stack element to be a structure of type `StackElemType`, whose components are `address` which is a pointer to `BinaryTreeNode`, and `tag` which is of type `char`.

To see whether our program works correctly, we must run it on test binary trees represented as indicated in Figure 6.9. To this end, we need to write a procedure, say `CREATE`, which creates a binary tree in computer memory and returns a pointer to it, say in `T`. There are various ways to do this. For instance, given the preorder and inorder sequences of the nodes of a binary tree, we can construct the binary tree uniquely. Before reading

6.4 Implementing the traversal algorithms 159

on, try constructing (by hand) the binary tree whose preorder and inorder sequences are ALGORITHMS and OGRLIAHMTS, respectively, and be convinced that this is the only binary tree defined by the given sequences.

In Program 6.1 given below the code for procedure CREATE is intentionally not given; instead, you are encouraged to write your own procedure.

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

typedef char BinaryTreeDataType;
typedef struct binarytreenode BinaryTreeNode;
struct binarytreenode
{
    BinaryTreeNode    * LSON;
    BinaryTreeDataType DATA;
    BinaryTreeNode    * RSON;
};
struct stackelemtype
{
    BinaryTreeNode * address;
    char           tag;
};
typedef struct stackelemtype StackElemType;

\* See Program 4.2 or 4.4 for declarations for the stack S * \

void CREATE(BinaryTreeNode * * );
void POSTORDER(BinaryTreeNode *T);
void VISIT(BinaryTreeNode *T);
void InitStack(Stack *);
int IsEmpty(Stack *);
void PUSH(Stack *, StackElemType);
void POP(Stack *, StackElemType *);
void StackOverflow(void);
void StackUnderflow(void);

main()
{
    BinaryTreeNode * T;
    clrscr();
    CREATE(&T);
    POSTORDER(T);
    return 0;
}
```

```

\* This procedure constructs the internal representation of a binary tree using the node *\
\* structure (LSON,DATA,RSON) and returns a pointer to it in T. *\
void CREATE(BinaryTreeNode * * T)
{
    \* Supply the code. *\

void POSTORDER(BinaryTreeNode *T)
{
    BinaryTreeNode *alpha; Stack S; StackElemType elem;
    InitStack(&S);
    alpha = T;
A: while(alpha != NULL)
    {
        elem.address = alpha;
        elem.tag = '-';
        PUSH(&S,elem);
        alpha = alpha->LSON;
    }
B: if(IsEmpty(&S)) return;
    else
    {
        POP(&S,&elem);
        alpha = elem.address;
        if(elem.tag == '-')
        {
            elem.tag = '+';
            PUSH(&S,elem);
            alpha = alpha->RSON;
            goto A;
        }
        else
        {
            VISIT(alpha);
            goto B;
        }
    }
}

void VISIT(BinaryTreeNode *T)
{
    printf("%c",T->DATA);
}

\* See Program 4.2 or 4.4 of Session 4 for the stack routines. These routines can be *\
\* stored in a separate file and included here using the #include directive. *\

```

Program 6.1 C implementation of non-recursive postorder traversal

6.4.3 Applications of the traversal algorithms to other operations on binary trees

Preorder, inorder and postorder traversal are often used as ‘templates’ for various other algorithms on binary trees, e.g., in constructing a binary tree, making a duplicate copy of a binary tree, determining whether two binary trees are exactly the same, and so on. The procedures given below, adapted from HOROWITZ[1976], pp. 234–235, should amply illustrate this point; they are also excellent examples of the power and elegance of recursion, and are good exercises for thinking recursively.

1. Making a duplicate copy of a binary tree

To make a copy of a binary tree, say T , represented as in Figure 6.10 (recall that we refer to a binary tree by the name of the pointer to its root node), we can use postorder traversal as follows: for every node, say node α , in T

- (a) Traverse the left subtree of node α in postorder and make a copy of it
- (b) Traverse the right subtree of node α in postorder and make a copy of it
- (c) Make a copy of node α and attach copies of its left and right subtrees

The following EASY procedure formalizes the idea.

```

1  procedure COPY( $T$ )
▷ Given a binary tree  $T$  represented by the data structure  $[(LSON, DATA, RSON), T]$ ,
▷ COPY constructs a copy of  $T$  and returns a pointer to the root node of the duplicate
▷ copy.
2   $\alpha \leftarrow \Lambda$ 
3  if  $T \neq \Lambda$  then [  $\lambda \leftarrow COPY(LSON(T))$ 
4                       $\rho \leftarrow COPY(RSON(T))$ 
5                      call GETNODE( $\alpha$ )
6                       $DATA(\alpha) \leftarrow DATA(T)$ 
7                       $LSON(\alpha) \leftarrow \lambda$ 
8                       $RSON(\alpha) \leftarrow \rho$  ]
9  return( $\alpha$ )
10 end COPY

```

Procedure 6.5 Making a duplicate copy of a binary tree

To invoke procedure COPY, we can write

$$S \leftarrow COPY(T)$$

Then S points to a binary tree which is an exact replica of T .

2. Determining whether two binary trees are equivalent

Two binary trees are equivalent if one is an exact duplicate of the other, such as binary trees S and T above. To determine whether two binary trees, say S and T , are equivalent, we can apply preorder traversal as follows: for every corresponding nodes, say node α in S and node β in T

- (a) Check whether node α and node β contain the same data
- (b) Traverse the left subtrees of node α and node β in preorder and check whether they are equivalent
- (c) Traverse the right left subtrees of node α and node β in preorder and check whether they are equivalent

If at any point during the traversal it is discovered that S and T are not equivalent, the test is terminated. The following EASY procedure formalizes the idea.

```

1  procedure EQUIVALENT( $S, T$ )
▷ Given two binary trees  $S$  and  $T$ , EQUIVALENT returns true if  $S$  and  $T$  are equivalent;
▷ else, it returns false.
2   $ans \leftarrow \mathbf{false}$ 
3  case
4      :  $S = \Lambda$  and  $T = \Lambda$  :  $ans \leftarrow \mathbf{true}$ 
5      :  $S \neq \Lambda$  and  $T \neq \Lambda$  : [  $ans \leftarrow (DATA(S) = DATA(T))$ 
6                                  if  $ans$  then  $ans \leftarrow EQUIVALENT(LSON(S), LSON(T))$ 
7                                  if  $ans$  then  $ans \leftarrow EQUIVALENT(RSON(S), RSON(T))$  ]
8  endcase
9  return( $ans$ )
10 end EQUIVALENT

```

Procedure 6.6 Determining whether two binary trees are equivalent

6.5 Threaded binary trees

Consider again the linked representation of a binary tree using the node structure $(LSON, DATA, RSON)$, such as the binary tree T in Figure 6.10. There are $n = 10$ nodes in T and $2n = 20$ link fields of which $n + 1 = 11$ are null. This means that more than half of the link fields are occupied simply by Λ . This result is true for any binary tree represented as in Figure 6.10, as proved below.

$$\begin{array}{c} \text{Number of null links} = 2n - (n - 1) = n + 1 \\ \begin{array}{ccc} \uparrow & \uparrow & \\ \text{total number of links} & \text{number of non-null links} & = \text{number of branches} \end{array} \end{array}$$

Is there some way by which the space occupied by the null links can be put to better use? The answer is 'Yes, there is.' Instead of storing a null link in the $LSON$ or $RSON$ field to indicate that a node, say node α , has no left son or no right son, respectively, we store a pointer to some other node in the binary tree which has some special relation to node α . We will call such a pointer a *thread*, and to distinguish a thread from a pointer to a bona fide son, we will employ a *tag bit*.

Figure 6.12 shows a threaded binary tree. The dashed arrows which point upward are threads, planted where a null link would have been. Except for the leftmost thread, a left thread points to a node's *inorder predecessor*, and, except for the rightmost thread, a right thread points to a node's *inorder successor*, yielding what is called an *inorder threaded binary tree*.

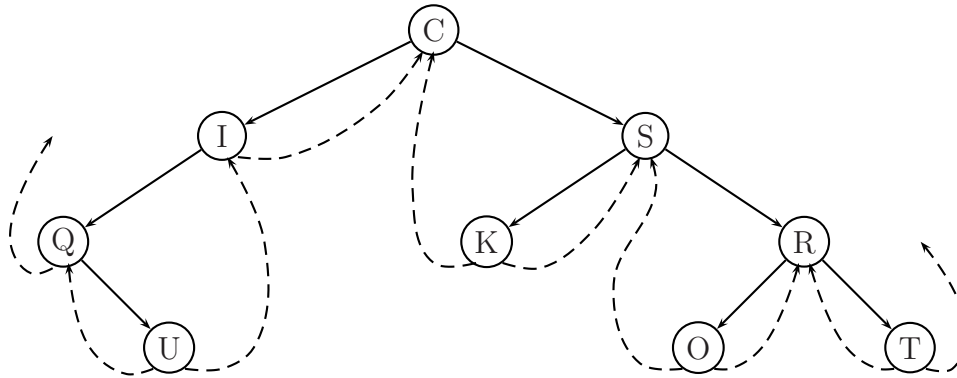
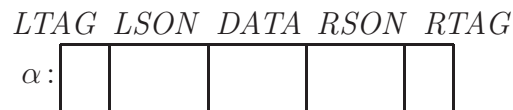


Figure 6.12 An inorder threaded binary tree

To represent an inorder threaded binary tree in computer memory, we will use the node structure



where

$LSON(\alpha)$ points to the left son of node α if it exists, in which case $LTAG(\alpha) = 1$; otherwise $LSON(\alpha)$ points to the inorder predecessor of node α , in which case $LTAG(\alpha) = 0$

$RSON(\alpha)$ points to the right son of node α if it exists, in which case $RTAG(\alpha) = 1$; otherwise $RSON(\alpha)$ points to the inorder successor of node α , in which case $RTAG(\alpha) = 0$

In order that the leftmost and rightmost threads are not left hanging with nothing to point to, we will introduce a *head node*, and make these threads point to the head node. We will adopt the convention whereby the root node of the binary tree is the left son of the head node and the head node is its own right son; this implies that the binary tree is *both* the left subtree and the right subtree of the head node! The reason for this seemingly convoluted setup will become clear shortly. Figure 6.13 shows the representation of the binary tree in Figure 6.12 according to these conventions (see KNUTH1[1997], p. 324).

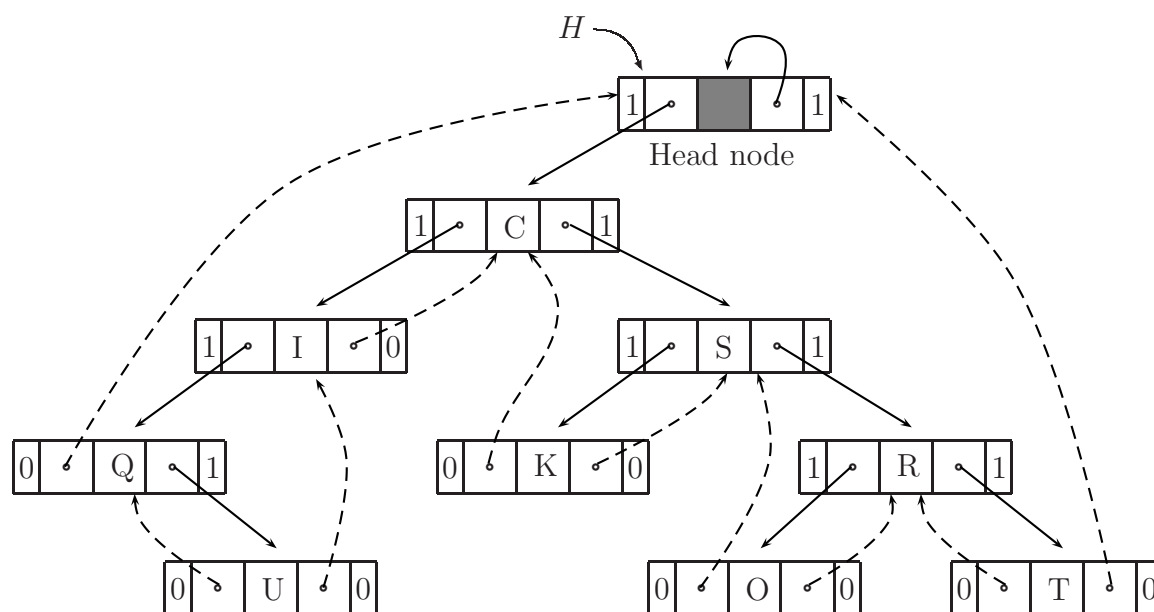


Figure 6.13 Linked representation of an inorder threaded binary tree

Note that with the introduction of a head node, the pointer to the binary tree now points to the head node rather than to the root of the binary tree. The condition which indicates that a binary tree is null is a head node with the *LSON* thread pointing to the head node, as shown below.

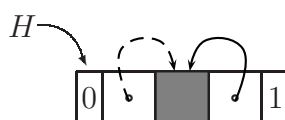


Figure 6.14 A null inorder threaded binary tree

6.5.1 Finding successors and predecessors in a threaded binary tree

In Section 6.2, we mastered quite easily how to enumerate the nodes of a binary tree in preorder, inorder and postorder. Let us now examine more closely the relationship between any two successive nodes when listed in these traversal sequences. Given some node, say node α , in a binary tree how do we find the inorder predecessor of node α ? What about its inorder successor? How do we locate its preorder predecessor, or its postorder successor? It turns out that these and other relatives of node α can be readily found in an inorder threaded binary tree without having to start the search from the root and without having to use a stack. This is essentially because the threads provide us with the means to climb up a binary tree.

Consider the problem of finding the inorder successor of some node in a binary tree given the node's address, say α . Let us call the inorder successor of node α as $insuc(\alpha)$. The following definition and accompanying figures should help us see more clearly the relationship between the two.

DEFINITION 6.3. Inorder successor of node α in a binary tree

- (a) If node α has a right subtree, then $insuc(\alpha)$ is the leftmost node of the right subtree of node α .
- (b) If node α has no right subtree, then $insuc(\alpha)$ is the root of the left subtree whose rightmost node is node α .

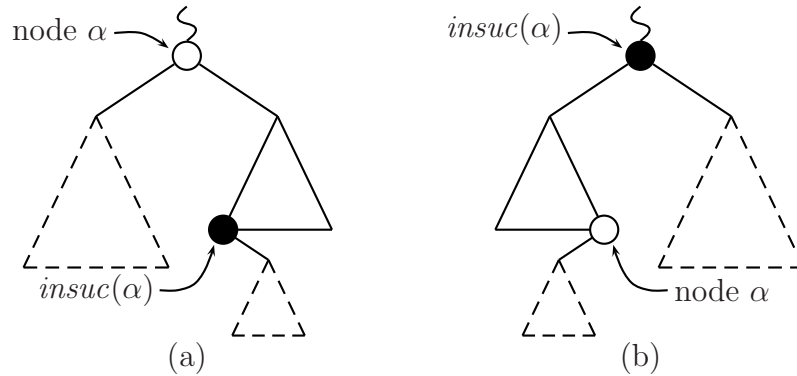


Figure 6.15 Finding the inorder successor

The EASY function INSUC accepts as input the address α of an arbitrary node in an inorder threaded binary tree and returns the address of its inorder successor.

```

1  procedure INSUC( $\alpha$ )
  ▷ Returns the address of the inorder successor of node  $\alpha$  in an inorder threaded
  ▷ binary tree.
2   $\beta \leftarrow RSON(\alpha)$ 
3  if RTAG( $\alpha$ ) = 1 then [ while LTAG( $\beta$ ) = 1 do
4                         $\beta \leftarrow LSON(\beta)$ 
5                        endwhile ]
6  return( $\beta$ )
7  end INSUC

```

Procedure 6.7 Finding the inorder successor in an inorder threaded binary tree

In line 2, β is the address of the right son of node α if it exists, otherwise β is the address of the inorder successor of node α . In the latter case, β is simply returned as the function value (line 6). In the former case, the leftmost node of the subtree rooted at node β is found (lines 3–5) and its address returned as the function value; this follows from part (a) of DEFINITION 6.3 as depicted in Figure 6.15(a).

Note that with threading, it becomes possible to locate the inorder successor of a node *knowing only the address of the node*. With an unthreaded binary tree such as T in Figure 6.10, this is not possible. We will have to start at the root and essentially traverse the binary tree in inorder (using a stack, say) in order to find the inorder successor of a given node.

If in DEFINITION 6.3 we replace ‘left’ by ‘right’ and ‘right’ by ‘left’ then we obtain the definition of the *inorder predecessor*, $inpred(\alpha)$, of node α in a binary tree. Applying the same transpositions on procedure INSUC yields procedure INPRED.

DEFINITION 6.4. Inorder predecessor of node α in a binary tree

- (a) If node α has a left subtree, then $\text{inpred}(\alpha)$ is the rightmost node of the left subtree of node α .
- (b) If node α has no left subtree, then $\text{inpred}(\alpha)$ is the root of the right subtree whose leftmost node is node α .

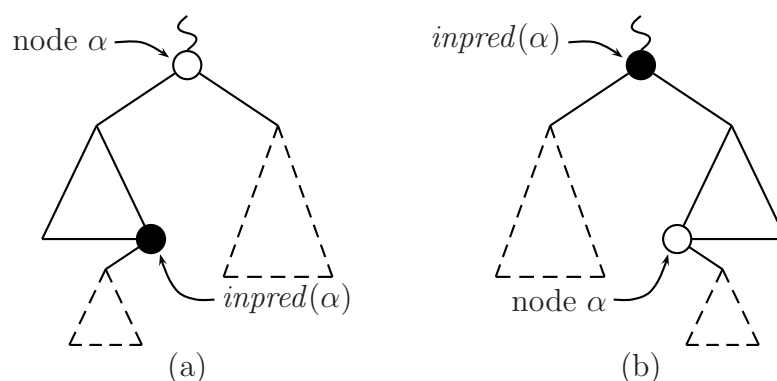


Figure 6.16 Finding the inorder predecessor

```

1  procedure INPRED( $\alpha$ )
▷ Returns the address of the inorder predecessor of node  $\alpha$  in an inorder threaded
▷ binary tree.
2   $\beta \leftarrow \text{LSON}(\alpha)$ 
3  if  $\text{LTAG}(\alpha) = 1$  then [ while  $\text{RTAG}(\beta) = 1$  do
4                         $\beta \leftarrow \text{RSON}(\beta)$ 
5                        endwhile ]
6  return( $\beta$ )
7  end INPRED

```

Procedure 6.8 Finding the inorder predecessor in an inorder threaded binary tree

Inorder threading not only allows us to readily locate the inorder relatives of a given node in a binary tree, but its preorder and postorder relatives as well. Consider, for instance, finding a node's *preorder successor*, which we may define as follows:

DEFINITION 6.5. Preorder successor of node α in a binary tree

- (a) If node α has a left subtree, then $\text{presuc}(\alpha)$ is its left son.
- (b) If node α has no left subtree, but has a right subtree, then $\text{presuc}(\alpha)$ is its right son.
- (c) If node α has neither a left nor a right subtree, then $\text{presuc}(\alpha)$ is the right son of the *nearest* root whose left subtree contains node α .

You may want to construct diagrams in the manner of Figures 6.15 and 6.16 to better see the relationships described in the above definition, particularly case (c). The following EASY function accepts as input a pointer to an arbitrary node in an inorder threaded binary tree and returns a pointer to its preorder successor.

```

1  procedure PRESUC( $\alpha$ )
  ▷ Returns the address of the preorder successor of node  $\alpha$  in an inorder threaded
  ▷ binary tree.
2  if LTAG( $\alpha$ ) = 1 then return (LSON( $\alpha$ ))
3      else [  $\beta \leftarrow \alpha$ 
4              while RTAG( $\beta$ ) = 0 do
5                   $\beta \leftarrow$  RSON( $\beta$ )
6              endwhile ]
7      return(RSON( $\beta$ ) ]
8  end PRESUC

```

Procedure 6.9 Finding the preorder successor in an inorder threaded binary tree

Line 2 is a straightforward implementation of case (a) of DEFINITION 6.5. Otherwise (line 3), if node α has a right subtree, then the test in line 4 immediately evaluates to **false**, the **while** loop is skipped and the address of node α 's right son is returned in line 7 (case (b)). Finally, lines 4–6 implement case (c) by using threads to climb up the binary tree in search of the nearest root, β , which has a right subtree; the right son of β is node α 's preorder successor.

Analogous procedures can be written to find other relatives in a binary tree, namely, the preorder predecessor, postorder predecessor and postorder successor of any given node α ; these are left as exercises. Of these six procedures the most useful are the three already given. These are normally utilized to implement other algorithms on binary trees, for instance, in constructing, duplicating and traversing inorder threaded binary trees. Likewise, locating the inorder predecessor and inorder successor of a node are basic operations on so called *binary search trees*; more of this in Session 14.

6.5.2 Traversing inorder threaded binary trees

With functions PRESUC, INSUC and POSTSUC available, implementing the traversal algorithms for an inorder threaded binary tree becomes a straightforward, almost trivial, task. For instance, to traverse a threaded binary tree in inorder, it is only necessary to call INSUC successively starting with the head node until we get back to the head node. Procedure INORDER_THREADED formalizes this idea.

```

1  procedure INORDER_THREADED( $H$ )
  ▷ Traverses an inorder threaded binary tree in inorder;  $H$  is the pointer to the head node.
2   $\alpha \leftarrow H$ 
3  loop
4       $\alpha \leftarrow$  INSUC( $\alpha$ )
5      if  $\alpha = H$  then return
6      else call VISIT( $\alpha$ )
7  forever
8  end INORDER_THREADED

```

Procedure 6.10 Inorder traversal of an inorder threaded binary tree

Despite its disarming simplicity, Procedure 6.10 has a number of interesting features which merit mention at this point.

1. In the *first* call to INSUC within the loop (line 4), the argument α points to the head node, and INSUC returns a pointer to the leftmost node of the binary tree (verify) which is the first node to be visited in inorder. This is a consequence of the convention whereby the head node is taken to be its own right son, thus making the binary tree the head node's right subtree (see Figure 6.13). From part (a) of DEFINITION 6.3, it follows that the inorder successor of the head node is the leftmost node of the binary tree.
2. Successive calls to INSUC visits the nodes in inorder, culminating in a last call with the argument α pointing to the rightmost node in the binary tree whose inorder successor, by convention, is the head node, and the procedure terminates.
3. There is no need for a special test to handle the case of an empty binary tree. This again is a consequence of the convention whereby an empty binary tree is represented by a head node with the left thread pointing to the head node (see Figure 6.14).
4. The procedure does not use a stack.
5. Each node is encountered at most twice: first as a left son in lines 3–5 of procedure INSUC, and subsequently when it is visited in line 6 of procedure INORDER_THREADED. Thus the procedure executes in $O(n)$ time.

Observations 1, 2 and 3 point to the fact that it is often possible to arrange things (in this case, conventions pertaining to the head node) so that our procedures work smoothly and special tests for special cases eliminated. We will see more applications of this idea in subsequent sessions.

Observation 4 is significant in that we have here an example of a *stackless traversal algorithm*. We will discuss other examples of such algorithms in Section 6.6. Such procedures are desirable because they execute in bounded workspace, in direct contrast to a procedure which utilizes a stack, which is said to use unbounded workspace (unbounded meaning unpredictable, not infinite).

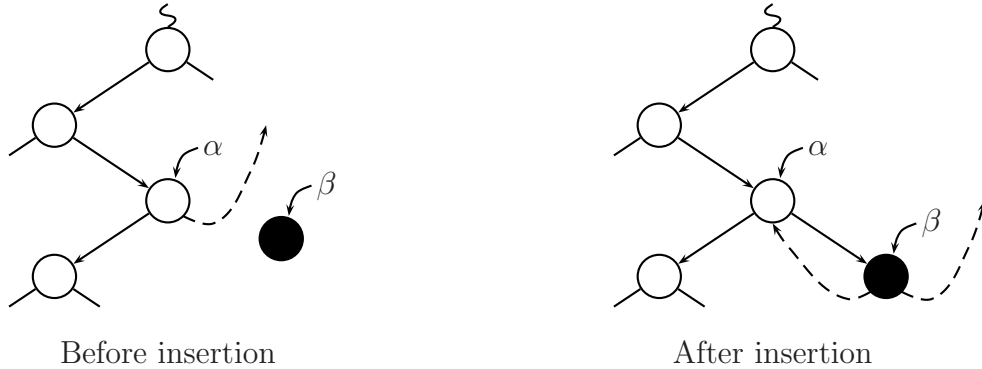
To traverse an inorder threaded binary tree in preorder, we simply replace INSUC by PRESUC in line 5 of Procedure 6.10 (and rename it as PREORDER_THREADED). An analogous procedure can be written for postorder traversal.

6.5.3 Growing threaded binary trees

We can grow a threaded binary tree one node at a time by inserting a node as the left son or the right son of another node in the binary tree. For instance, to construct a new binary tree we start with an empty binary tree as depicted in Figure 6.14. Next we insert the root node of the binary tree as the *left son* of the head node, following our conventions for an inorder threaded binary tree (see Figure 6.13). Subsequently, we can insert one node at a time as the left son or the right son of an already existing node. To accomplish these tasks we need two procedures, say INSERT_LEFT(α, β) and INSERT_RIGHT(α, β), to insert node β as the left son or right son, respectively, of node α .

The figure below depicts the process of inserting a new node β as the left son of node α in an inorder threaded binary tree. Two cases must be distinguished: (a) node α has no right son, and (b) node α already has a right son (or right subtree).

Case 1: Node α has no right son



Case 2: Node α has a right subtree

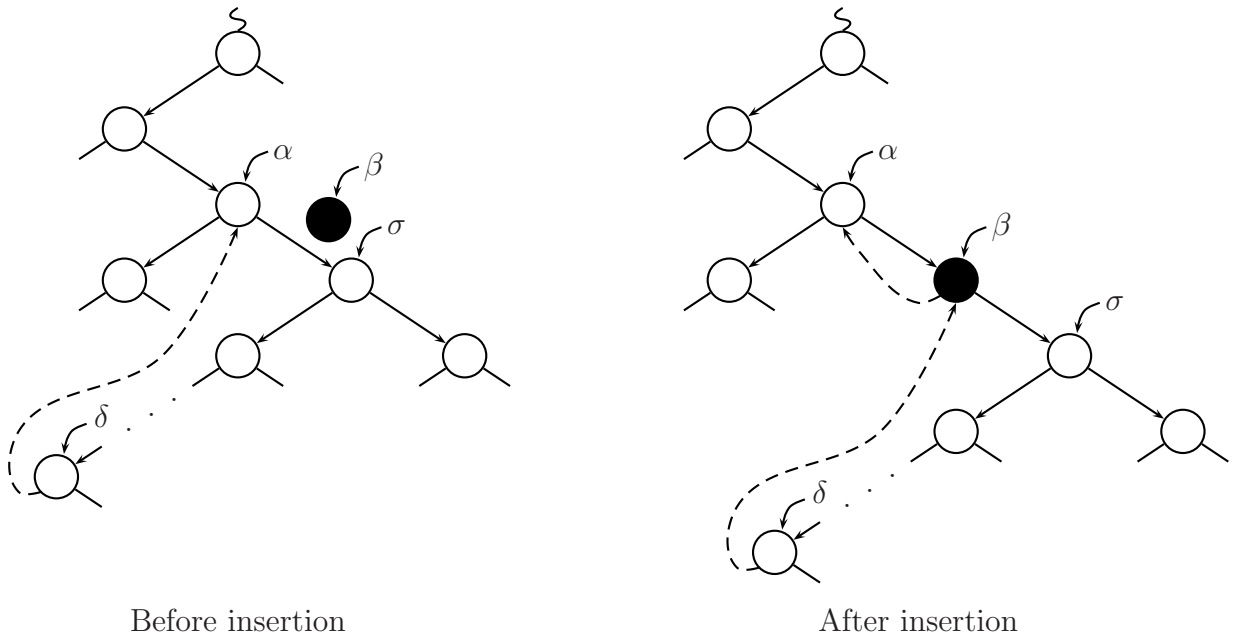


Figure 6.17 Inserting a new node into an inorder threaded binary tree

With these ‘before’ and ‘after’ diagrams as a guide, the process is readily coded as procedure INSERT_RIGHT.

```

1  procedure INSERT_RIGHT( $\alpha, \beta$ )
2   $RSN(\beta) \leftarrow RSN(\alpha)$ ;  $RTAG(\beta) \leftarrow RTAG(\alpha)$ 
3   $RSN(\alpha) \leftarrow \beta$ ;  $RTAG(\alpha) \leftarrow 1$ 
4   $LSN(\beta) \leftarrow \alpha$ ;  $LTAG(\beta) \leftarrow 0$ 
5  if  $RTAG(\beta) = 1$  then [  $\delta \leftarrow INSUC(\beta)$ ;  $LSN(\delta) \leftarrow \beta$  ]
6  end INSERT_RIGHT

```

Procedure 6.11 Inserting a new node into an inorder threaded binary tree

An analogous procedure to insert a new node β as the left son of node α is obtained by simply interchanging left and right and replacing INSUC by INPRED in the above procedure.

A closely related problem is that of *replacing* a subtree of the binary tree by another tree. For instance, if the binary tree represents an expression such as Figure 6.9, replacing a *leaf node* (which is an operand) with another tree is the equivalent of a *substitution operation*. If the binary trees are threaded the replacement requires resetting one link, three threads and one tag.

To illustrate, assume that node α is the father of the leaf node, say node β , to be replaced by a new binary tree. Assume further that H points to the head node of this replacement tree. Then we perform the following operations to effect the replacement.

1. Make the root node of H the left son of node α if node β is a left son; otherwise, make the root node of H the right son of node α .
2. Make the inorder predecessor of node β the inorder predecessor of the leftmost node of H .
3. Make the inorder successor of node β the inorder successor of the rightmost node of H .
4. Make H null and free node β .

The EASY procedure REPLACE_RIGHT accepts a pointer α to the father of the leaf node to be replaced (which is a right son) and a pointer H to the head node of the binary tree which is to replace the leaf node, and performs the replacement as described above. Both expression trees are assumed to be threaded in inorder.

```

1  procedure REPLACE_RIGHT( $\alpha, H$ )
2   $\lambda \leftarrow \text{INSUC}(H)$             $\triangleright$  Find leftmost node of  $H$ 
3   $\rho \leftarrow \text{INPRED}(H)$         $\triangleright$  Find rightmost node of  $H$ 
4   $\beta \leftarrow \text{RSN}(\alpha)$ 
5   $\text{RSN}(\alpha) \leftarrow \text{LSN}(H)$ 
6   $\text{LSN}(\lambda) \leftarrow \alpha$ 
7   $\text{RSN}(\rho) \leftarrow \text{INSUC}(\beta)$ 
8   $\text{LSN}(H) \leftarrow H; \text{LTAG}(H) \leftarrow 0$ 
9  call RETNODE( $\beta$ )
10 end REPLACE_RIGHT

```

Procedure 6.12 Substitution operation in a threaded binary expression tree

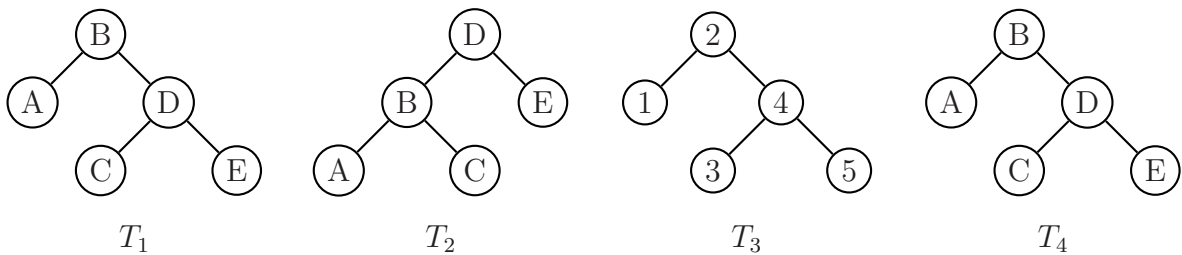
The leftmost and rightmost nodes of H are readily found by invoking INSUC and INPRED with argument H (lines 2 and 3); this follows from DEFINITION 6.3(a) and DEFINITION 6.4(a), respectively, and from our conventions with respect to the head node of an inorder threaded binary tree. There is no need to invoke $\text{INPRED}(\beta)$ in line 6 because the inorder predecessor of a *terminal* right son is its own father, in this case, node α . Lines 8 and 9 perform the final housekeeping tasks: making binary tree H empty and returning node β to the memory pool.

Constructing before and after diagrams in the manner of Figure 6.17 should help you to better understand how Procedure 6.12 works and to write the companion procedure REPLACE_LEFT.

6.5.4 Similarity and equivalence of binary trees

Another use of the *LTAG* and *RTAG* fields of a threaded binary tree is in determining whether two binary trees are *similar* or *equivalent*.

Two binary trees are similar if they have the same topology (same size and shape) and they are equivalent if, in addition, corresponding nodes contain the same information. Thus binary trees T_1 and T_2 are not similar (and therefore not equivalent), T_1 and T_3 are similar (but not equivalent) while T_1 and T_4 are equivalent.



Our minds can readily discern similarity and equivalence of any two given binary trees; a computer needs a more formal approach to accomplish the same thing; for instance, a theorem.

THEOREM 6.1. Similarity and equivalence of two binary trees

Let the nodes of binary trees T and T' be $u_1, u_2, u_3, \dots, u_n$ and $u'_1, u'_2, u'_3, \dots, u'_n$, respectively, in *preorder*. Then T and T' are similar if and only if $n = n'$ and $LTAG(u_j) = LTAG(u'_j)$ and $RTAG(u_j) = RTAG(u'_j)$, $1 \leq j \leq n$. T and T' are equivalent iff, in addition, $DATA(u_j) = DATA(u'_j)$, $1 \leq j \leq n$.

It is intuitively obvious that two sequences of 0's and 1's denoting *LTAG*'s and *RTAG*'s in preorder define a unique binary tree. Try, for instance, constructing the binary tree whose *LTAG* and *RTAG* sequences in preorder are 1 1 0 1 1 0 0 1 0 0 and 1 1 0 0 0 0 1 0 1 0 and you will be convinced that the binary tree you constructed is the only one defined by the given sequences. It follows from this that two binary trees having the same *LTAG* and *RTAG* sequences in preorder are similar.

Note that two binary trees which have the same *LTAG* and *RTAG* sequences in *inorder* are *not* necessarily similar. Using proof by counterexample, note that binary trees T_1 and T_2 shown above have the same *LTAG* and *RTAG* sequences in inorder, namely, 0 1 0 1 0 and 0 1 0 1 0, but they are obviously not similar. [What about postorder?]

To implement the theorem, we simply traverse both binary trees in preorder and compare the *LTAG*, *RTAG* and *DATA* fields of corresponding nodes. To test whether both trees have the same number of nodes, we merely check whether both traversals terminate at the same time. The EASY procedure EQUIVALENT formalizes the idea. You may want to compare this with Procedure 6.6 of the previous section.

```

1  procedure EQUIVALENT( $S, T$ )
  ▷ Determines if two threaded binary trees are equivalent and returns true if they are;
  ▷ else, returns false.  $S$  and  $T$  are the pointers to the head nodes.
2    $\alpha \leftarrow S; \beta \leftarrow T$ 
3   loop
4      $\alpha \leftarrow \text{PRESUC}(\alpha); \beta \leftarrow \text{PRESUC}(\beta)$ 
5     case
6       :  $\alpha = S$  : if  $\beta = T$  then return(true)
7               else return(false)
8       :  $\alpha \neq S$  : if  $\text{LTAG}(\alpha) \neq \text{LTAG}(\beta)$  or  $\text{RTAG}(\alpha) \neq \text{RTAG}(\beta)$ 
9                   or  $\text{DATA}(\alpha) \neq \text{DATA}(\beta)$  then return(false)
10    endcase
11  forever
12  end EQUIVALENT

```

Procedure 6.13 Determining whether two threaded binary trees are equivalent

6.6 Traversal by link inversion

We have seen that when we traverse a binary tree in any of the traversal orders, it is necessary to be able to subsequently locate the nodes we passed by as we descend the binary tree, so that they can be visited when their turn is up and/or their right subtrees traversed. The stack provided a simple way to do this, but as we already pointed out, the use of a stack implies unbounded workspace.

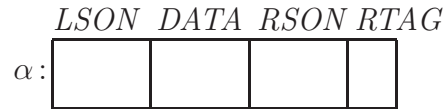
The use of threads in an inorder threaded binary tree eliminated the need for a stack when traversing the binary tree in preorder, inorder and postorder. In this and the next section, we will examine two other techniques for traversing a binary tree without the need for an auxiliary stack, viz., *traversal by link inversion* and *Siklóssy traversal*.

The primary idea behind traversal by link inversion is to leave an upward path by inverting links while descending the binary tree. Specifically, we invert a *LSON* link (such that it points to the father of a node rather than to its left son) as we traverse the left subtree of the node, and we invert a *RSON* link as we traverse the right subtree of the node. On ascent, after traversing the left or the right subtree of a node, the inverted link is used to locate a parent node, after which it is restored to its original function of pointing to a son. Thus, at any instant during the traversal process, some links will be inverted while the rest are in their normal state.

For any given node only one link, if any, is inverted at some point during the traversal process. It is necessary to identify which link this is, so that on ascent we will know which field, *LSON* or *RSON*, to use to locate the parent node. To this end, we will assume that each node has a *TAG* field which is initialized to 0. Subsequently, the tag is retained at 0 when a *LSON* link is inverted, but is set to 1 when a *RSON* link is inverted. The tag is reset to 0 when the *RSON* link is restored. Link inversion does away with the stack at the expense of an additional tag bit per node.

Binary tree representation for traversal by link inversion

We will use the node structure



where

$TAG(\alpha) = 0$ if $LSON(\alpha)$ points to the father of node α (in which case $RSON(\alpha)$ will be pointing to the bonafide right son of node α)

$TAG(\alpha) = 1$ if $RSON(\alpha)$ points to the father of node α (in which case $LSON(\alpha)$ will be pointing to the bonafide left son of node α)



Figure 6.18 shows a binary tree T that is set up for traversal by link inversion. Note that except for the additional TAG field in each node, this is the same as the basic linked representation of a binary tree as depicted in Figure 6.10.

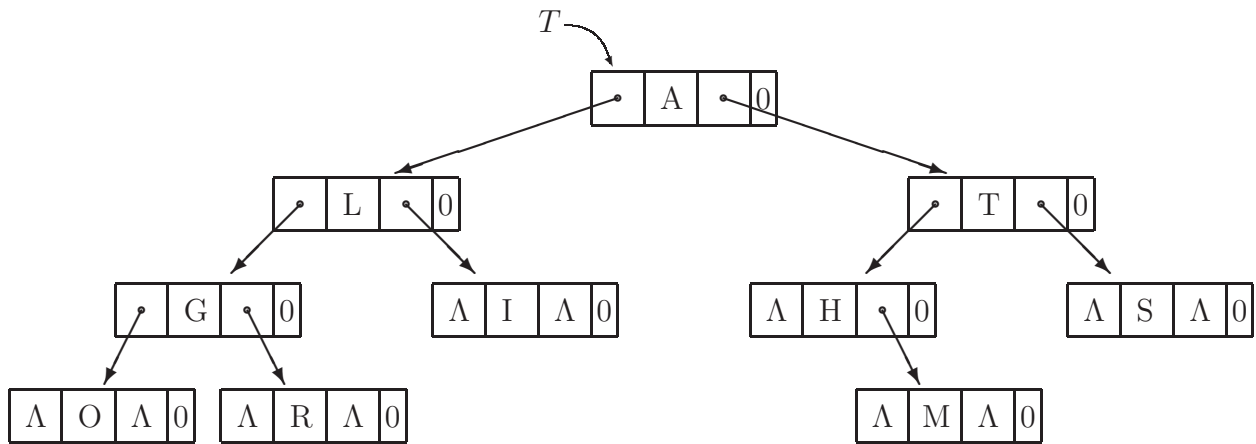


Figure 6.18 Binary tree representation for traversal by link inversion

Implementing binary tree traversals by link inversion

The procedure given below performs preorder traversal by link inversion. It utilizes three auxiliary pointer variables (and nothing else), namely α , β and σ , which consistently point to three generations of nodes as shown:

α : father of current node

β : current node

σ : left or right son of current node

On ascent, σ momentarily points to the grandfather of the current node (i.e., the father of node α), but soon afterwards, σ will again be pointing to a son.

Procedure 6.14 accepts a pointer T (line 1) to the root node of the binary tree to be traversed and represented as in Figure 6.18. When traversal commences the auxiliary pointer β points to the root node (line 4), which is the *current* node, and α points to the unknown father, represented by Λ , of the root node (line 3). Lines 5–9 mirror the basic steps in preorder traversal: visit, traverse left, traverse right. In line 7, the *LSON* link of the current node is inverted and made to point to its father ($LSON(\beta) \leftarrow \alpha$), after which leftward descent continues ($\alpha \leftarrow \beta$; $\beta \leftarrow \sigma$). In line 8, the *RSON* link of the current node is inverted and made to point to its father ($RSON(\beta) \leftarrow \alpha$), after which rightward descent continues ($\alpha \leftarrow \beta$; $\beta \leftarrow \sigma$). In addition, the tag bit of the current node is set to 1 to indicate that this time it is the *RSON* link which is inverted.

```

1  procedure PREORDER_BY_LINK_INVERSION( $T$ )
2  if  $T = \Lambda$  then return
3   $\alpha \leftarrow \Lambda$ 
4   $\beta \leftarrow T$ 
 $\triangleright$  Visit current node
5  1: call VISIT( $\beta$ )
 $\triangleright$  Descend left
6   $\sigma \leftarrow LSON(\beta)$ 
7  if  $\sigma \neq \Lambda$  then [  $LSON(\beta) \leftarrow \alpha$ ;  $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; go to 1 ]
 $\triangleright$  Descend right
8  2:  $\sigma \leftarrow RSON(\beta)$ 
9  if  $\sigma \neq \Lambda$  then [  $RSON(\beta) \leftarrow \alpha$ ;  $TAG(\beta) \leftarrow 1$ ;  $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; go to 1 ]
 $\triangleright$  Ascend
10 3: if  $\alpha = \Lambda$  then return
11      else if  $TAG(\alpha) = 0$  then [  $\sigma \leftarrow LSON(\alpha)$ 
12                                 $LSON(\alpha) \leftarrow \beta$ 
13                                 $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; go to 2 ]
14      else [  $\sigma \leftarrow RSON(\alpha)$ 
15               $RSON(\alpha) \leftarrow \beta$ ;  $TAG(\alpha) \leftarrow 0$ 
16               $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; go to 3 ]
17  end PREORDER_BY_LINK_INVERSION

```

Procedure 6.14 Preorder traversal by link inversion

Lines 10–16 handle ‘terminal’ conditions during the traversal. If upon ascent the pointer α is Λ , i.e., back to its initial value in line 3, then the traversal is done and a return is made to the calling program (line 10). Otherwise, only the traversal of the subtree rooted at the current node (not node T) is done. To proceed with the traversal, it is necessary to climb up the tree. If the subtree whose traversal has just been completed is a left subtree, then ascent is from the left (lines 11 and 13); otherwise, if it is a right subtree then ascent is from the right (lines 14 and 16). The procedure cleans up after itself by restoring the *LSON* and *RSON* fields to their original states (lines 12 and 15).

Procedure 6.14 is an exceptionally well-crafted procedure (the **go to**’s notwithstanding) whose inner workings the above brief description can’t fully capture. The best, and simplest, way to understand how the procedure works is to trace its action, by hand, on a typical binary tree using pencil and paper (and eraser). Then you will discover how each line of code fits beautifully into the whole scheme of preorder traversal by link inversion.

To obtain the equivalent procedures for inorder and postorder traversal, it is only necessary to transfer the call to VISIT to an appropriate position.

6.7 Siklóssy traversal

The main idea behind this technique is to represent a binary tree in read-only memory such that the address of a son (on descent) or a father (on ascent) can be found *by computation*, specifically, by using the *exclusive or* operator.

The exclusive or operator, denoted by \oplus , is defined by the following truth table:

\oplus	0	1
0	0	1
1	1	0

Let $\alpha = a_1a_2a_3 \dots a_n$ and $\beta = b_1b_2b_3 \dots b_n$ be bit strings of length n ; we define $\alpha \oplus \beta$ as the bitwise exclusive or of corresponding bits in α and β , i.e.,

$$\alpha \oplus \beta = (a_1 \oplus b_1)(a_2 \oplus b_2)(a_3 \oplus b_3) \dots (a_n \oplus b_n) \quad (6.5)$$

The following identities, which are the basis for Siklóssy traversal, are readily verified:

$$(\alpha \oplus \beta) \oplus \alpha = \beta \quad (6.6)$$

$$(\alpha \oplus \beta) \oplus \beta = \alpha \quad (6.7)$$

Binary tree representation for Siklóssy traversal

Each node of the binary tree will be represented using the node structure

	<i>LSON</i>	<i>DATA</i>	<i>RSON</i>	<i>RTAG</i>
α :				

where $TAG(\alpha) = 0$ means node α is a left son of its father and $TAG(\alpha) = 1$ means node α is a right son of its father. The root node is assumed to be the right son of its unknown father Λ . The contents of the *LSON* and *RSON* fields are defined as indicated below:

Father	α_1 :				
--------	--------------	--	--	--	--

Current node	α_2 :	$\alpha_1 \oplus \alpha_3$		$\alpha_1 \oplus \alpha_4$	
--------------	--------------	----------------------------	--	----------------------------	--

Sons	α_3 :				0	α_4 :				1
------	--------------	--	--	--	---	--------------	--	--	--	---

In other words, the *LSON* field of a node contains the exclusive or of the address of its father with the address of its left son, and the *RSON* field of a node contains the exclusive

or of the address of its father with the address of its right son. Figure 6.19 shows a binary tree represented according to these conventions.

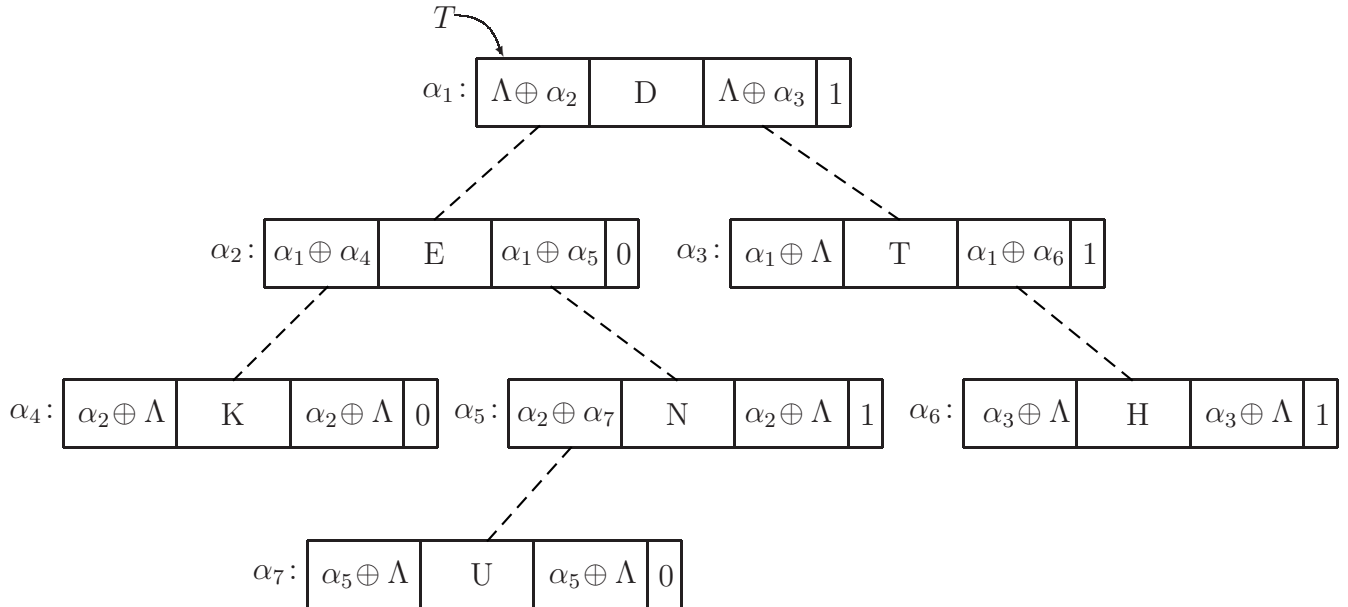
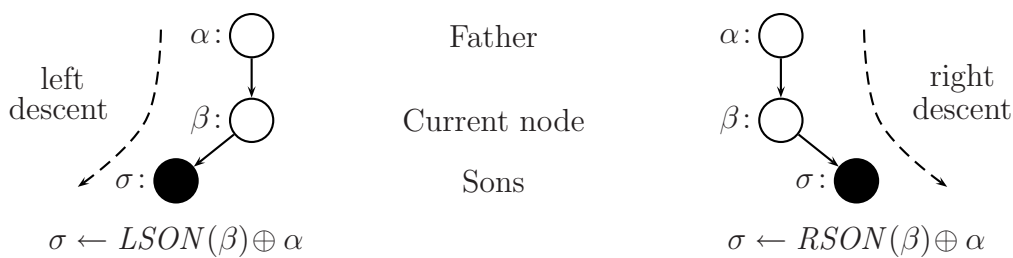


Figure 6.19 Binary tree representation for Siklóssy traversal

With the binary tree stored memory as depicted in Figure 6.19, the left son and right son of the current node are located by applying Eq.(6.6); on ascent, the grandfather of the current node is found by applying Eq.(6.7). It is this same grandfather node that we located using an inverted link in traversal by link inversion. The computations are best illustrated with diagrams as shown below.

(a) Locating the sons of a node



(b) Locating the grandfather of a node

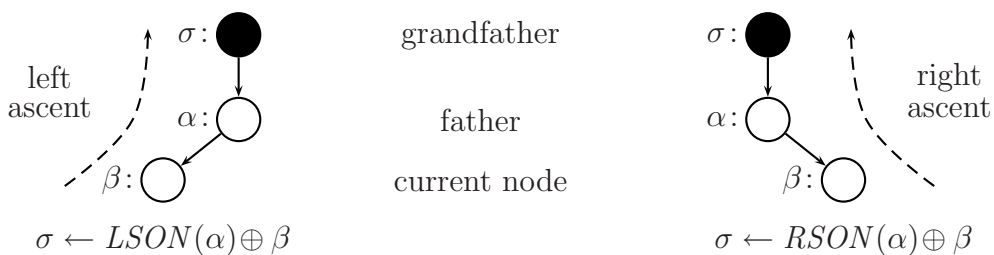


Figure 6.20 Climbing down and up a Siklóssy tree

Implementing Siklóssy traversal of a binary tree

The procedure given below performs preorder traversal of a binary tree represented as in Figure 6.19 using Siklóssy's technique. In a manner akin to traversal by link inversion, only three auxiliary pointer variables which point to three generations of nodes as indicated below are all that it takes to perform Siklóssy traversal.

α : father of current node
 β : current node
 σ : left or right son of current node

Note the striking similarity between Procedure 6.14 and Procedure 6.15; they differ only in the way sons and fathers are located—by following links (normal and inverted) in the former, and by computation using the exclusive or operator in the latter.

```

1  procedure PREORDER_SIKLÓSSY( $T$ )
2  if  $T = \Lambda$  then return
3   $\alpha \leftarrow \Lambda$ 
4   $\beta \leftarrow T$ 
▷ Visit current node
5  1: call VISIT( $\beta$ )
▷ Descend left
6   $\sigma \leftarrow LSON(\beta) \oplus \alpha$ 
7  if  $\sigma \neq \Lambda$  then [  $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; go to 1 ]
▷ Descend right
8  2:  $\sigma \leftarrow RSON(\beta) \oplus \alpha$ 
9  if  $\sigma \neq \Lambda$  then [  $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; go to 1 ]
▷ Ascend
10 3: if  $\alpha = \Lambda$  then return
11      else if  $TAG(\alpha) = 0$  then [  $\sigma \leftarrow LSON(\alpha) \oplus \beta$ ;  $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; go to 2 ]
12      else [  $\sigma \leftarrow RSON(\alpha) \oplus \beta$ ;  $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; go to 3 ]
13
14  end PREORDER_SIKLÓSSY

```

Procedure 6.15 Preorder traversal using Siklóssy's technique

Summary

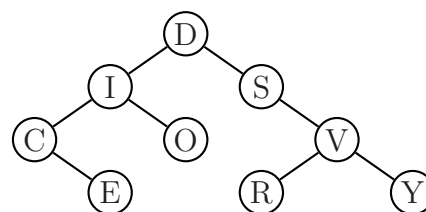
- The binary tree is an extremely versatile nonlinear ADT. It is utilized in a wide variety of applications, including searching (binary search trees, AVL trees, red-black trees), sorting (heaps in heapsort), efficient encoding of strings (Huffman coding tree), representation of algebraic expressions (expression trees), and so on. Binary trees are used to implement other ADT's such as priority queues and tables.
- Metaphorical language is often used in describing binary trees and algorithms on them, with words drawn from three sources of terminology, viz., botanical (tree, root, leaf, branch), spatial (left, right, bottom, up) and familial (father, son, brother, ancestor, descendant, etc.), as in the phrase 'left son of the root'.

- Binary trees are usually represented in computer memory using the linked design to mimic the way binary trees are depicted on the printed page. An outstanding exception is the complete binary tree, which is efficiently represented in computer memory using the sequential design.
- Since a binary tree is structurally nonlinear, a sequential processing of the information contained in its nodes requires a process that ‘visits’ the nodes of the binary tree in a linear fashion, where to visit a node means to perform the computations local to the node. Such a process is called a *traversal*.
- Three fundamental traversal sequences are preorder, inorder and postorder. Two other traversal sequences, usually defined on complete binary trees, are level order and reverse level order.
- The algorithms to perform preorder, inorder and postorder traversal can be implemented recursively, implicitly using the runtime stack, or iteratively by using an explicit stack.
- Alternative implementations of the algorithms to perform preorder, inorder and postorder traversal which do not require the use of a stack are: traversal using threads, traversal by link inversion and Siklóssy traversal. Stackless traversal procedures are desirable because they execute in bounded workspace.
- Threading provides an efficient way of locating special ‘relatives’ of a node given only its address, e.g., the node’s inorder successor, inorder predecessor, preorder successor, and so on.

Exercises

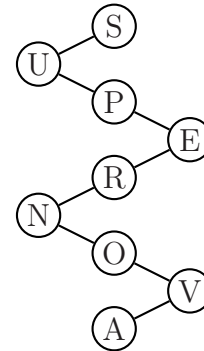
1. What is the number of distinct binary trees on 13 unlabeled nodes?
2. Prove that the minimum height of a binary tree on n nodes is $\lfloor \log_2 n \rfloor$.
3. The maximum height of a binary tree on n nodes is $n - 1$. What is the number of distinct binary trees on n unlabeled nodes whose height is $n - 1$?
4. Given a *strictly binary tree* on n nodes:
 - (a) What is the number of terminal nodes?
 - (b) What is the maximum height of the binary tree?
 - (c) What is the minimum height of the binary tree?
5. List the nodes of the binary tree shown in:

(a) preorder	(d) converse preorder
(b) inorder	(e) converse inorder
(c) postorder	(f) converse postorder



6. List the nodes of the binary tree shown in:

- | | |
|-----------------|-------------------------|
| (a) preorder | (e) converse preorder |
| (b) inorder | (f) converse inorder |
| (c) postorder | (g) converse postorder |
| (d) level order | (h) reverse level order |



7. Give all the binary trees whose nodes appear in exactly the same sequence in both:

- (a) preorder and inorder

- (b) preorder and postorder

- (c) inorder and postorder

8. Draw all the binary trees which have: (a) *exactly four* unlabeled nodes (b) *at most four* unlabeled nodes.

9. Knuth (KNUTH1[1997], p. 332) defines another traversal order called **double order** as follows: If the binary tree is empty, do nothing; otherwise

- visit the root, for the first time;
- traverse the left subtree, in double order;
- visit the root, for the second time;
- traverse the right subtree, in double order.

List the nodes of the binary trees in Items 3 and 4 in double order.

10. The preorder and inorder sequences of the nodes of a binary tree, together, uniquely define the binary tree. Construct the binary tree whose nodes in preorder and inorder are:

- KIGOHRNFEATS and GIHROKNEFTAS, respectively.
- DOUETRCMNAY and OERTUDMAYNC, respectively.
- SMNPADEROLI and PNAMDSRELOI, respectively.

11. The postorder and inorder sequences of the nodes of a binary tree, together, uniquely define the binary tree. Construct the binary tree whose nodes in postorder and inorder are:

- RKENWLDATHIOG and WREKNLGODTAIH, respectively.
- ZIHWRETUPMOCA and CMUEWIZHRTPOA, respectively.
- RFALUEPYTSNV and EFRULAVNYPST, respectively.

12. Prove that the preorder and postorder sequences of the nodes of a binary tree, together, do *not* uniquely define the binary tree.

13. Write an EASY procedure which takes as input the preorder and inorder sequences of the nodes of a binary tree and generates the linked representation of the binary tree in the manner of Figure 6.10.

14. Write an EASY procedure which takes as input the postorder and inorder sequences of the nodes of a binary tree and generates the linked representation of the binary tree in the manner of Figure 6.10.

180 SESSION 6. Binary Trees

15. Write an EASY procedure LEVELORDER(T) which traverses a binary tree in level order. Assume that T is represented as in Figure 6.10.
16. Draw the binary tree representation of the following arithmetic expressions.
 - (a) $A + X * (B + X * (C + X * D))$
 - (b) $A / (B + C) ^ D * E / (F - G) * H$
 - (c) $((A - B) * ((C + D) ^ 2 + F) / G ^ {1/2})$
17. Draw the inorder threaded representation of the binary trees shown in Item 5 and Item 6.
18. Construct the binary tree whose LTAG and RTAG sequences in preorder are:
 - (a) 1101100011000 and 1010100111000, respectively
 - (b) 101000001000 and 111101011100, respectively
19. Construct the binary tree whose LTAG and RTAG sequences in postorder are:
 - (a) 000110001101 and 001010001111, respectively
 - (b) 01010001011 and 00101010101, respectively.

In each case, is the binary tree uniquely defined by the given sequences?
20. Show *before* and *after* diagrams for the insertion of node α as the left son of node β in an inorder threaded binary tree and write an EASY procedure INSERT-LEFT(α, β) to perform the insertion.
21. Write an EASY procedure FATHER(α) which finds the father of node α in an inorder threaded binary tree.
22. Write an EASY procedure POSTSUC(α) which finds the postorder successor of node α in an inorder threaded binary tree. Hint: Use the result in Item 21.
23. Write an EASY procedure PREORDER_THREADED(H) to traverse an inorder threaded binary tree in preorder. Assume that the binary tree is represented in computer memory according to the conventions indicated in Figures 6.13 and 6.14.
24. Write an EASY procedure POSTORDER_THREADED(H) to traverse an inorder threaded binary tree in postorder. Assume that the binary tree is represented in computer memory according to the conventions indicated in Figures 6.13 and 6.14.
25. Modify (and rename) procedure PREORDER_BY_LINK_INVERSION(T) (Procedure 6.14) such that it traverses T in: (a) inorder (b) postorder.
26. Modify (and rename) procedure PREORDER_SIKLÓSSY(T) (Procedure 6.15) such that it traverses T in: (a) inorder (b) postorder.

Bibliographic Notes

Procedure PREORDER is an EASY implementation of Algorithm 3.5 (*Preorder stack traversal of binary tree*) given in STANDISH[1980], p. 75. An alternative rendering of postorder traversal, which does not require tagging the addresses stored in the stack (as we did in procedure POSTORDER) but uses another auxiliary pointer, is given in KNUTH1[1997], p. 331 (Exercise 13) and p. 565. Procedures COPY and EQUIVALENT are adopted from HOROWITZ[1976], pp. 234–235 where they are given as SPARKS procedures COPY and EQUAL, respectively.

Threaded binary trees, which were originally devised by A.J. Perlis and C. Thornton, are discussed in KNUTH1[1997], STANDISH[1980], TENENBAUM[1986], HOROWITZ[1976], among others. Procedure INSUC(α) is an EASY implementation of Algorithm S (*Symmetric (inorder) successor in a threaded binary tree*) given in KNUTH1[1997], p. 323. A detailed analysis of the algorithm shows that line 4 of INSUC(α) is executed only *once* on the average if node α is a random node of the binary tree. Procedure INSERT_RIGHT is an implementation of Algorithm I (*Insertion into a threaded binary tree*) from the same reference, p. 327. Procedure REPLACE_RIGHT is an implementation of Algorithm 3.3 (*Inserting a symmetrically threaded tree T_1 as the right subtree of a given node N in another symmetrically threaded tree T_2*) given in STANDISH[1980], p. 63.

Theorem 6.1 on the similarity and equivalence of two binary trees is given as Theorem A in KNUTH1[1997], p. 328, where a proof of its validity by induction on n (the size of the binary trees) is also given. The reader who is not satisfied with our informal ‘proof by construction’ of the theorem in section 6.5.4 is invited to take a look at this formal proof.

Procedure PREORDER_BY_LINK_INVERSION is an implementation of Algorithm 3.6 (*Link inversion traversal of binary tree*) given in STANDISH[1980], p.76. The algorithm as given actually considers all three traversal orders (preorder, inorder and postorder) by indicating when to perform the visit. Procedure PREORDER_SIKLÓSSY is an implementation of Algorithm 3.9 (*Siklóssy traversal of a binary tree*), also from STANDISH[1980], p.82. Other methods for enumerating the nodes of a binary tree are Robson traversal and Lindström scanning. A description of the algorithms can be found in the same reference as Algorithm 3.7 (*Robson traversal of a binary tree*) and Algorithm 3.8 (*Lindström scanning of a binary tree*).

NOTES

SESSION 7

Applications of binary trees

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Construct the binary tree representation of a given network of irrigation canals.
2. Explain implementation issues pertaining to the linked representation of a heap.
3. Explain implementation issues pertaining to the array representation of a heap.
4. Apply the heapsort algorithm on a given set of keys.
5. Explain implementation issues pertaining to the various array representations of a priority queue.
6. Assess the suitability of using a binary tree to solve a given problem on a computer.

READINGS STANDISH[1980], pp. 84–92; CORMEN[2001], pp. 127–140; KNUTH3[1998], pp. 144–155; WEISS[1997], pp. 177–192, 226–230.

DISCUSSION

In this session we will look at three applications of a binary tree. The first application is to a water resources engineering problem: that of *calculating the conveyance losses in an irrigation network*. To this end, we will utilize a binary tree represented using linked allocation to model the problem. The second application is to a vintage CS problem: that of *sorting keys on which total order has been defined*. To this end, we will define a new ADT called a *heap* which in structure is a complete binary tree. For reasons that will become clear later, we will use this time a sequential array to represent the binary tree. The third application has to do with the implementation of a particularly useful ADT called a *priority queue*. While an ordinary list can be utilized to realize a priority queue, a truly elegant and efficient implementation is obtained with the use of a heap.

7.1 Calculating conveyance losses in an irrigation network

In the mid 70's of the last century, the author was a member of a small team of faculty members from the U.P. College of Engineering tasked by the National Irrigation Administration (NIA) through the National Hydraulic Research Center to 'design and implement computerized diversion control programs for the Upper Pampanga River Project'. The UPRP centered on the management of water that was intended to irrigate an area of approximately 83,000 hectares of agricultural land in Nueva Ecija, with the water for irrigation coming mainly from the Pantabangan Dam located in San Jose City of the same province. One component of the Project was to determine the amount of water to be released at certain control points, called diversion dams, in the network: too much and precious water is wasted, too little and crop failure may result.

Given the amount of water required by the crops to be irrigated, it is clear that the amount of water that must be released at the source must be equal to the amount needed *plus the amount of water lost* (due to seepage, evaporation, and other causes) in delivering this water through canals from the source to the farmlands. Thus the problem reduces to calculating these so-called *conveyance losses* in an irrigation network.

The problem

Figure 7.1 depicts schematically a network of irrigation canals which services a certain irrigation area. A *segment* of a canal, such as segment *F*, irrigates the farmlands adjacent to it and the amount of water needed to accomplish this is called the *farm irrigation requirement*, or *FIR*. The inflow at the upstream end of a canal segment, such as section *a – a* of segment *F*, is called the *irrigation diversion requirement*, or *IDR*, and is the sum of the *FIR*'s for all areas downstream of the particular section, plus the conveyance losses incurred in delivering this requirement.

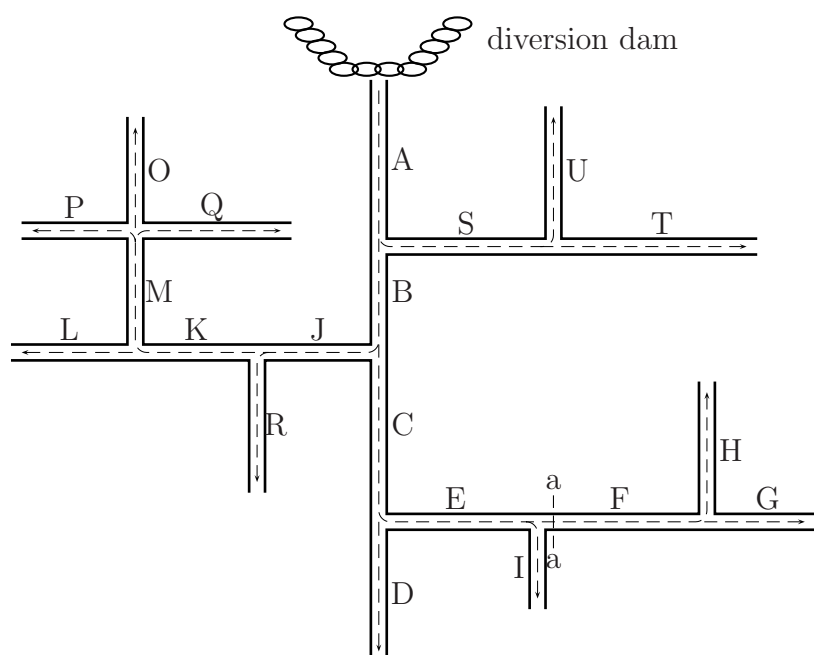


Figure 7.1 An irrigation network (top view)

Experiments conducted by the Agricultural Development Division of the NIA showed that the conveyance loss along an unlined canal could be expressed roughly as a fixed fraction k of the flow Q through the canal. Our task was to construct a mathematical model to determine whether this empirical result would yield reasonable estimates for the conveyance losses.

Consider Figure 7.2 which depicts the flow along a typical canal segment. The *inflow* at the upstream end, Q_i , can be calculated if the *FIR* supplied by the segment and the *outflow* at the downstream end, Q_o , of the segment are known. Assuming that water is uniformly withdrawn over the length of the canal segment to satisfy the *FIR* for the area serviced by the segment, the flow along the canal varies with the distance x according to the differential equation

$$\frac{dQ}{dx} = -kQ - \frac{FIR}{L} \quad (7.1)$$

where Q is the instantaneous flow at section x and L is the total length of the canal segment.

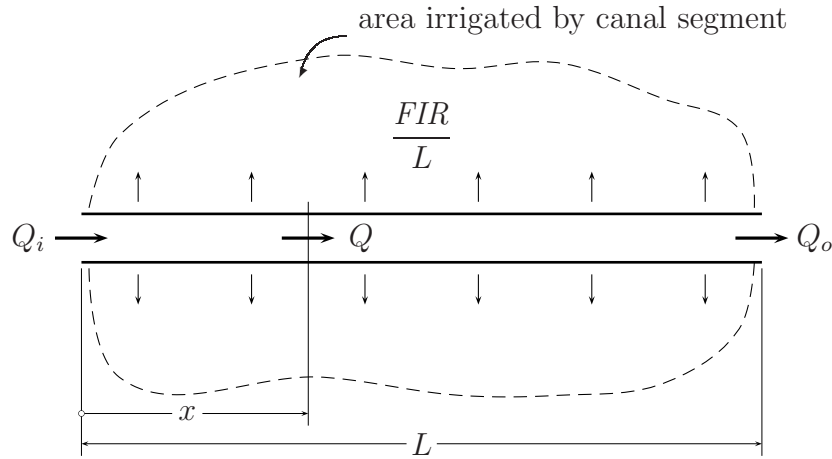


Figure 7.2 Flow across a typical canal segment

Integrating Eq.(7.1) yields Q_i , which is the *IDR* at the upstream end of the canal segment.

$$Q_i = \frac{e^{kL} - 1}{kL} FIR + e^{kL} Q_o \quad (7.2)$$

The first term on the right represents the inflow at the upstream end of the canal segment needed to satisfy the *FIR* for the area served by the segment, plus the losses thereby incurred. The second term represents the inflow at the upstream end of the segment needed to satisfy the outflow at the downstream end, plus the losses thereby incurred.

To illustrate how Eq.(7.2) is applied consider the irrigation network shown below. Assume that we are given the length of each canal segment, i.e., L_A, L_B, \dots, L_G , and the farm irrigation requirement $FIR_A, FIR_B, \dots, FIR_G$ for the areas served by the segments. Assume further that the parameter k is constant for the entire area. We want to find the amount of water to be released at the diversion dam; in terms of Eq.(7.2) this is the inflow at the upstream end of segment A, or Q_i^A .

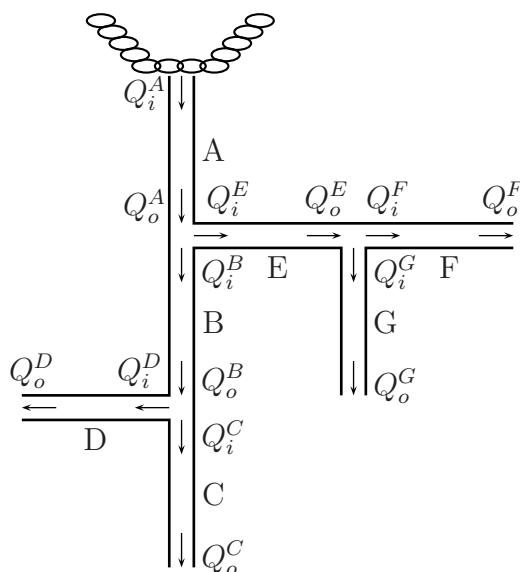


Figure 7.3 Calculating the *IDR*'s in an irrigation network

The following is a summary of the computations we need to perform to find Q_i^A .

$Q_i^A = \frac{e^{kL_A} - 1}{kL_A} FIR_A + e^{kL_A} Q_o^A$	where $Q_o^A = Q_i^B + Q_i^E$
$Q_i^B = \frac{e^{kL_B} - 1}{kL_B} FIR_B + e^{kL_B} Q_o^B$	where $Q_o^B = Q_i^C + Q_i^D$
$Q_i^C = \frac{e^{kL_C} - 1}{kL_C} FIR_C + e^{kL_C} Q_o^C$	where we may assume that $Q_o^C = 0$
$Q_i^D = \frac{e^{kL_D} - 1}{kL_D} FIR_D + e^{kL_D} Q_o^D$	where we may assume that $Q_o^D = 0$
$Q_i^E = \frac{e^{kL_E} - 1}{kL_E} FIR_E + e^{kL_E} Q_o^E$	where $Q_o^E = Q_i^F + Q_i^G$
$Q_i^F = \frac{e^{kL_F} - 1}{kL_F} FIR_F + e^{kL_F} Q_o^F$	where we may assume that $Q_o^F = 0$
$Q_i^G = \frac{e^{kL_G} - 1}{kL_G} FIR_G + e^{kL_G} Q_o^G$	where we may assume that $Q_o^G = 0$

Operationally we compute for the *IDR*'s in the following sequence: Q_i^C , Q_i^D , Q_i^B , Q_i^F , Q_i^G , Q_i^E , Q_i^A . That is, we start at the downstream reaches of the network and backtrack towards the source (*paatras*). If the network is small, the computations can be readily carried out by hand. However, if the network is large or if there are several such networks (there were actually 16 covered by the UPRP), it is worthwhile to computerize the entire process. This means that we have to expand our mathematical model to address the following questions:

7.1 Calculating conveyance losses in an irrigation network 187

1. How do we represent an irrigation network in the memory of a computer?
2. Having chosen a particular representation, how do we carry out the computations according to Eq.(7.2)?

It turns out that the topology of an irrigation network is suitably modeled by a binary tree and the ‘retrograde’ nature of the computations implied by Eq.(7.2) readily carried out by traversing the binary tree in postorder.

The model

Irrigation network and binary tree: there exists between these two a close affinity. The proposed model is based on the following conventions which define a one-to-one correspondence between the elements of the former and those of the latter.

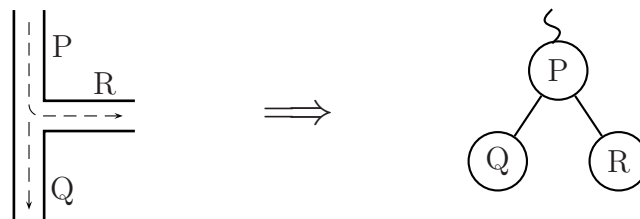
1. Each canal segment in the irrigation network is a node of the binary tree. A node has the structure

<i>LSON</i>	<i>LABEL</i>	<i>LENGTH</i>	<i>FIR</i>	<i>IDR</i>	<i>RSN</i>

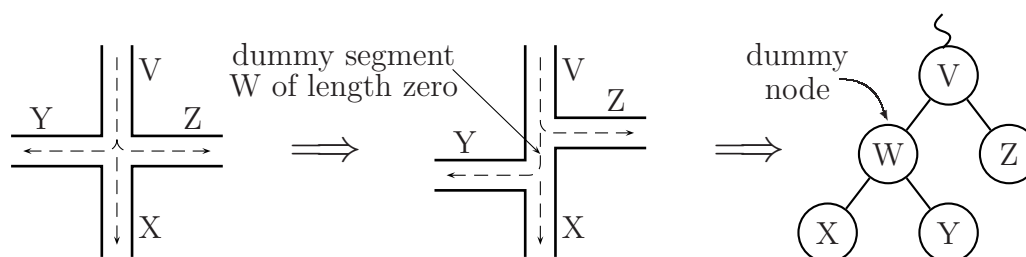
where

- LSON* = pointer to left subtree of the node
- LABEL* = label of the canal segment
- LENGTH* = length of the canal segment
- FIR* = farm irrigation requirement for the area serviced by the canal segment
- IDR* = irrigation diversion requirement at the upstream end of the canal segment
- RSN* = pointer to right subtree of the node

2. The canal segment which continues in the same direction will be the left son, and the segment which branches out in the perpendicular direction will be the right son. This is illustrated below.



3. Where two segments branch out in the perpendicular direction, a dummy node with a zero entry in the *LENGTH* field will be included, as indicated below



4. Where $n > 2$ segments branch out in several directions, apply Rule 3 $n-1$ times.

Figure 7.4 Rules for transforming an irrigation network into a binary tree

Figure 7.5 shown below depicts the binary tree representation of the irrigation network shown in Figure 7.1 according to the rules listed in Figure 7.4 The internal linked

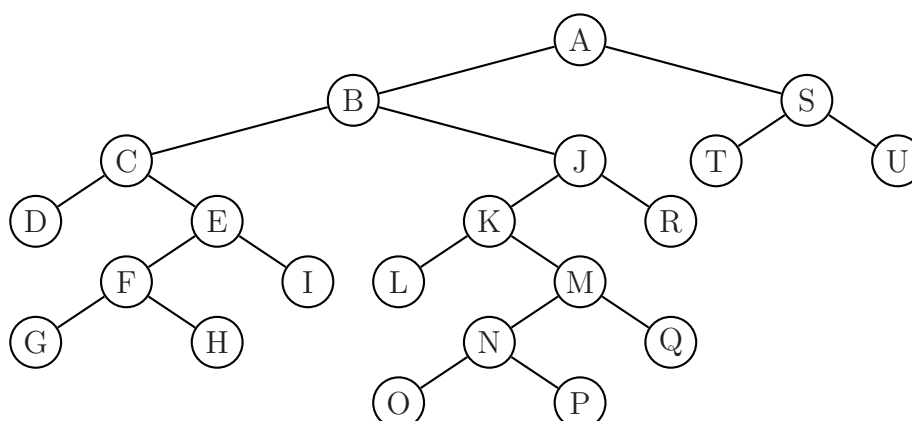


Figure 7.5 Binary tree representation of the irrigation network in Figure 7.1

representation of the binary tree is readily generated by inputting the data pertaining to each canal segment, and growing the binary tree, in preorder. For the irrigation network in Figure 7.1 the input sequence is A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T and U; i.e., the data for segment A is read in and the corresponding node constructed, followed by the data for segment B, and so on. A dummy segment N of length zero is added at the intersection of M, O, P and Q in accordance with Rule 3.

With the irrigation network represented internally as a binary tree, the computations for the irrigation diversion requirements according to Eq.(7.2) are readily performed by traversing the binary tree in postorder, i.e., according to the sequence D, G, H, F, I, E, C, L, O, P, N, Q, M, K, R, J, B, T, U, S and A.

Implementation of the model

The solution to the problem of calculating conveyance losses in an irrigation network using a binary tree as the computational model is implemented by five EASY procedures — CONSTRUCT, GETDATA, CALCULATE, COMPUTE_IDR and PRINT_IDR.

7.1 Calculating conveyance losses in an irrigation network 189

Procedure CONSTRUCT builds the linked binary tree representation of an irrigation network according to the conventions shown in Figure 7.4. The template for CONSTRUCT is *preorder* traversal; the basis is THEOREM 6.1 of Session 6 on the equivalence of binary trees. Consider, for instance, the network shown in Figure 7.3 and reproduced below for quick reference. The following data, enumerated in preorder, allow us to construct the binary tree representation of the network uniquely.

<i>label</i>	<i>length</i>	<i>fir</i>	<i>ltag</i>	<i>rtag</i>
A	xxx	xxx	1	1
B	xxx	xxx	1	1
C	xxx	xxx	0	0
D	xxx	xxx	0	0
E	xxx	xxx	1	1
F	xxx	xxx	0	0
G	xxx	xxx	0	0

In terms of the irrigation network, $ltag_A = 1$ means there is a segment, namely B, which continues in the same direction as segment A, and $rtag_A = 1$ means there is a segment, namely E, which branches out in the perpendicular direction. In terms of the binary tree, $ltag_A = 1$ means node A has a left son, namely node B; and $rtag_A = 1$ means node A has a right son, namely node E. Similarly, $ltag_C = rtag_C = 0$ means segment C is terminal (*dulo na*), or equivalently, node C is a leaf. Given the *ltag* and *rtag* sequences in preorder, we can construct the binary tree.

CONSTRUCT invokes procedure GETDATA which performs the operations local to a node. Essentially this involves reading in the data (*label, length, fir, ltag, rtag*) pertaining to the current canal segment and storing these in the fields of the current node. At this point, the *LSON* and *RSON* fields contain, not addresses, but *tags* to indicate whether the node has a left subtree and/or right subtree awaiting construction. Since in preorder, processing the right subtree of a node is postponed until the left subtree has been fully processed, a stack is used to store the addresses of nodes with right subtrees to be subsequently constructed.

Procedures CONSTRUCT and GETDATA invoke the stack routines of Session 4; either the array implementation or the linked list implementation of the stack \mathbb{S} may be utilized.

Procedure CALCULATE carries out the computations for the *IDR*'s using Eq.(7.2). The template for CALCULATE is *postorder* traversal, which is our way of implementing the retrograde manner in which the computations for the *IDR*'s are performed. Procedure CALCULATE is essentially procedure POSTORDER (Procedure 6.2) of Session 6; the generic procedure VISIT in POSTORDER has simply been made more specific as procedure COMPUTE_IDR in CALCULATE.

190 SESSION 7. Applications of binary trees

Upon return from procedure CALCULATE, the contents of the data fields *LABEL*, *LENGTH*, *FIR* and *IDR* can be printed node by node (i.e., for each canal segment in the network) by traversing the binary tree in preorder, as in procedure PRINT_IDR.

Now the procedures. Study them carefully.

```

1  procedure CONSTRUCT(T)
▷ Creates the linked binary tree representation of an irrigation network. Pointer to the
▷ created binary tree is T
2  node(LSON, LABEL, LENGTH, FIR, IDR, RSON)
3  call InitStack(S)
▷ Set up root node
4  call GETDATA(S, T)
5   $\alpha \leftarrow T$ 
▷ Create rest of binary tree in preorder
6  loop
7      while LSON( $\alpha$ )  $\neq 0$  do
8          call GETDATA(S,  $\beta$ )
9          LSON( $\alpha$ )  $\leftarrow \beta$ ;  $\alpha \leftarrow \beta$ 
10     endwhile
11     if IsEmptyStack(S) then return
12         else [ call POP(S,  $\alpha$ )
13                 call GETDATA(S,  $\beta$ )
14                 RSON( $\alpha$ )  $\leftarrow \beta$ ;  $\alpha \leftarrow \beta$  ]
15 forever
16 end CONSTRUCT

```

```

1  procedure GET_DATA(S,  $\beta$ )
2  node (LSON, LABEL, LENGTH, FIR, IDR, RSON)
3  call GETNODE( $\beta$ )
4  input label, length, fir, ltag, rtag
5  LABEL( $\beta$ )  $\leftarrow$  label
6  LENGTH( $\beta$ )  $\leftarrow$  length
7  FIR( $\beta$ )  $\leftarrow$  fir
8  IDR( $\beta$ )  $\leftarrow 0$ 
9  LSON( $\beta$ )  $\leftarrow$  ltag
10 RSON( $\beta$ )  $\leftarrow$  rtag
11 if RSON( $\beta$ ) = 1 then call PUSH(S,  $\beta$ )
12 end GETDATA

```

Procedure 7.1 Constructing a binary tree to represent an irrigation network

7.1 Calculating conveyance losses in an irrigation network 191

```

1  procedure CALCULATE(T)
2  node(LSON, LABEL, LENGTH, FIR, IDR, RSON)
3  if T ≠ Λ then [ call CALCULATE(LSON(T))
4                  call CALCULATE(RSON(T))
5                  call COMPUTE_IDR(T) ]
6  end CALCULATE

1  procedure COMPUTE_IDR( $\alpha$ )
2  node (LSON, LABEL, LENGTH, FIR, IDR, RSON)
3  if LSON( $\alpha$ ) ≠ Λ then IDR( $\alpha$ ) ← IDR( $\alpha$ ) + IDR(LSON( $\alpha$ ))
4  if RSON( $\alpha$ ) ≠ Λ then IDR( $\alpha$ ) ← IDR( $\alpha$ ) + IDR(RSON( $\alpha$ ))
5  if LENGTH( $\alpha$ ) ≠ 0 then [ kL ← 0.015 * LENGTH( $\alpha$ )
6                          IDR( $\alpha$ ) ← ( $e^{kL} - 1$ ) * FIR( $\alpha$ )/kL +  $e^{kL}$  * IDR( $\alpha$ ) ]
7  end COMPUTE_IDR

```

Procedure 7.2 Computing the *IDR*'s in an irrigation network

```

1  procedure PRINT_IDR(T)
2  node(LSON, LABEL, LENGTH, FIR, IDR, RSON)
3  if T ≠ Λ then [ output LABEL(T), LENGTH(T), FIR(T), IDR(T)
4                  call PRINT_IDR(LSON(T))
5                  call PRINT_IDR(RSON(T)) ]
6  end PRINT_IDR

```

Procedure 7.3 Printing the computed *IDR*'s

Sample results

The table below shows the computed *IDR*'s for the hypothetical network shown in Figure 7.3 using the hypothetical data in the *Length* and *FIR* columns. The parameter *k* is taken to be 1.5% of the flow per kilometer of canal over riceland.

<i>Label</i>	<i>Length</i> , km	<i>FIR</i> , cms	<i>IDR</i> , cms
A	1.54	0.075	0.888
B	1.83	0.133	0.436
C	2.60	0.164	0.167
D	3.75	0.122	0.126
E	1.98	0.108	0.358
F	2.45	0.146	0.149
G	2.24	0.091	0.092

7.2 Heaps and the heapsort algorithm

In 1964 R. W. Floyd and J. W. J. Williams put forth an elegant sorting algorithm called *heapsort*. The algorithm is built around a nifty data structure called a *heap* and an equally nifty process called *sift-up*.

What is a heap?

Structurally a heap is a *complete* binary tree. In Session 6 we saw that a complete binary tree is a binary tree with leaves on at most two adjacent levels, l and $l-1$, in which leaves at level l occupy the leftmost positions. An important characteristic of a complete binary tree is that if leaves are deleted in reverse level order the resulting binary trees are also complete.

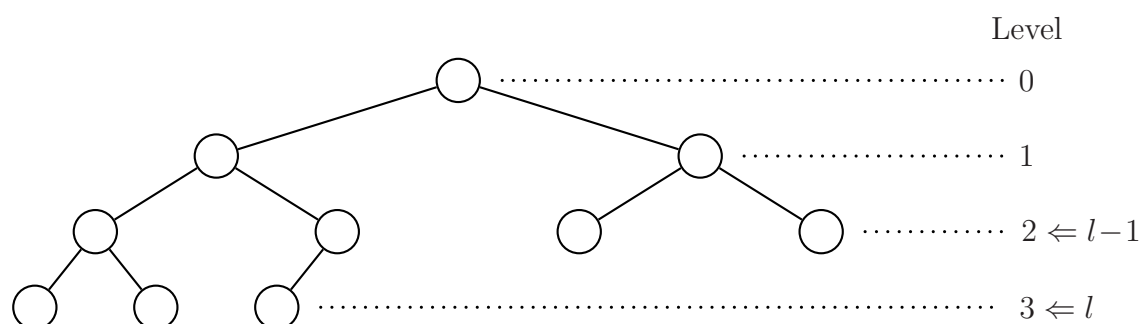


Figure 7.6 A complete binary tree

Now let $\{K_1, K_2, K_3, \dots, K_n\}$ be a set of *keys* on which *total order* has been defined. Recall from Session 2 that a relation R on a set is a total order if every pair of elements in the set is comparable; in this case, given any two keys K_i and K_j we have either $K_i \leq K_j$ or $K_j \leq K_i$, numerically or lexicographically.

DEFINITION 7.1. A **max-heap** (**min-heap**) is a complete binary tree with keys assigned to its nodes such that the key in each node is greater (less) than or equal to the keys in its left and right son nodes.

The second half of the above definition is called the *heap-order property* and is depicted in the figure below.

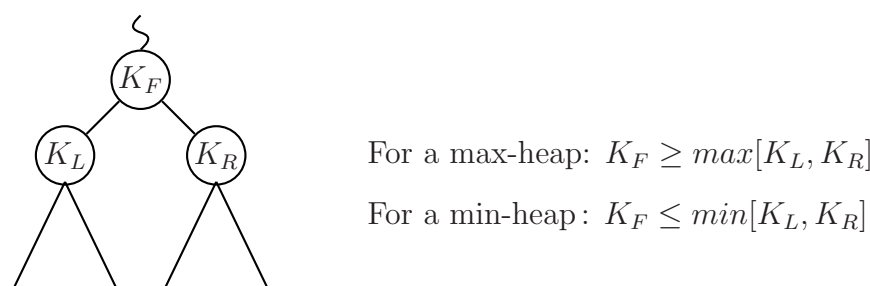


Figure 7.7 The heap-order property

For example, using IBM/370 Assembly language instruction codes as alphabetic keys, Figure 7.8 depicts a non-heap and a heap.

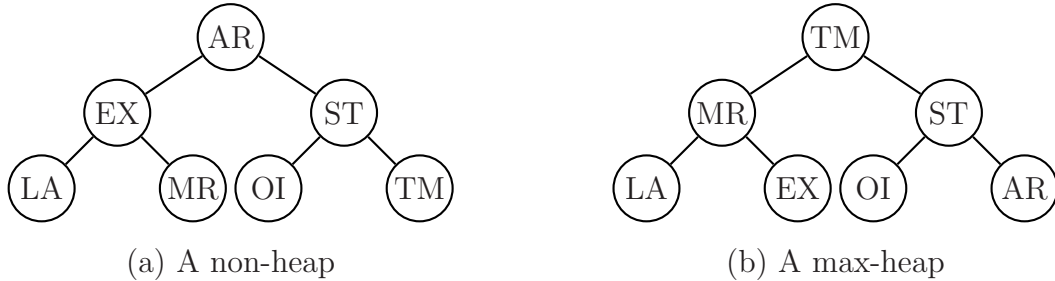


Figure 7.8 A non-heap and a heap

The following statements are readily verifiable facts about a heap.

1. Every subtree in a heap is a heap.
2. In a max-heap, the root contains the largest key; in a min-heap, the root contains the smallest key.
3. Given a set of keys, there is more than one assignment of the keys to the nodes of a complete binary tree to form a heap. For instance, if the subtrees of any node in the heap of Figure 7.8(b) are interchanged, the result is still a heap.
4. The height of a heap on n nodes is $\lfloor \log_2 n \rfloor$.

7.2.1 Sift-up: converting a complete binary tree into a heap

Given a non-heap, there are various ways by which we can form a heap. (In the rest of this section, we will use the term heap to mean a max-heap.) Shown below are three different sequences of key interchanges that will convert the non-heap of Figure 7.8(a) into a heap, as you may easily verify. The first sequence is the result of an unsystematic, essentially ad hoc process that produces a heap. The third sequence is the result of a well-defined procedure called *sift-up*, requiring only four key exchanges instead of nine.

AR \Leftrightarrow EX	AR \Leftrightarrow ST	ST \Leftrightarrow TM
AR \Leftrightarrow LA	AR \Leftrightarrow TM	EX \Leftrightarrow MR
EX \Leftrightarrow LA	ST \Leftrightarrow TM	AR \Leftrightarrow TM
EX \Leftrightarrow MR	EX \Leftrightarrow LA	AR \Leftrightarrow ST
LA \Leftrightarrow MR	LA \Leftrightarrow MR	
MR \Leftrightarrow ST		
MR \Leftrightarrow OI		
OI \Leftrightarrow TM		
ST \Leftrightarrow TM		

DEFINITION 7.2. **Sift-up** is a bottom-up, right-to-left process in which the smallest subtrees of a complete binary tree are converted into heaps, then the subtrees which contain them, and so on, until the entire binary tree is converted into a heap.

Example 7.1. Heapifying the non-heap of Figure 7.8(a) using sift-up

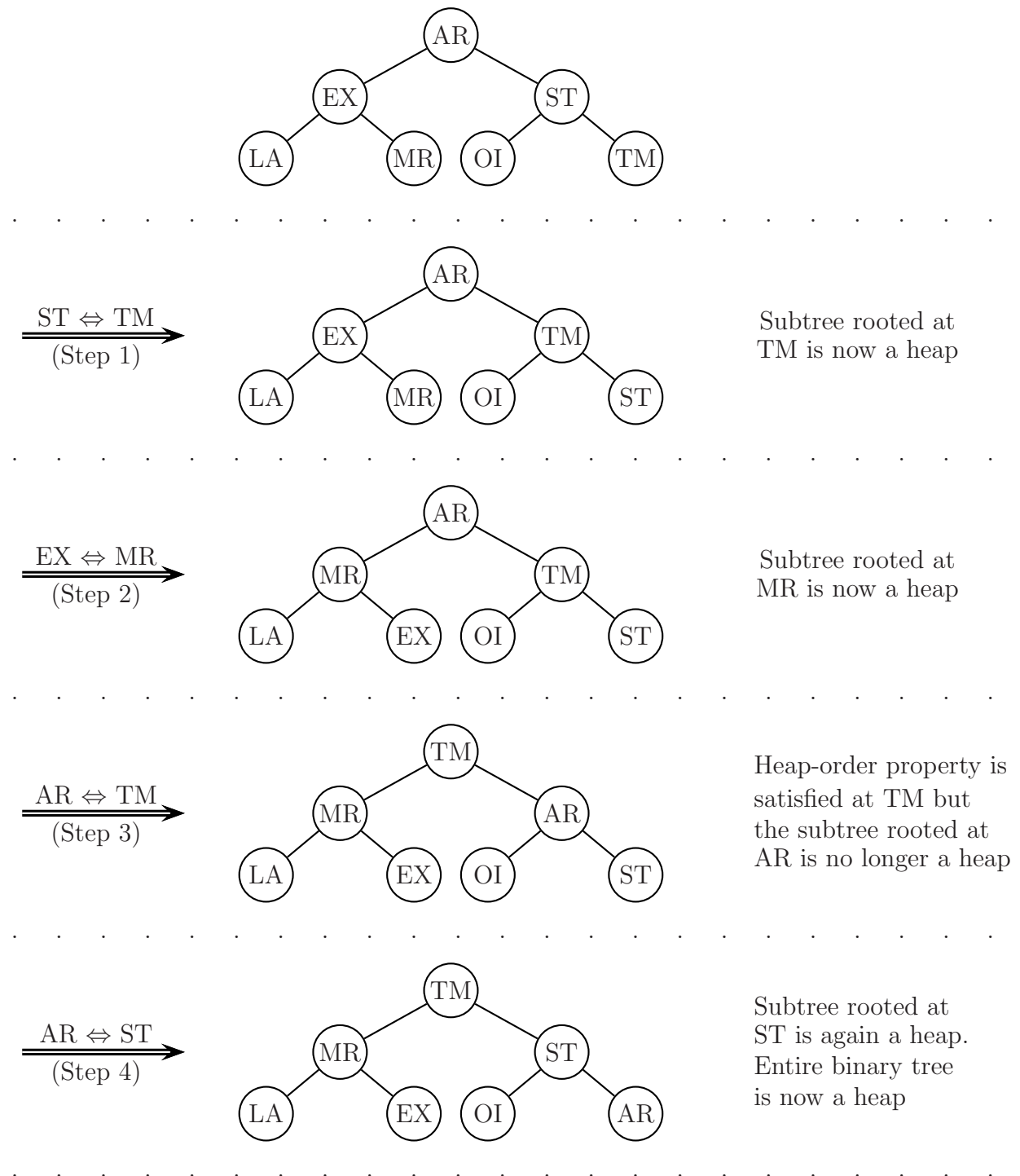


Figure 7.9 Converting a complete binary tree into a heap

With Figure 7.9 as an illustrative example, we take note of the following facts regarding sift-up as a process to convert a complete binary tree on n nodes into a heap.

1. The terminal nodes of the binary tree are already heaps. In the above example, these are the trees rooted at TM, OI, MR and LA.

2. A total of $\lfloor n/2 \rfloor$ trees are successively converted into heaps in *reverse level order* before the entire binary tree becomes a heap. In the example given above, these are the $\lfloor 7/2 \rfloor = 3$ trees rooted at ST, EX and AR.
3. When the tree rooted at any node, say node ρ , is converted into a heap, its left and right subtrees are already heaps. We call such a tree an *almost-heap*. When an almost-heap is converted into a heap, one of its subtrees may cease to be a heap, i.e., it may become an almost-heap. This subtree is once more converted into a heap, but this may again cause one of its subtrees to lose the heap-order property. And so on. The process continues as smaller and yet smaller subtrees lose and regain the heap-order property, with larger keys migrating upwards, until the smaller key from the starting root node ρ finds its final resting place.

In the example shown above, when the key AR is exchanged with the key TM in Step 3, the resulting right subtree rooted at AR ceases to be a heap, and is converted back into a heap in Step 4, at which point AR finds final placement.

Procedure HEAPIFY formalizes the process described in Item 3 for an almost-heap in a binary tree represented using linked allocation, as in Figure 6.10 of Session 6, but with node structure (LSON,KEY,RSON).

```

1  procedure HEAPIFY( $\rho$ )
2     $k \leftarrow KEY(\rho)$ 
3     $\alpha \leftarrow \rho$ 
4     $\beta \leftarrow LSON(\alpha)$ 
5    while  $\beta \neq \Lambda$  do
6       $\sigma \leftarrow RSON(\alpha)$ 
7      if  $\sigma \neq \Lambda$  and  $KEY(\sigma) > KEY(\beta)$  then  $\beta \leftarrow \sigma$ 
8      if  $KEY(\beta) > k$  then [  $KEY(\alpha) \leftarrow KEY(\beta)$ 
9                           $\alpha \leftarrow \beta$ 
10                          $\beta \leftarrow LSON(\alpha)$  ]
11                      else exit
12  endwhile
13   $KEY(\alpha) \leftarrow k$ 
14  end HEAPIFY

```

Procedure 7.4 Converting an almost-heap into a heap (linked representation)

Procedure HEAPIFY accepts as input a pointer, ρ , to the root node of the almost-heap to be converted into a heap (line 1). Two roving pointers are used: α , which points to the current root node (lines 3 and 9); and β , which points to whichever of the two sons of the current root node contains the larger key (line 7). The key k in the starting root node ρ (line 2) is compared but not actually exchanged with keys along a downward path, with keys larger than k migrating upwards (line 8) until a position is found for k such that the heap-order property is satisfied (line 11); at this point, k finds final placement (line 13).

To convert a complete binary tree on n nodes into a heap, HEAPIFY is called $\lfloor n/2 \rfloor$ times starting with the rightmost almost-heap at the bottommost level and continuing in

reverse level order. With the linked representation of a binary tree in which the pointer to the tree points to its root, locating the roots of the almost-heaps which are to be converted into heaps results in unnecessary additional overhead.

Consider, for instance, the binary tree shown in Figure 7.10 and imagine this to be represented in memory using linked allocation with an external pointer to the root node MR. To convert the binary tree into a heap we call on HEAPIFY six times with the argument ρ pointing successively to nodes AR, TM, OI, ST, LA and finally MR. Except for node MR, we gain access to the other nodes only by traversing the binary tree, say with the use of a deque, first in level order and subsequently in reverse level order. This overhead in terms of time and space can be eliminated altogether if we use the *sequential* representation of a complete binary tree in a one-dimensional array.

Sequential representation of a complete binary tree

In the sequential representation of a complete binary tree on n nodes, the nodes of the binary tree (actually, the contents of the data field of the nodes) are stored in a one-dimensional array in *level order*. If we think of the nodes of the binary tree as being numbered from 1 through n in level order then these numbers serve as the indices to these nodes in the array in which they are stored. This is depicted in Figure 7.10 where the array is called *KEY*. Note that no links are maintained; the father-son relationship between nodes is implicit in the position of the nodes in the array.

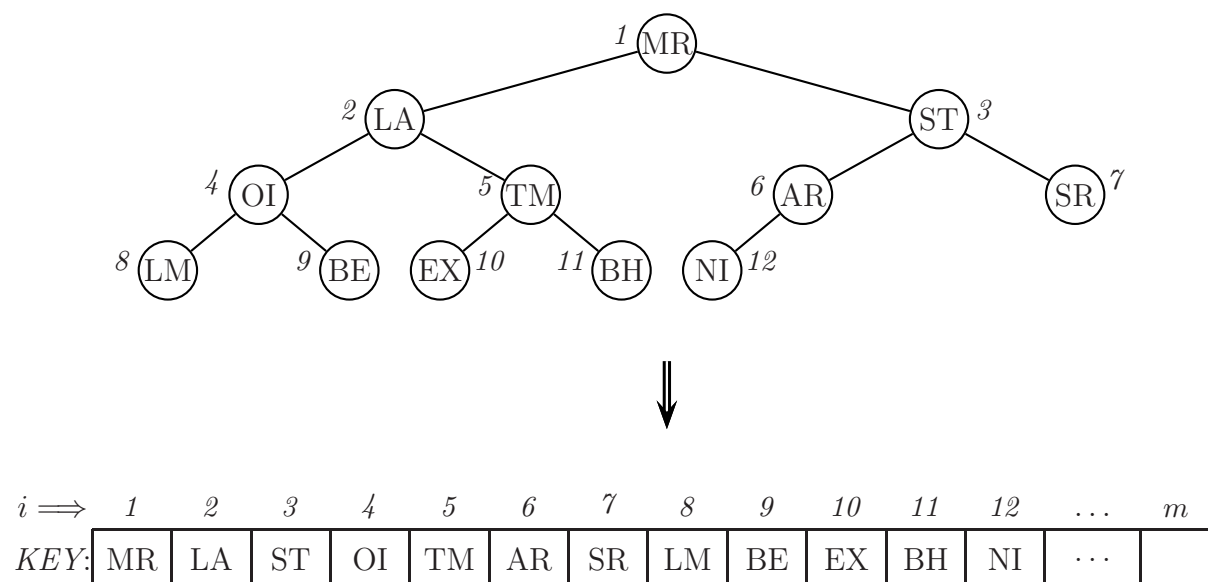


Figure 7.10 Sequential representation of a complete binary tree

The following formulas allow us to locate, in constant time, the sons (if any) and the father (if it exists), of *any* node, say node i , in a sequentially allocated complete binary tree on n nodes. These formulas are in fact the reason why we require that a heap be a complete binary tree; they are not applicable, in general, to a binary tree that is not complete.

- (a) The left son of node i is node $2*i$, provided that $2*i \leq n$;
otherwise, node i has no left son.
- (b) The right son of node i is node $2*i+1$, provided that $2*i+1 \leq n$;
otherwise, node i has no right son.
- (c) The father of node i is node $\lfloor i/2 \rfloor$, provided that $1 < i \leq n$.

(7.3)

To illustrate the use of these formulas, consider for instance, node TM which is in cell 5; its sons EX and BH are found in cells $2*5 = 10$ and $2*5 + 1 = 11$ as predicted. Now consider BE which is in cell 9; its father OI is found in cell $\lfloor 9/2 \rfloor = 4$ as predicted.

We will denote the data structure shown in Figure 7.10 as $\mathbb{B} = [KEY(1:m), size\mathbb{B}]$, where $size\mathbb{B}$ is the size (number of nodes) of the binary tree and KEY is the array of size m where the keys in the nodes of the binary tree are stored in level order. Shown below is the version of HEAPIFY for this sequential implementation of a complete binary tree.

```

1  procedure HEAPIFY( $\mathbb{B}, r$ )
2     $k \leftarrow KEY(r)$ 
3     $i \leftarrow r$ 
4     $j \leftarrow 2*i$ 
5    while  $j \leq size\mathbb{B}$  do
6      if  $j < size\mathbb{B}$  and  $KEY(j+1) > KEY(j)$  then  $j \leftarrow j+1$ 
7      if  $KEY(j) > k$  then [  $KEY(i) \leftarrow KEY(j)$ 
8                            $i \leftarrow j$ 
9                            $j \leftarrow 2*i$  ]
10     else exit
11  endwhile
12   $KEY(i) \leftarrow k$ 
13  end HEAPIFY

```

Procedure 7.5 Converting an almost-heap into a heap (array representation)

Procedure HEAPIFY(\mathbb{B}, r) accepts as input a pointer to the structure \mathbb{B} , thus allowing access to its components, and the index r in KEY of the root of the almost-heap in \mathbb{B} that is to be converted into a heap. This version of HEAPIFY is analogous, line by line, to the previous version HEAPIFY(ρ) for the linked case, with the indices i and j replacing the pointers α and β , respectively. Study the procedure carefully.

The following segment of EASY code converts a complete binary tree on n nodes, stored sequentially in the array $KEY(1:m)$, into a heap. Note how easy it is to locate the roots of the subtrees of the binary tree in reverse level order, something not as easily accomplished with the linked representation of a binary tree.

```

for  $r \leftarrow \lfloor n/2 \rfloor$  to 1 by -1 do
  call HEAPIFY( $\mathbb{B}, r$ )
endfor

```

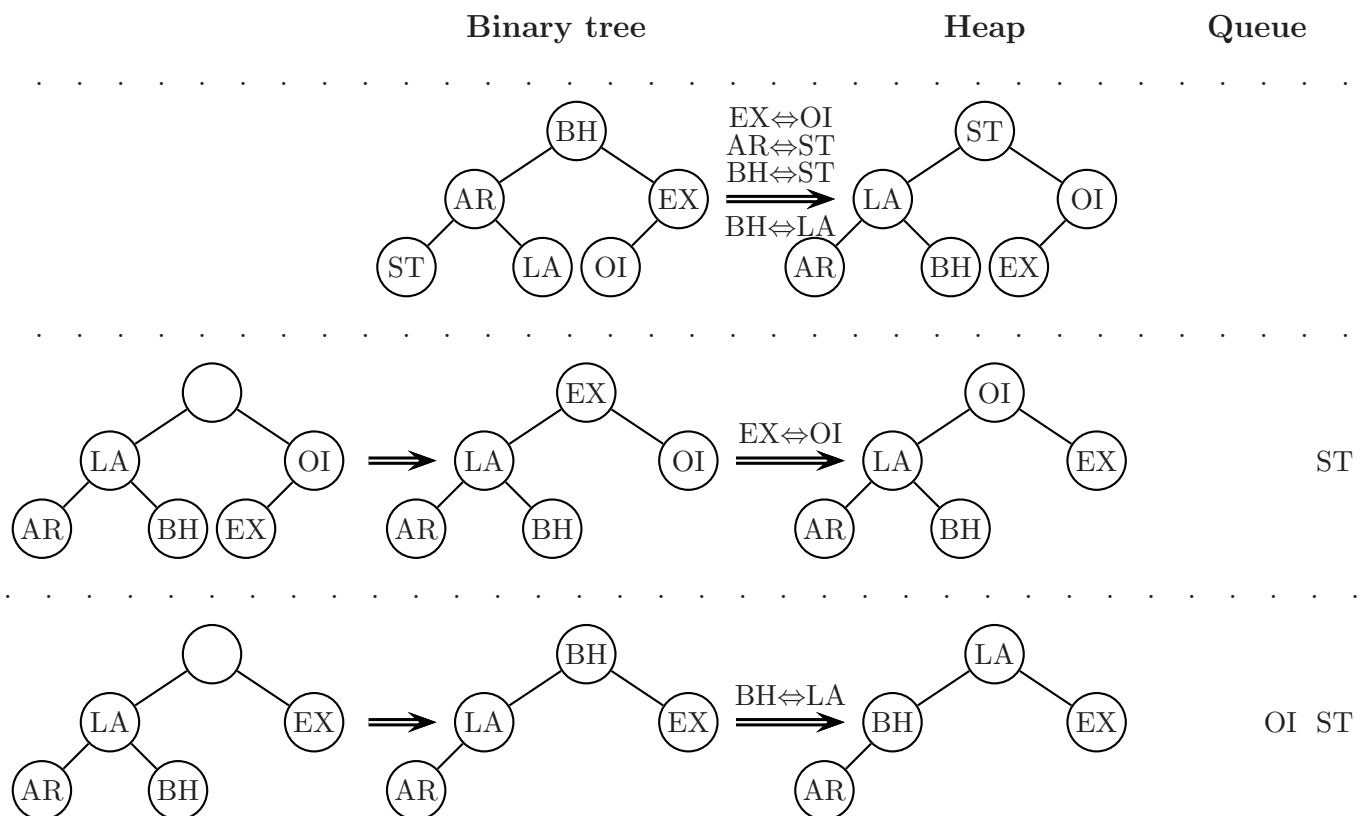
7.2.2 The heapsort algorithm of Floyd and Williams

With the heap as the implementing data structure we obtain the following elegant sorting algorithm (STANDISH[1980], p. 88):

Algorithm HEAPSORT (R. W. Floyd and J. W. J. Williams)

1. Assign the keys to be sorted to the nodes of a complete binary tree.
2. Convert this binary tree into a heap by applying sift-up to its nodes in reverse level order.
3. Repeatedly do the following until the heap is empty:
 - (a) Remove the key at the root of the heap (the largest in the heap) and place it in an output queue.
 - (b) Detach from the heap the rightmost leaf node at the bottommost level, extract its key, and store this key at the root of the heap.
 - (c) Apply sift-up to the root to convert the binary tree into a heap once again.

Example 7.2. The figure below depicts heapsort in action on the keys BH, AR, EX, ST, LA and OI. The figure may be imagined as divided into 7 rows and 4 columns. Shown in row 1 is the binary tree (in column 2) after Step 1 completes and the heap (in column 3) after Step 2 completes. Shown in rows 2 through 7 is the binary tree (in columns 1 and 2) at the end of Step 3(a) and 3(b), respectively, and the heap (in column 3) at the end of Step 3(c); in column 4 of each row is the output queue.



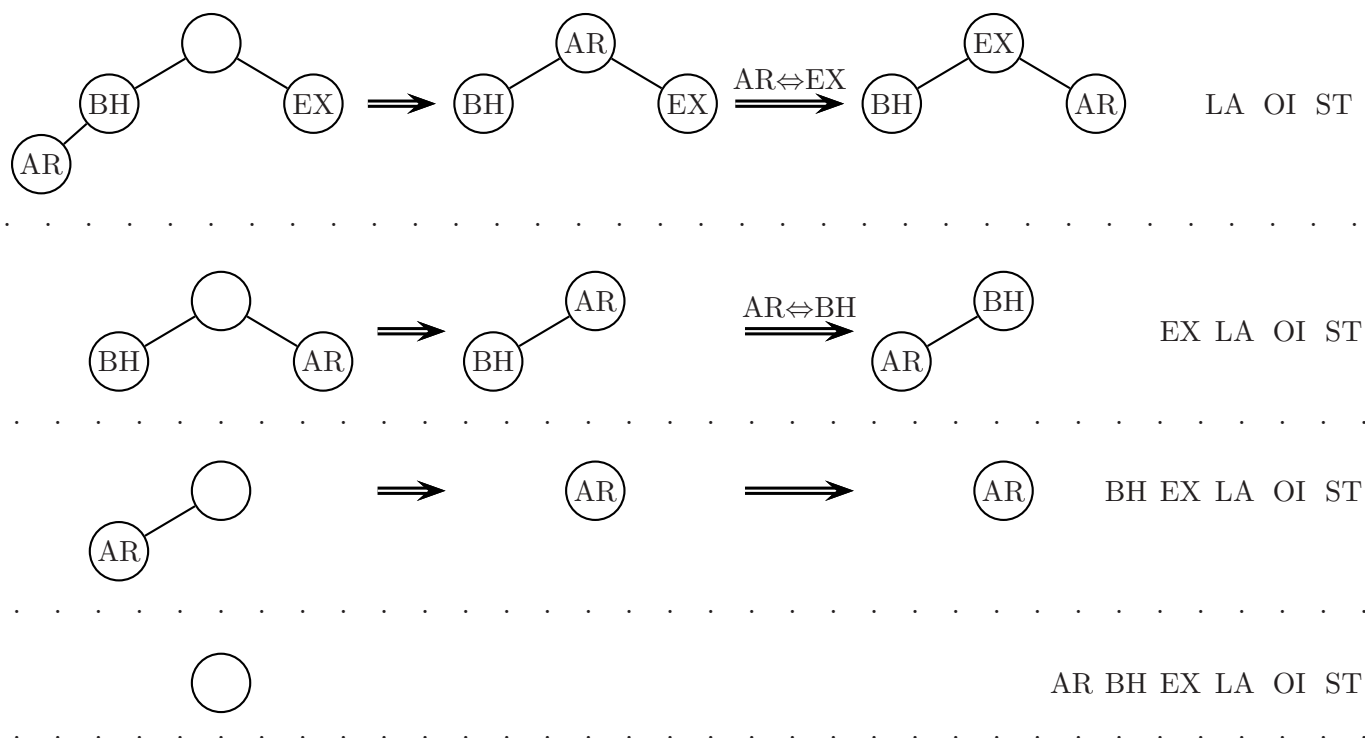


Figure 7.11 Heapsort in action

Procedure $\text{HEAPSORT}(\mathbb{K})$ accepts as input a set of n keys which we may imagine to be assigned to the nodes of a complete binary tree, with the binary tree stored in level order in an array KEY of size m . This input is represented by the data structure $\mathbb{K} = [\text{KEY}(1:m), \text{size}\mathbb{K}]$, where $\text{size}\mathbb{K} = n$. HEAPSORT performs an *in-place* sort of the keys with the heap and the output queue coexisting in KEY . The key at the root of the heap that is removed in Step 3(a) of Algorithm HEAPSORT occupies the cell that is vacated when a leaf is detached from the heap in Step 3(b). This particular implementation of heapsort, due to Floyd, is without question a most elegant one.

```

1  procedure HEAPSORT( $\mathbb{K}$ )
2     $n \leftarrow \text{size}\mathbb{K}$ 
3  ▷ Step 2: Convert binary tree into a heap
4  for  $r \leftarrow \lfloor n/2 \rfloor$  to 1 by -1 do
5    call HEAPIFY( $\mathbb{K}, r$ )
6  endfor
7  ▷ Steps 3(a),(b),(c): Heapsort proper
8  for  $j \leftarrow n$  to 2 by -1 do
9     $\text{KEY}(1) \Leftrightarrow \text{KEY}(j)$           ▷ swap keys in root and last leaf
10    $\text{size}\mathbb{K} \leftarrow j - 1$           ▷ Detach last leaf from binary tree
11   call HEAPIFY( $\mathbb{K}, 1$ )          ▷ Sift-up to root of new binary tree
12 endfor
13  $\text{size}\mathbb{K} \leftarrow n$               ▷ Restore value of  $\text{size}\mathbb{K}$ 
14 end HEAPSORT

```

Procedure 7.6 Heapsort

The figure below shows the resulting heap and output queue as stored in the array *KEY* at successive stages of the algorithm for the example given in Figure 7.11. Each of the 7 instances of the array *KEY* in Figure 7.12 is a snapshot of the heap and the queue as depicted in each of the 7 rows in Figure 7.11 as they coexist in *KEY*. Note that the boundary between heap and queue is simply established by the value of the indexing variable *j* in the second **for**-loop (line 6) in procedure HEAPSORT.

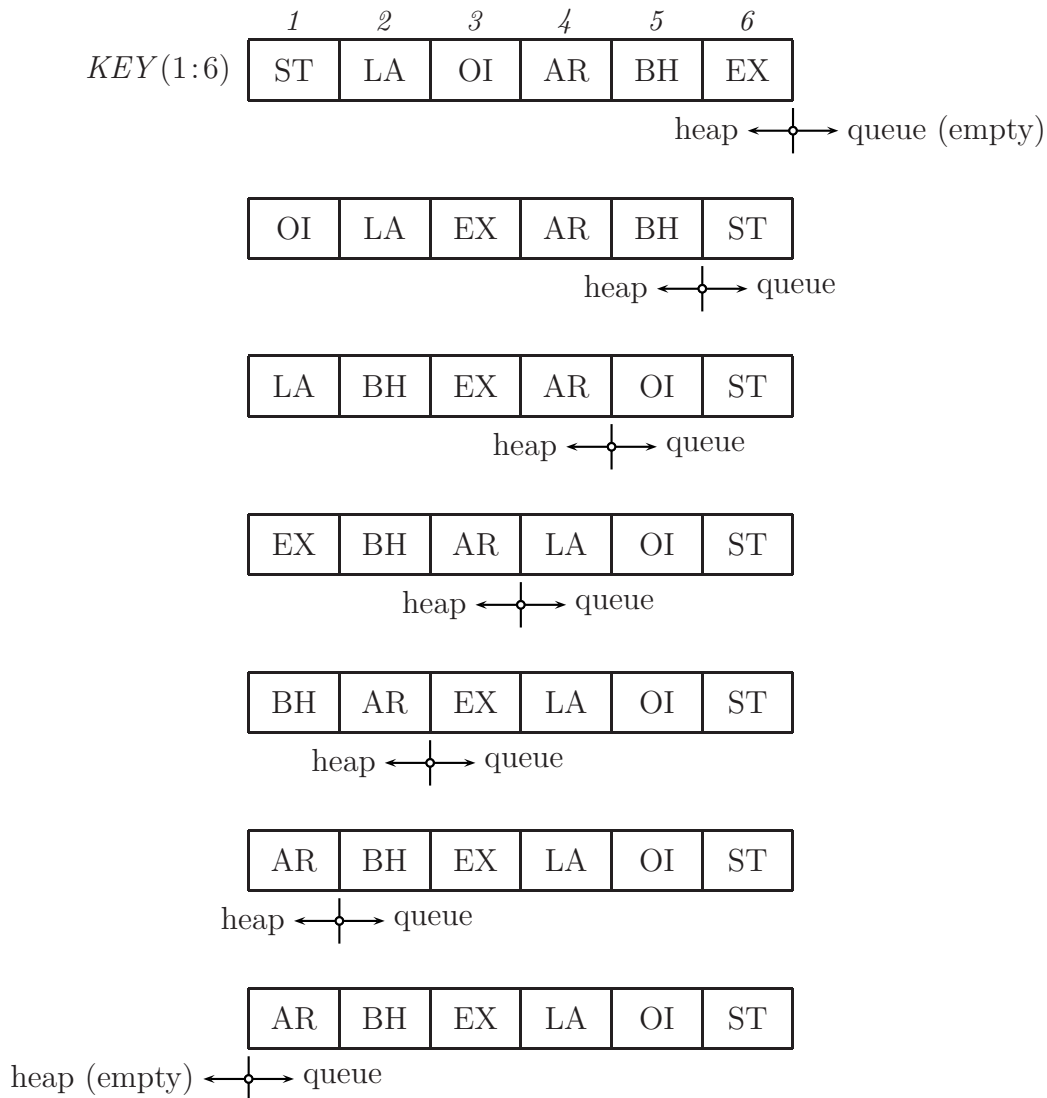


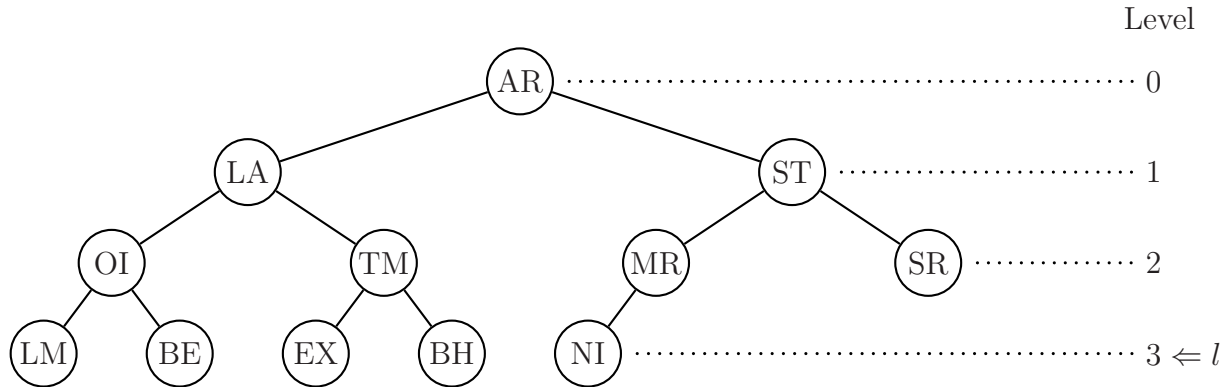
Figure 7.12 The heap and queue coexisting in computer memory during heapsort

To invoke procedure HEAPSORT, we simply write

call HEAPSORT(\mathbb{K})

where $\mathbb{K} = [KEY(1:m), size\mathbb{K}]$ and $size\mathbb{K} = n$. At entry into the procedure the array *KEY* contains the *n* keys to be sorted; upon return to the calling program *KEY* contains the sorted keys. Since only the array *KEY* and a few auxiliary variables are used, the space complexity of heapsort as implemented by Procedure 7.6 is clearly $O(n)$.

Time complexity of heapsort



With the figure shown above as a guide, we analyze each of the three steps of Algorithm HEAPSORT.

1. Assigning n keys to the nodes of a complete binary tree clearly takes $O(n)$ time.
2. Converting a complete binary tree on n nodes into a heap

A key, say k , in a node on level i may be exchanged with keys along a downward path at most $l - i$ times. This worst case occurs if k comes to rest at the bottommost level l . Since there are at most 2^i nodes on level i , each of which may be exchanged at most $l - i$ times, the total number of key exchanges performed in heapifying a complete binary tree of height l cannot exceed the quantity

$$\sum_{i=0}^l 2^i (l - i) < 2^{l+1} \quad (7.4)$$

(For a derivation of the upper bound 2^{l+1} , see STANDISH[1980], pp. 87–88.) Since $2^{l+1} \leq 2 * n$ (the upper limit applies when there is only one node on level l), it follows that converting a complete binary tree on n nodes into a heap takes $O(n)$ time.

3. Heapsort proper

Applying sift-up to the root may cause at most $\lfloor \log_2 n \rfloor$ key exchanges in the worst case in which the key at the root migrates to the bottommost level in a complete binary tree on n nodes. Since sift-up to the root is applied $n - 1$ times, each time the size of the binary tree decreasing by 1, the total number of key exchanges performed in sorting n keys cannot exceed the quantity

$$\sum_{i=2}^n \lfloor \log_2 i \rfloor = (n + 1) \lfloor \log_2(n + 1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2 \quad (7.5)$$

(See KNUTH1[1997], p. 400 for more details on this formula.) For sufficiently large n , the first term is dominant; hence, Step 3 takes $O(n \log_2 n)$ time.

The heapsort algorithm, therefore, executes in $O(n) + O(n) + O(n \log_2 n) = O(n \log_2 n) = O(n \log n)$ time.

7.3 Priority queues

In addition to the central role it plays in heapsort, one other important application of the heap is in the implementation of a particularly useful ADT called a *priority queue*.

DEFINITION 7.3. A priority queue is a set of elements where each element has attached to it a priority, specified by a key, which indicates which element in the set may be removed first.

The fundamental operations defined on a priority queue are:

1. *insert* an element into the priority queue
2. *extract* (i.e., delete and return) the element with highest priority

Additional operations which a certain application may require from a priority queue include:

3. *change* the priority of an element
4. *return* (i.e., return but don't delete) the element with highest priority
5. *delete* (i.e., delete and don't return) an arbitrary specified element

Then of course we have the usual auxiliary operations: initialize a priority queue, test if a priority queue is empty and test if a priority queue is full.

The priorities assigned to the elements of a priority queue normally depend on the requirements of a particular application. For instance, in a multi-programming environment a priority queue may be used to store jobs awaiting execution such that jobs which require less computing resources are assigned a higher priority over jobs which require more. In an event-driven simulation application a priority queue may be used to store events to be simulated such that events which ought to occur first are assigned a higher priority over events which are supposed to occur later (but of course). In Session 10, we will encounter applications involving graphs (recall from Session 1 that a graph is simply a set of vertices connected by edges) in which 'costs' are assigned to the edges; a priority queue is used to store the edges with the edge of least cost having highest priority.

A priority queue may be made to function as a stack if we consider 'time of insertion' as an element's priority with the element last inserted having the highest priority, and as an ordinary queue if the element first inserted has highest priority.

Implementing a priority queue

As with all the ADT's we have considered thus far, we can realize a priority queue using either the contiguous design or the linked design; we may use either an array or a linked list in which to store the queue elements. In this session we will consider three array-based implementations of a priority queue, namely, an unsorted array implementation, a sorted array implementation and a heap-ordered array implementation. In all three implementations we will use the notation $\mathbb{P} = [P(1:m), n]$ to denote the priority queue, where P is the array of size m in which resides the priority queue of size n .

The same boundary conditions apply to all three implementations, namely:

$n = 0$ means the priority queue is empty

$n = m$ means the priority queue is full

An attempt to remove an element from an empty priority queue results in an underflow condition; an attempt to insert an element into a full priority queue results in an overflow condition.

Consistent with the boundary condition $n = 0$ indicating an empty priority queue, we initialize the queue by setting

$$n \leftarrow 0.$$

Three versions each of procedure PQ_INSERT and procedure PQ_EXTRACT to perform the insert and extract operations, respectively, corresponding to the three array-based implementations of a priority queue, are given below. The following general observations and assumptions apply to all the procedures; additional commentaries on specific implementation issues are given at the appropriate places.

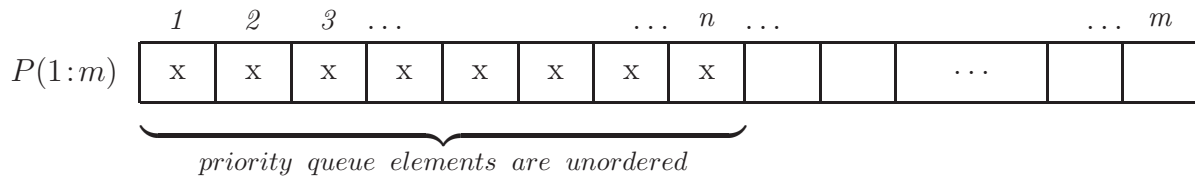
1. All six procedures have the same parameter list (\mathbb{P}, x) . A procedure accepts a pointer to the data structure \mathbb{P} , allowing it access to the components of \mathbb{P} . Any change made to these components is seen by the calling procedure. The parameter x contains the element to be inserted (if PQ_INSERT) or the deleted element (if PQ_EXTRACT).
2. As previously stated, associated with each element in a priority queue is a key which indicates its priority relative to the other elements. Our notation does *not* make a distinction between an element and its associated key, but it should be obvious from context if an operation involves only the key. For instance, in the statement

if $P(j) > x$ **then** ...

it is clear that the comparison is between the priorities of the elements $P(j)$ and x .

3. An overflow condition is handled by invoking PQ_OVERFLOW. We allow for the possibility that PQ_OVERFLOW will take corrective action by extending the array; in such a case a return is made to PQ_INSERT and the insert operation proceeds. Otherwise, PQ_OVERFLOW returns control to the runtime system.
4. An underflow condition is handled by invoking PQ_UNDERFLOW, which we leave unspecified.
5. We assume that the element with the largest key has highest priority. The procedures are easily modified to suit an application in which the element with smallest key has highest priority.

7.3.1 Unsorted array implementation of a priority queue

**Figure 7.13** Unsorted array implementation of $\mathbb{P} = [P(1:m), n]$

```

1  procedure PQ_INSERT( $\mathbb{P}, x$ )
2  if  $n = m$  then call PQ_OVERFLOW
3   $n \leftarrow n + 1$ 
4   $P(n) \leftarrow x$ 
5  end PQ_INSERT

```

Procedure 7.7 Insert operation on a priority queue (unsorted array implementation)

The new element is simply stored in the next available cell in P . Procedure 7.7 clearly takes $O(1)$ time.

```

1  procedure PQ_EXTRACT( $\mathbb{P}, x$ )
2  if  $n = 0$  then call PQ_UNDERFLOW
3  else [  $x \leftarrow P(1)$ ;  $i \leftarrow 1$ 
4  for  $j \leftarrow 2$  to  $n$  do
5  if  $P(j) > x$  then [  $x \leftarrow P(j)$ ;  $i \leftarrow j$  ]
6  endfor
7   $P(i) \leftarrow P(n)$ 
8   $n \leftarrow n - 1$  ]
9  end PQ_EXTRACT

```

Procedure 7.8 Extract operation on a priority queue (unsorted array implementation)

The element with highest priority is located and returned in x (lines 3–6). Subsequently, the last element is transferred to the vacated cell (line 7) and the size of the priority queue is updated (line 8). All n elements must be checked since each one is potentially the element with highest priority. Procedure 7.8 clearly takes $O(n)$ time to execute.

7.3.2 Sorted array implementation of a priority queue

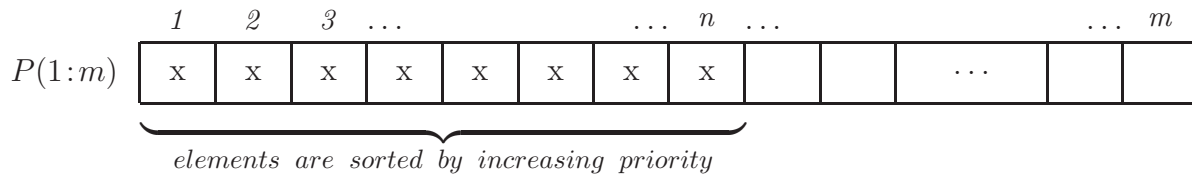


Figure 7.14 Sorted array implementation of $\mathbb{P} = [P(1:m), n]$

```

1  procedure PQ_INSERT( $\mathbb{P}, x$ )
2  if  $n = m$  then call PQ_OVERFLOW
3   $i \leftarrow n$ 
4  while  $i > 0$  and  $P(i) > x$  do
5       $P(i+1) \leftarrow P(i)$ 
6       $i \leftarrow i - 1$ 
7  endwhile
8   $P(i+1) \leftarrow x$ 
9   $n \leftarrow n + 1$ 
10 end PQ_INSERT

```

Procedure 7.9 Insert operation on a priority queue (sorted array implementation)

The new element x is inserted into a position such that the resulting sequence remains sorted by increasing priority. Note the similarity between lines 4–8 in the above procedure and lines 4–7 of procedure INSERTION_SORT in Session 3. In the worst case, all n elements will have to be moved up by one cell to make room for x in cell 1. It is clear that Procedure 7.9 takes $O(n)$ time to execute.

```

1  procedure PQ_EXTRACT( $\mathbb{P}, x$ )
2  if  $n = 0$  then call PQ_UNDERFLOW
3      else [  $x \leftarrow P(n)$ 
4              $n \leftarrow n - 1$  ]
5  end PQ_EXTRACT

```

Procedure 7.10 Extract operation on a priority queue (sorted array implementation)

With the elements sorted by increasing priority, the element with highest priority is stored in cell n . This is simply retrieved into x (line 3) and the size of the priority queue subsequently updated (line 4). Procedure 7.10 obviously takes $O(1)$ time to execute.

7.3.3 Heap implementation of a priority queue

The preceding two implementations of a priority queue are pretty straightforward but not quite satisfactory, since although one of the two basic operations on the queue takes $O(1)$ time, the other operation takes $O(n)$ time. The heap provides the vehicle for an unusually elegant and efficient realization of a priority queue in which both the insert and extract operations take $O(\log n)$ time.

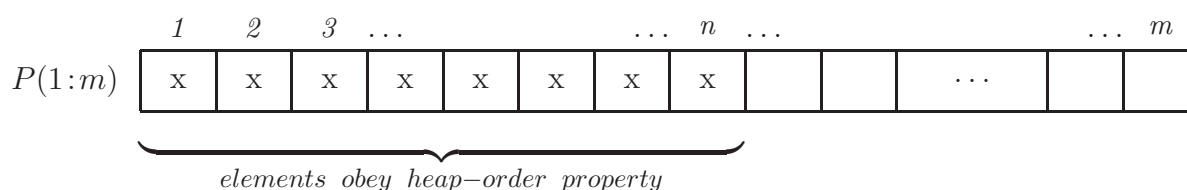


Figure 7.15 Heap implementation of $\mathbb{P} = [P(1:m), n]$

Shown in Figure 7.15 is a priority queue, implemented as a heap and stored in the sequential array P . Implied in the way the queue elements are stored in P is the father-son relationship between nodes in the heap in which the key in a son never exceeds the key in its father (the heap-order property). Thus the element with the largest key (highest priority) is at the root of the heap or in cell 1 of P .

In an insert operation, the new element x is initially stored in a new leaf at the bottommost level of the heap and is subsequently exchanged with smaller keys along an upward path towards the root such that the heap-order property is maintained. In the worst case the new element lands at the root of the heap, traversing a path of length equal to the height $h = \lfloor \log_2 n \rfloor$ of the heap. Thus the insert operation takes $O(\log n)$ time.

In an extract operation the element at the root of the heap is deleted and returned as the element with highest priority. Then, in a manner reminiscent of Step 3 of the heapsort algorithm, the rightmost leaf at the bottommost level of the heap is detached and its element stored at the vacated root of the binary tree which may no longer be a heap (it has become an almost-heap). To convert this almost-heap into a heap, HEAPIFY is invoked, in which the new key at the root is exchanged with larger keys along a downward path until the heap-order property is restored. In the worst case the key lands at the bottommost level of the heap, traversing a path of length equal to the height $h = \lfloor \log_2 n \rfloor$ of the heap. Thus the extract operation takes $O(\log n)$ time.

The final, and recommended, versions of PQ_INSERT and PQ_EXTRACT given below formalize the processes described above. The best way to fully understand and in the process appreciate the beauty of these procedures is to try them out, by hand, on a sample heap. Draw a sketch of *both* the binary tree and the array in which it is stored.

Writing the procedures to perform the other operations on a priority queue listed above (e.g., *pq_change*, *pq_return*, *pq_delete*) is left as an exercise. They should be coded for the heap representation of the priority queue.

```

1  procedure PQ_INSERT( $\mathbb{P}, x$ )
2  if  $n = m$  then call PQ_OVERFLOW
3   $n \leftarrow n + 1$ 
4   $i \leftarrow n$ 
5   $j \leftarrow \lfloor i/2 \rfloor$ 
6  while  $i > 1$  and  $P(j) < x$  do
7       $P(i) \leftarrow P(j)$ 
8       $i \leftarrow j$ 
9       $j \leftarrow \lfloor i/2 \rfloor$ 
10 endwhile
11  $P(i) \leftarrow x$ 
12 end PQ_INSERT

```

Procedure 7.11 Insert operation on a priority queue (heap implementation)

```

1  procedure PQ_EXTRACT( $\mathbb{P}, x$ )
2  if  $n = 0$  then call PQ_UNDERFLOW
3      else [  $x \leftarrow P(1)$ 
4               $P(1) \leftarrow P(n)$ 
5               $n \leftarrow n - 1$ 
6              call HEAPIFY( $\mathbb{P}, 1$ ) ]
7  end PQ_EXTRACT

```

Procedure 7.12 Extract operation on a priority queue (heap implementation)

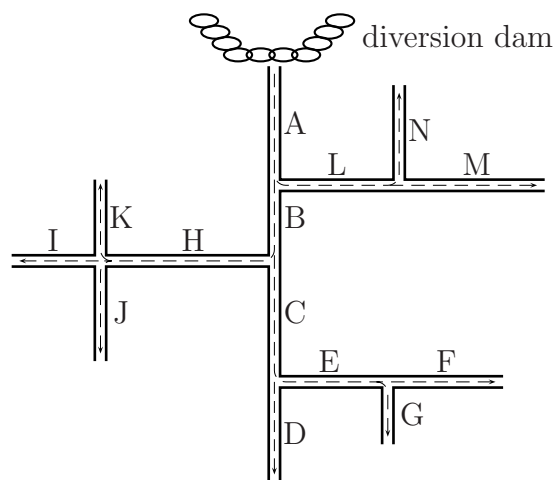
Summary

- Three different problems, all solved with the use of a binary tree, attest to its versatility as an implementing data structure. The problems are: calculating conveyance losses in an irrigation network, sorting keys on which total order has been defined, and implementing priority queues.
- A network of irrigation canals, no matter how complicated the topology, is suitably modelled by a binary tree represented using the linked design. The representation of the irrigation network in computer memory as a binary tree is constructed using preorder traversal as the template. Computation of the conveyance losses as water flows through the network is performed by traversing the binary tree in postorder, which implements in a most natural way the retrograde sequence in which the computations should be carried out.
- The heap, which structurally is a complete binary tree, lies at the heart of heapsort, an in-place $O(n \log n)$ sorting algorithm. Sift-up, the process which underlies heapsort, is efficiently carried out with the binary tree represented using the sequential design. Heapsort is a fast sorting algorithm, and a truly elegant one as well.

- Of the three sequential representations of a priority queue (as an unsorted, sorted or heap-ordered array), the last which utilizes a binary tree as the organizing data structure yields $O(\log n)$ implementations of the two basic operations (insert and extract) on a priority queue.

Exercises

1. Construct the binary tree representation of the irrigation network shown below.



2. Using a language of your choice, transcribe the EASY procedures CONSTRUCT (Procedure 7.1), CALCULATE (Procedure 7.2) and PRINT_IDR (Procedure 7.3) into a running program. Test your program using the irrigation network in Item 1 with the data below.

<i>Label</i>	<i>Length, km</i>	<i>FIR, cms</i>
A	1.40	0.131
B	0.80	0.043
C	1.45	0.140
D	1.21	0.098
E	1.33	0.118
F	1.42	0.134
G	0.75	0.038
H	2.10	0.294
I	1.12	0.084
J	1.20	0.096
K	1.00	0.067
L	1.55	0.160
M	1.86	0.231
N	1.23	0.101

3. Where is the second largest key in a max-heap? Where is the third largest key in a max-heap?
4. T is a *complete* binary tree on n nodes.
 - (a) How many levels does T have?
 - (b) If the nodes of T are labeled from 1 thru n in level order, on what level does node i , $1 \leq i \leq n$, reside?
5. The alphabetic keys **C**, **Y**, **B**, **E**, **R**, **T**, **A**, **L** and **K** are stored in the nodes of a sequentially allocated complete binary tree on 9 nodes in level order. Apply the heapsort algorithm and show the heap and queue at each stage of the process in the manner of Figure 7.12.
6. Using a language of your choice, transcribe the EASY procedures HEAPIFY (Procedure 7.5) and HEAPSORT (Procedure 7.6) into a running program. To test your program, take as input the course table for UP Iloilo on page 393. Sort the table:
 - (a) by course title
 - (b) by quota
 - (c) by predictor
7. Generalize your program in Item 6 such that it sorts on multiple keys. To test your program, take as input the course table for UP Iloilo on page 393. Sort the table:
 - (a) by quota by course title
 - (b) by predictor by course title
 - (c) by predictor by quota by course code
8. A priority queue is implemented using an unsorted array. Show the priority queue that results after inserting the instruction codes shown below into an initially empty priority queue in the order as enumerated through 10 calls to PQ_INSERT (Procedure 7.7). Assume that priorities are assigned in alphabetic order, i.e., XI has highest priority and AP has lowest priority.

CL BC DR MH AP OI XI EX ST TM
9. Show the priority queue that results after 2 calls to PQ_EXTRACT (Procedure 7.8) on the priority queue in Item 8.
10. A priority queue is implemented using a binary heap. Show the priority queue that results after inserting the instruction codes shown below into an initially empty priority queue in the order as enumerated through 10 calls to PQ_INSERT (Procedure 7.11). Assume that priorities are assigned in alphabetic order, i.e., XI has highest priority and AP has lowest priority.

CL BC DR MH AP OI XI EX ST TM
11. Show the priority queue that results after 2 calls to PQ_EXTRACT (Procedure 7.12) on the priority queue in Item 10.

12. Some applications of priority queues require an operation that *changes* the priority of an element in the queue, either by increasing it or by decreasing it. Write an EASY procedure to perform the operation of increasing the priority of an element in a priority queue that is implemented using a max-heap.

Bibliographic Notes

The original solution to the problem of computing conveyance losses in an irrigation network (see Pacheco, E. S., *et. al.* *Computerized Diversion Control Programs: Upper Pampanga River Project*. Report No. 20, NHRC, Diliman, Q.C., 1978) used a generalized list, instead of a binary tree, as the implementing data structure. On hindsight, this author realized that a binary tree is the more natural model. The results of the study showed that expressing the conveyance loss along an unlined canal as a fraction of the flow through the canal overly overestimated the losses.

The references cited in the Readings present and analyze the heapsort algorithm in quite different ways; we have adopted that given in STANDISH[1980], pp. 88–92. A more rigorous analysis of the algorithm is given in KNUTH3[1998], pp. 152–155; see also WEISS[1997], pp. 228–230.

Procedure HEAPIFY is an implementation of Subroutine 3.11 (*Sift-up*) and procedure HEAPSORT is an implementation of Algorithm 3.11 (*Heapsort*) from STANDISH[1980], albeit with a minor change, viz., the binary tree is converted into a heap, not into an almost-heap, at the outset (see also HEAPSORT(*A*) in CORMEN[2001], p. 136). This mirrors the natural flow of the computations although it results in an extra, trivial, call to HEAPIFY when the size of the binary tree is already one.

The (classic) binary heap implementation of a priority queue is due to Williams. Other implementations of a priority queue abound in the CS literature; these include *leftist heaps*, *skew heaps*, *binomial queues*, *Fibonacci heaps*, and so on. The interested reader is referred to WEISS[1997].

SESSION 8

Trees and forests

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define and give examples of an ordered tree, oriented tree and free tree.
2. State important properties of trees and forests.
3. Define and illustrate natural correspondence.
4. Traverse any given ordered forest in preorder and postorder.
5. Explain implementation issues pertaining to the linked and sequential representation of trees and forests.
6. State the equivalence problem and explain how a forest of oriented trees is utilized to model and solve the problem.
7. Explain and illustrate the weighting rule for union and the collapsing rule for find.
8. Explain the effect of these rules on the solution to the equivalence problem.

READINGS KNUTH1[1997] pp. 308–311, 334–355; STANDISH[1980], pp. 67–73; AHO[1974], pp. 129–139; HOROWITZ[1976], pp. 248–257; CORMEN[2001], pp. 498–522; WEISS[1997], pp. 261–279.

DISCUSSION

Somewhat related to the binary tree but fundamentally a different sort of structure altogether is the ADT called a **tree**. Some very important applications of trees include (a) representation of sets (e.g., for the efficient implementation of the union and find operations on sets), (b) representation of general expression (expression trees) (c) decision making (decision trees) (d) game playing (game trees) (e) searching (B-trees, tries, etc.) (f) graph traversals (spanning trees), and the like.

In this session, we will consider three species of trees, namely: *ordered*, *oriented* and *free*.

8.1 Definitions and related concepts

We begin our study of trees by defining the most ‘constrained’ of the three, which is the ordered tree. To this end, we adopt the following definition from KNUTH1[1997], p. 308:

DEFINITION 8.1. An **ordered tree** is a finite set, say T , of one or more nodes such that there is a specially designated node called the root and the remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, T_2, \dots, T_m , where each of these sets is an ordered tree. T_1, T_2, \dots, T_m are called the subtrees of the root.

Note that whereas a binary tree may be empty, an ordered tree is never empty. Further, a node in a binary tree may have at most two subtrees (called the left subtree and the right subtree to distinguish one from the other); a node in an ordered tree may have any number of subtrees.

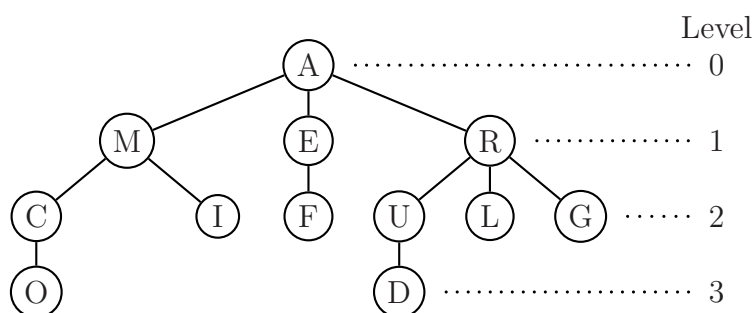


Figure 8.1 An ordered tree

Figure 8.1 depicts an ordered tree. Node A is the root and the ordered trees rooted at M, E and R are its subtrees. Similarly, the trees rooted at C and I are the subtrees of node M. And so on. The concepts of *degree*, *level* and *height* defined in Session 6 for binary trees apply in an analogous way to trees. Recall that the degree of a node is the number of subtrees of the node; in a binary tree this may not exceed 2, but in a tree it may. Thus the degree of nodes A and R is 3; that of node M is 2; that of nodes C, E and U is 1; the rest of the nodes have degree 0. A node with degree zero is called a *terminal node* or a *leaf*. A line joining two nodes is called a *branch*. The degree of a node is also the number of branches emanating from it. The *degree of a tree* is defined as the degree of its node(s) of highest degree. The ordered tree of Figure 8.1 has degree 3.

As with binary trees, the level of a node is defined by specifying that the root of the tree is at level 0, the roots of its subtrees are at level 1, and so on. The *height* of a tree is the level of its bottommost node(s); thus the height of the tree in Figure 8.1 is 3.

As with binary trees, spatial and familial terms are also used to describe trees and operations on trees. In the ordered tree of Figure 8.1, node A is the father of nodes M, E and R. Nodes M, E and R are the sons of node A, and are therefore brothers. The leftmost son of a node is considered to be the oldest son, and the rightmost son is the youngest in the brood. Ancestors and descendants are defined as before.

If the order of the subtrees of any node in a tree is immaterial, then we obtain an **oriented tree**. Thus trees T_1 and T_2 shown in Figure 8.2 are two different ordered trees but the same oriented tree.

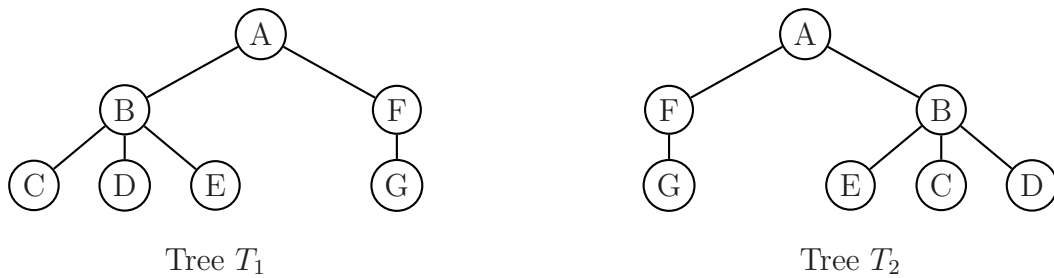


Figure 8.2 Two different ordered trees but the same oriented tree

If we do not care to specify any node of a tree as a root node, such that orientation and order cease to be meaningful, then we obtain a **free tree**. Depicted below is a free tree.

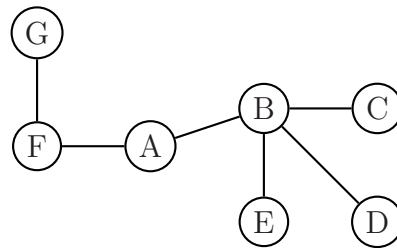


Figure 8.3 A free tree

Free trees, oriented trees and ordered trees constitute a progression from a less to a more constrained kind of tree. In Figure 8.4, choosing A to be the root node assigns an orientation to each branch in the tree, and a relation among the nodes based on how far removed a node is from the root, yielding an oriented tree. Nodes which are one branch removed from the root are the sons of the root, two branches away are grandsons, etc. Note that there is no implied ordering of the sons, grandsons, and so on. Picking up the tree by its root and letting its subtrees dangle below assigns an ordering to the subtrees from leftmost to rightmost, which if relevant, yields an ordered tree.

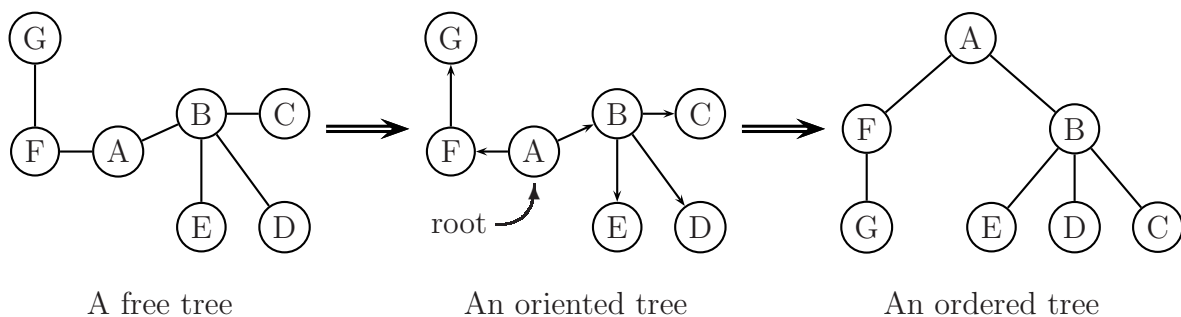


Figure 8.4 Obtaining orientation and order

Figure 8.5 further illustrates the distinction among these three species of trees. The figure shows the five distinct ordered trees on four unlabeled nodes; on the same set of nodes there are four distinct oriented trees and two distinct free trees.

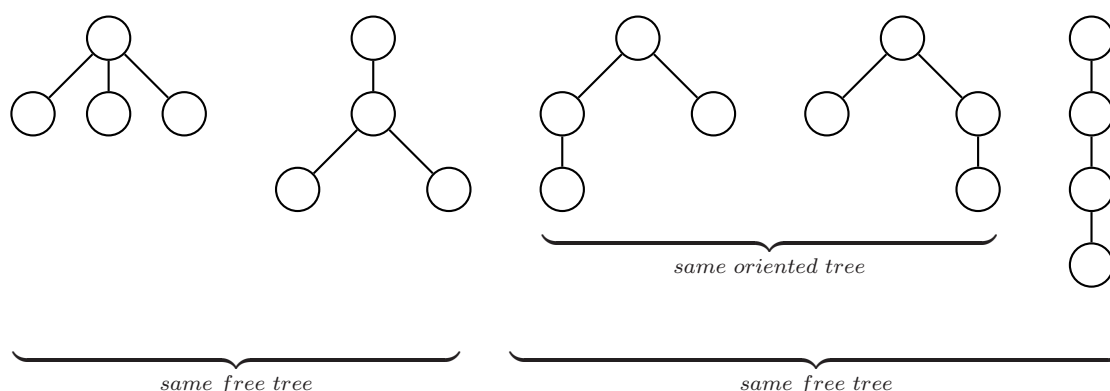


Figure 8.5 Ordered, oriented and free trees on four unlabeled nodes

A *forest* is a set of zero or more disjoint trees. Thus we may have a forest with no tree at all (much like our natural forests today). If the trees comprising a forest are ordered trees and if their order in the forest is relevant then we get an ordered forest. On the other hand, if the trees are oriented and their order in the forest is irrelevant then we get a forest of oriented trees. Shown below is a forest of three trees.

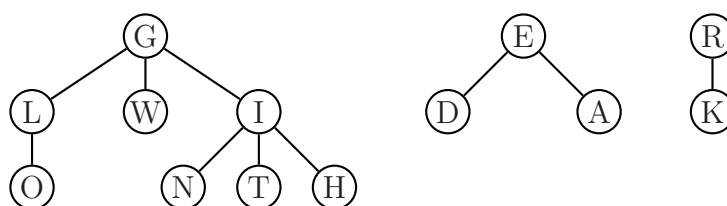


Figure 8.6 A forest of trees

If we delete the root of a tree, we obtain a forest, and if we add a node to a forest and take this node to be the root, we get a tree. A single tree is a forest of one tree. We often use the terms tree and forest interchangeably.

8.2 Natural correspondence

There is a well defined process which transforms an ordered forest, say F , into a unique binary tree, $B(F)$, and vice versa. Following Knuth, this transformation is called *natural correspondence*, and is defined as follows (KNUTH1[1997], p. 335):

DEFINITION 8.2. Natural correspondence

Let $F = (T_1, T_2, \dots, T_n)$ be an ordered forest of ordered trees. The binary tree $B(F)$ corresponding to F is obtained as follows: (a) If $n = 0$, then $B(F)$ is empty. (b) If $n > 0$, then the root of $B(F)$ is the root of T_1 ; the left subtree of $B(F)$ is $B(T_{11}, T_{12}, \dots)$ where T_{11}, T_{12}, \dots are the subtrees of the root of T_1 ; and the right subtree of $B(F)$ is $B(T_2, T_3, \dots, T_n)$.

The definition is recursive, but it can be applied in a straightforward manner to generate $B(F)$ given F . For instance, if the forest in Figure 8.6 is considered to be an ordered forest $F = (T_1, T_2, T_3)$, then the root of $B(F)$ is node G; the left subtree of $B(F)$ is the binary tree which corresponds to the ordered trees rooted at L, W and I; and the right subtree of $B(F)$ is the binary tree which corresponds to the trees T_2 and T_3 . The root of the left subtree of $B(F)$ is L; the left subtree of L is the binary tree rooted at O; and the right subtree of L is the binary tree which corresponds to the ordered trees rooted at W and I. And so on.

Note that in the above transformation, the oldest son of a node in F becomes its left son in $B(F)$; the next younger brother of a node in F becomes its right son in $B(F)$. If a node has no son in F , then its left son in $B(F)$ is null; and if a node has no younger brother in F , then its right son in $B(F)$ is null. Based on these observations, we can construct $B(F)$ from F without directly applying the recursive definition of natural correspondence, as follows:

1. Link together the sons in each family from left to right. (Note: the roots of the trees in the forest are brothers, sons of an unknown father.)
2. Remove links from a father to all his sons except the oldest (or leftmost) son.
3. Tilt the resulting figure by 45° .

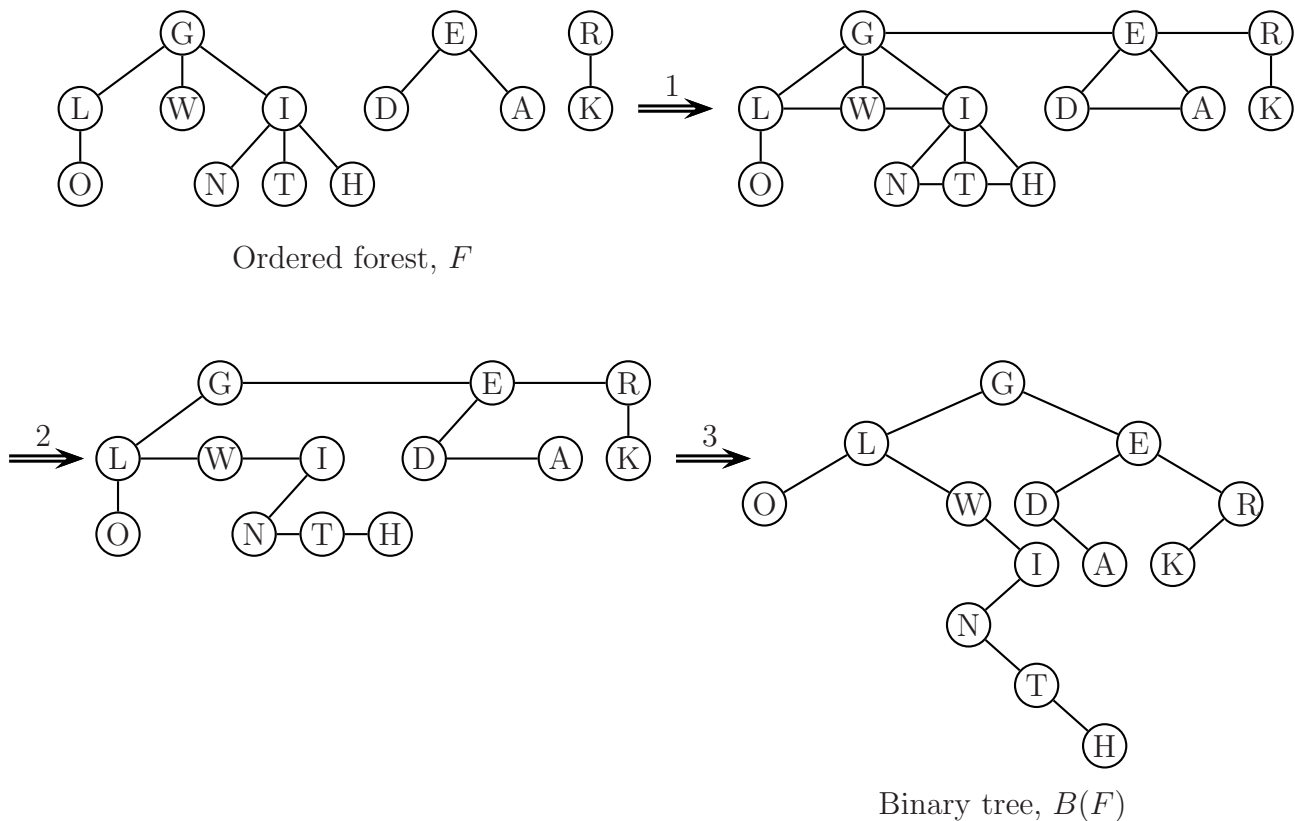


Figure 8.7 Natural correspondence

It is obvious from this example that the process is reversible; given $B(F)$, we can readily obtain F .

8.3 Forest traversal

If ‘root’ is replaced by ‘root of the first tree’, ‘left subtree’ by ‘subtrees of the first tree’ and ‘right subtree’ by ‘remaining trees’ in the definitions of the traversal orders for binary trees (see section 6.2 of Session 6), we obtain what most authors (HOROWITZ[1976], TENENBAUM[1986], AHO[1983], among others) consider as the definitions of the corresponding traversal orders for ordered forests. Applying these definitions on the forest F of Figure 8.7, we obtain the following sequences (verify):

Forest preorder: G L O W I N T H E D A R K

Forest inorder: O L W N T H I G D A E K R

Forest postorder: O H T N I W L A D K R E G

These sequences actually correspond to the preorder, inorder and postorder traversal of $B(F)$.

In preorder, a node in a binary tree immediately precedes its descendants. This is also the case in forest preorder, as defined above. For example, node G immediately precedes nodes L, W, I, O, N, T and H, which are its descendants; node I precedes nodes N, T and H; and so on. Thus, for the binary tree and the forest, the prefix ‘pre’ in preorder is appropriate.

In inorder, a node in a binary tree is preceded by its left descendants (if any) and followed by its right descendants (if any), hence the prefix ‘in’. In an ordered forest there is no obvious ‘middle’ position of a node among its descendants. In fact, in the forest inorder sequence given above, a node immediately follows *all* its descendants (if any). For instance, node G immediately follows nodes L, W, I, O, N, T and H, which are its descendants; node I follows nodes N, T and H; and so on. This is precisely the case with the postorder enumeration of the nodes of a binary tree, hence the prefix ‘post’. Thus, the middle sequence shown above is more appropriately called postorder, rather than inorder, and the third sequence called by some other name (postorder_2 or postorder_too?) or dropped altogether.

Following KNUTH1[1997], p. 336, we therefore define two basic traversal methods for ordered forests as follows:

DEFINITION 8.3. If the ordered forest is empty consider the traversal as ‘done’;
else

Preorder traversal

Visit the root of the first tree.
Traverse the subtrees of the first tree in preorder.
Traverse the remaining trees in preorder.

Postorder traversal

Traverse the subtrees of the first tree in postorder.
Visit the root of the first tree.
Traverse the remaining trees in postorder.

Applying these definitions on the forest F of Figure 8.7, we obtain the following sequences:

Preorder: G L O W I N T H E D A R K

Postorder: O L W N T H I G D A E K R

In preorder, a node is listed immediately before its descendants; in postorder, a node is listed immediately after its descendants. These sequences are the same as the preorder and inorder sequences, respectively, of the nodes of the corresponding binary tree, $B(F)$.

Knuth points out that preorder may also be called *dynastic order*: if we consider the trees in a forest as comprising a royal family, the preorder sequence is the equivalent of the order of succession to the throne.

Preorder and postorder traversal of the tree representation of a general algebraic expression yield the familiar prefix and postfix forms of the expression. (This shows once more why inorder is not an apt name for the latter sequence.) Recall that an expression in prefix or postfix form can be evaluated by scanning the expression only once from left to right. Specifically, in postfix form, each subexpression of the form *operand-unary operator* or *operand-operand-binary operator* is evaluated and replaced by the result (an operand) as the expression is scanned from left to right.

Figure 8.8 shows the tree representation of the familiar expression $(-b + \sqrt{b^2 - 4ac})/2a$. If now we traverse the tree in preorder and postorder we obtain the sequences

Preorder: / + - b $\sqrt{}$ - ^ b 2 * * 4 a c * 2 a

Postorder: b - b 2 ^ 4 a * c * - $\sqrt{}$ + 2 a * /

which are the prefix and postfix forms of the expression, as you may readily verify.

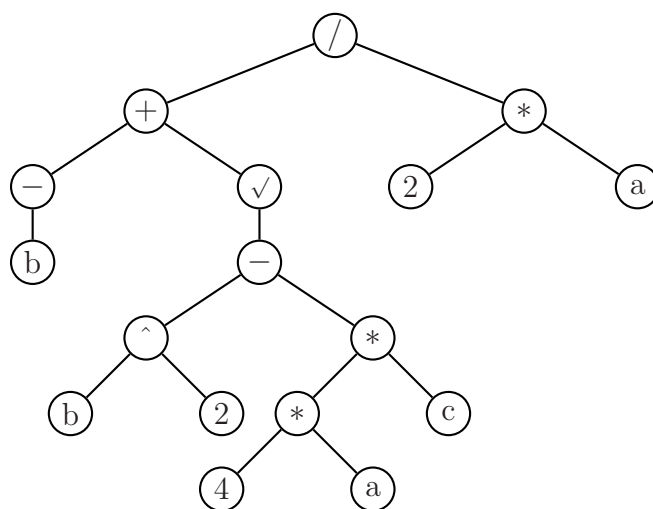
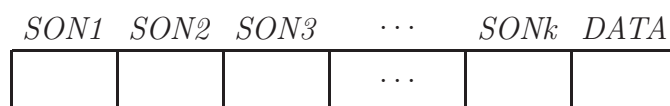


Figure 8.8 Expression tree for $(-b + \sqrt{b^2 - 4ac})/2a$

8.4 Linked representation for trees and forests

The most natural way to represent a tree in computer memory is to use the node structure



where k is the degree of the tree and $SON1, SON2, \dots, SON_k$ are links to the k possible sons of a node. As with binary trees, this representation mimics the way we draw a tree on paper and certain operations on trees are easier to conceptualize and implement with this representation. Clearly, however, much of the allocated link space is occupied by null links. Given a tree on n nodes and with degree k , the number of link fields occupied by Λ is

$$\text{number of null links} = nk - (n - 1) = n(k - 1) + 1$$

\uparrow
 \downarrow

\uparrow
 \downarrow

$\text{total number of links}$

$\text{number of non-null links} = \text{number of branches}$

This means that for a tree of degree 3, about 67% of the link space is occupied by Λ ; for a tree of degree 10, 90% of it is occupied by Λ .

A less wasteful way to represent a tree using linked allocation is to use the node structure

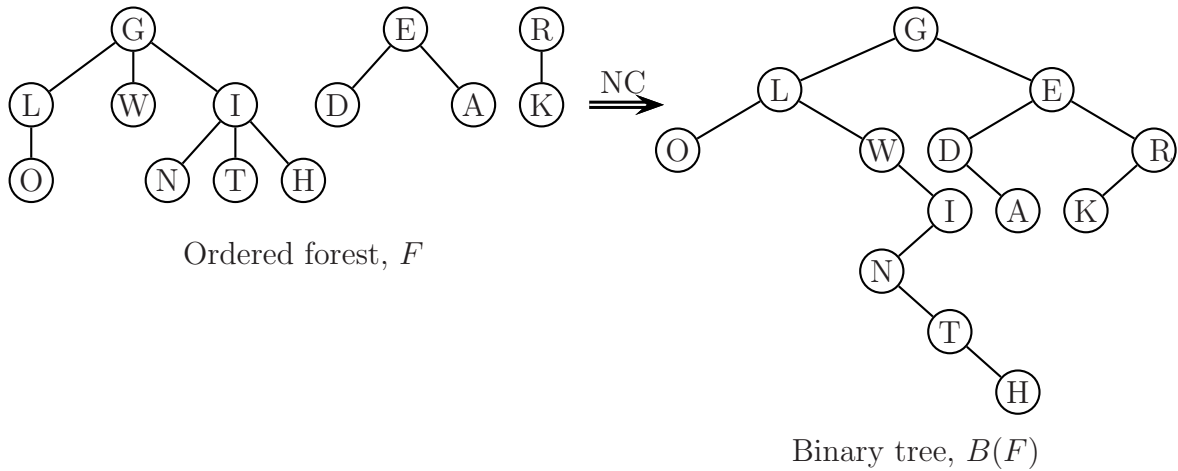


where *LLINK* points to the leftmost son of a node, if it exists, otherwise *LLINK* is Λ ; and *RLINK* points to the next younger brother, if it exists, otherwise *RLINK* is Λ . This is the equivalent of representing a tree by its corresponding binary tree.

8.5 Sequential representation for trees and forests

There are various sequential representation schemes which have been devised for forests. The challenge is to find representation schemes which require the least amount of storage to encode the forest structure, while allowing for the efficient implementation of the pertinent algorithms on forests in a given application. In this section we will consider two classes of such representation methods, namely, representations which utilize *links* and *tags* (see KNUTH1[1997], pp. 348–351) and representations which utilize *arithmetic tree information* (see STANDISH[1980], pp. 71–73).

To illustrate some of these sequential representation methods, we will use the forest F in Figure 8.7, reproduced below for quick reference, along with its corresponding binary tree, $B(F)$.



8.5.1 Representing trees and forests using links and tags

1. Preorder sequential representation

The figure below shows graphically the preorder sequential representation of the forest F shown above.

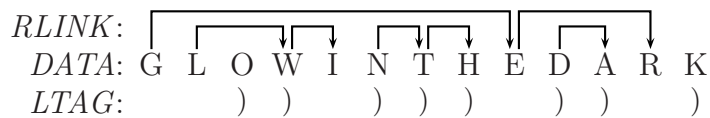


Figure 8.9 Preorder sequential representation of an ordered forest (version 1)

$RLINK$ is a pointer from an older to a next younger brother in F , or from a father to his right son in $B(F)$. Since in preorder, an oldest (or leftmost) son in F immediately comes after his father, there is no need for a pointer from father to leftmost son. However, a node which has no son in F must be identified, so that we will know that what comes next in preorder is not a leftmost son, but the younger brother of an earlier node. This is the function of $LTAG$, where a tag of ‘)’ indicates a terminal node in F , or a node with no left son in $B(F)$. For instance, node G is not tagged, so the node which follows (i.e., node L) is its leftmost son; node H is tagged, so the node which follows (i.e., node E) is not its son, but is rather the younger brother of a previous node (which is node G).

Actually, we can encode the forest F with lesser data. This claim is based on the following observations regarding Figure 8.9.

- (a) $RLINK$ arrows do not cross each other; thus the last arrow to begin is the first arrow to end.
- (b) An arrow always points to a node which immediately follows one which is tagged with a ‘)’.

These observations follow from fact that in preorder all the descendants of a node are enumerated first before its next younger brother. Thus we can either:

(a) eliminate *LTAG*, which identifies terminal nodes, since a terminal node always immediately precedes a node pointed to by an arrow, or is the last node in the sequence or

(b) replace *RLINK*, which indicates where an arrow begins and where it ends, with *RTAG* which simply indicates where an arrow begins. *RTAG* suffices since we know that an arrow always points to a node immediately following one which is tagged with a ‘), and that the last arrow to begin is the first one to end.

Of these two alternatives, the second obviously results in more saving in terms of storage used, since both *LTAG* and *RTAG* can be maintained as bit arrays. This alternative version of preorder sequential representation is depicted in Figure 8.10.

```

RTAG: ( ( ( ( ( ( ( (
DATA: G L O W I N T H E D A R K
LTAG:      ) )      ) ) )      ) )      )

```

Figure 8.10 Preorder sequential representation of an ordered forest (version 2)

To implement the preorder sequential representation of a forest F in computer memory, we use three parallel arrays as shown below. In version 1, a *RLINK* arrow which points to a next younger brother in F is replaced by the address (index) of the next younger brother; in version 2, a *RTAG* = ‘(’ which indicates an older brother in F (or a node with a right son in $B(F)$) is denoted by a 1. In both versions, a *LTAG* = ‘)’ which indicates a terminal node in F (or a node with no left son in $B(F)$) is denoted by a 0.

(a) Version 1 of preorder sequential representation

	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>RLINK</i> :	9	4	0	5	0	7	8	0	12	11	0	0	0
<i>DATA</i> :	G	L	O	W	I	N	T	H	E	D	A	R	K
<i>LTAG</i> :	1	1	0	0	1	0	0	0	1	0	0	1	0

(b) Version 2 of preorder sequential representation

	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>RTAG</i> :	1	1	0	1	0	1	1	0	1	1	0	0	0
<i>DATA</i> :	G	L	O	W	I	N	T	H	E	D	A	R	K
<i>LTAG</i> :	1	1	0	0	1	0	0	0	1	0	0	1	0

Figure 8.11 Implementing the preorder sequential representation of an ordered forest

2. Family-order sequential representation

Figure 8.12 shows the family-order sequential representation of the forest F in Figure 8.7.

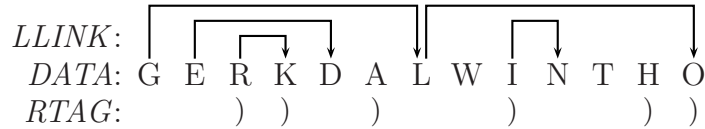


Figure 8.12 Family-order sequential representation of an ordered forest (version 1)

In family-order, a family is a brood of brothers. The first family to be listed consists of the root nodes of all trees in the forest (recall that the root nodes are brothers whose father is unknown). Subsequently, families are listed on a last-in first-out basis. Since brothers are enumerated consecutively, there is no need for a pointer from an older to a next younger brother; however, we need a pointer, say *LLINK*, from a father to his leftmost son in F (or left son in $B(F)$). Likewise, we provide a tag, say *RTAG*, to identify the youngest brother in a brood so that we know that what follows in family-order is not a younger brother but a leftmost son of a previous node. For instance, node G is not tagged, so the node which follows (i.e., node E) is its next younger brother; node A is tagged, so the node which follows (i.e., node L) is not its younger brother, but is rather the leftmost son of a previous node (which is node G).

In a manner analogous to preorder sequential representation, *LLINK* arrows do not cross each other, and an arrow always points to a node immediately following one which is tagged with a ')'. Hence, we can either eliminate *RTAG* (since we can deduce from *LLINK* where the ') will be), or replace *LLINK* with *LTAG* to indicate where *LLINK* begins (since we can deduce from *RTAG* where *LLINK* ends). As in preorder sequential representation, we choose the latter option to obtain version 2 of family-order sequential representation.



Figure 8.13 Family-order sequential representation of an ordered forest (version 2)

To implement the family-order sequential representation of a forest F in computer memory, we use three parallel arrays as shown below. In version 1, a *LLINK* arrow from a father to its leftmost son is replaced by the address (index) of the son; in version 2, a $LTAG = '('$ which indicates a father in F (or a node with a left son in $B(F)$) is denoted by a 1. In both versions, a $RTAG = ')'$ which indicates a rightmost brother in F (or a node with no right son in $B(F)$) is denoted by a 0.

(a) Version 1 of family-order sequential representation

	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>LLINK</i> :	7	5	4	0	0	0	13	0	10	0	0	0	0
<i>DATA</i> :	G	E	R	K	D	A	L	W	I	N	T	H	O
<i>RTAG</i> :	1	1	0	0	1	0	1	1	0	1	1	0	0

(b) Version 2 of family-order sequential representation

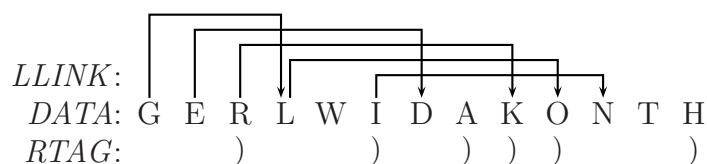
	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>LTAG</i> :	1	1	1	0	0	0	1	0	1	0	0	0	0
<i>DATA</i> :	G	E	R	K	D	A	L	W	I	N	T	H	O
<i>RTAG</i> :	1	1	0	0	1	0	1	1	0	1	1	0	0

Figure 8.14 Implementing the family-order sequential representation of an ordered forest

An immediate consequence of using version 2 in lieu of version 1 for either preorder or family-order sequential representation is some loss of flexibility when processing the forest. It is obvious that we need to start at the root node of the first tree each time and that we need to keep in a stack those nodes from which arrows emanate until such time that we get to the nodes to which the arrows point, in order to establish the older-to-younger-brother or father-to-first-son relationship in F .

3. Level-order sequential representation

Figure 8.15 shows the level-order sequential representation of the forest F in Figure 8.7.

**Figure 8.15** Level-order sequential representation of an ordered forest (version 1)

In level order, as in family order, brothers are listed consecutively; thus *LLINK* and *RTAG* are as defined in family order.

Unlike in preorder or family-order sequential representation, arrows cross each other; however, it is still the case that an arrow points to a node immediately following one which is tagged with a ')'. Likewise, although arrows cross there is still a pattern in the way they behave, which may be characterized as first-in, first-out; i.e., the first arrow to begin is the first arrow to end. Hence, as with family order sequential representation, we can either remove *RTAG* or replace *LLINK* with *LTAG*, to obtain a more economical representation of the forest (at the price of less flexibility, as already explained). This time, we need to use a queue, instead of a stack, to establish the relation that the replaced *LLINK* would have explicitly specified.

```

LTAG: ( ( ( ( (
DATA: G E R L W I D A K O N T H
RTAG:      )      )      )      )      )

```

Figure 8.16 Level-order sequential representation of an ordered forest (version 2)

The corresponding internal representations of Figures 8.15 and 8.16 are easily deduced and are left as exercises.

Converting from sequential to linked representation

We consider now an implementation of the transformation called natural correspondence. Any of the above sequential schemes can be used to represent the forest F from which the linked representation of the corresponding binary tree $B(F)$ can be constructed.

The EASY procedure CONVERT generates $B(F)$, represented by the data structure $\mathbb{B} = [(LLINK, INFO, RLINK), T]$ from the preorder sequential representation of F , as defined by the data structure $\mathbb{F} = [RTAG(1:n), DATA(1:n), LTAG(1:n)]$. CONVERT uses a stack \mathbb{S} to store nodes with a $RTAG = 1$, i.e., nodes which have right subtrees in $B(F)$ awaiting construction. The pointer to the generated binary tree is T .

```

▷ Generates the linked representation of  $B(F)$  from the preorder sequential
▷ representation of  $F$ . Pointer to  $B(F)$  is  $T$ .
1  procedure CONVERT( $\mathbb{F}, \mathbb{B}$ )
▷ Set up root node
2    call GETNODE( $\alpha$ );  $T \leftarrow \alpha$ 
▷ Generate rest of binary tree
3    call InitStack( $\mathbb{S}$ )
4    for  $i \leftarrow 1$  to  $n - 1$  do
5       $INFO(\alpha) \leftarrow DATA(i)$ 
6      call GETNODE( $\beta$ )
7      if  $RTAG(i) = 1$  then call PUSH( $\mathbb{S}, \alpha$ )
8      else  $RLINK(\alpha) \leftarrow \Lambda$ 
9      if  $LTAG(i) = 0$  then [  $LLINK(\alpha) \leftarrow \Lambda$ ; call POP( $\mathbb{S}, \sigma$ );  $RLINK(\sigma) \leftarrow \beta$  ]
10     else  $LLINK(\alpha) \leftarrow \beta$ 
11      $\alpha \leftarrow \beta$ 
12  endfor
▷ Fill in fields of rightmost node
13   $INFO(\alpha) \leftarrow DATA(n)$ 
14   $LLINK(\alpha) \leftarrow RLINK(\alpha) \leftarrow \Lambda$ 
15  end CONVERT

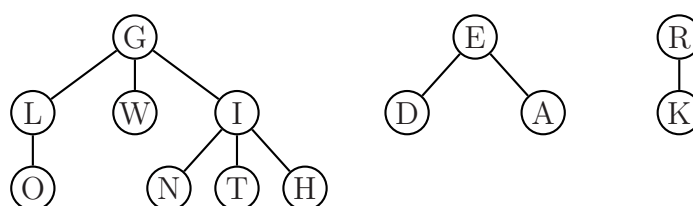
```

Procedure 8.1 An implementation of natural correspondence

Walk through procedure CONVERT using as input the forest F of Figure 8.11(b) and verify that it yields the binary tree $B(F)$ of Figure 8.7. Analogous procedures to construct $B(F)$ given the family-order or level-order sequential representation of F are easily written. These are left as exercises.

8.5.2 Arithmetic tree representations

Another way of encoding tree structure is to enumerate the nodes of the tree in preorder, postorder or level order along with arithmetic information pertaining to each node, specifically, the *degree* or the *weight* of the node; following STANDISH[1980], pp. 71–73, these are referred to as *arithmetic tree representations*. The degree of a node is the number of sons of the node; the weight of a node is the number of descendants of the node. These representation schemes apply to forests as well. Once more, we will use the forest F of Figure 8.7, reproduced below for quick reference, to illustrate this other class of sequential representation methods for trees and forests.



An ordered forest F of ordered trees

1. Preorder sequence with degrees

DATA: G L O W I N T H E D A R K
DEGREE: 3 1 0 0 3 0 0 0 2 0 0 1 0

Without looking at the figure, construct the forest from the given *DATA* and *DEGREE* sequences. It should become immediately clear that there is only one forest which fits the given data. Stated another way, the *DATA* and *DEGREE* sequences uniquely define the forest F .

2. Preorder sequence with weights

DATA: G L O W I N T H E D A R K
WEIGHT: 7 1 0 0 3 0 0 0 2 0 0 1 0

Do the same exercise on the given *DATA* and *WEIGHT* sequences and verify that indeed they uniquely define the forest F .

3. Postorder sequence with degrees

DATA: O L W N T H I G D A E K R
DEGREE: 0 1 0 0 0 0 3 3 0 0 2 0 1

4. Postorder sequence with weights

DATA: O L W N T H I G D A E K R
WEIGHT: 0 1 0 0 0 0 3 7 0 0 2 0 1

Verify once more that the given sequences generate the forest F , and this forest only. To construct F , process the sequences right to left.

5. Level order sequence with degrees

DATA: G E R L W I D A K O N T H
DEGREE: 3 2 1 1 0 3 0 0 0 0 0 0 0

6. Level order sequence with weights

DATA: G E R L W I D A K O N T H
WEIGHT: 7 2 1 1 0 3 0 0 0 0 0 0 0

Verify once more that the given sequences generate the forest F , and this forest only. To construct F , you need to first determine how many trees there are in the forest. This is left as an exercise.

Each of the six representation schemes shown above can be implemented using two parallel arrays of size n , where n is the number of nodes in the forest. For instance the data structure $\mathbb{PD} = [DATA(1:n), DEGREE(1:n)]$ shown below is a straightforward realization of preorder sequence with degrees.

	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>DATA:</i>	G	L	O	W	I	N	T	H	E	D	A	R	K
<i>DEGREE:</i>	3	1	0	0	3	0	0	0	2	0	0	1	0

Choosing which representation to use

Since all six representations utilize the same amount of space the primary consideration in choosing which scheme to use is the ease with which the intended operations on the tree or forest can be carried out. For example, suppose that a tree is used to represent some algebraic expression and that a common operation is replacing a subexpression with another subexpression. In terms of the tree, a subexpression is a subtree rooted at some internal node, say node i (here i is the index into the array in which the node is stored). Thus the operation involves: (a) locating the subtree rooted at node i , and (b) replacing this subtree with another subtree (of a possibly different weight).

In preorder sequence with weights, it is easy to find the subtree rooted at node i , since if we let $m = WEIGHT(i)$, then the m nodes immediately after node i in preorder comprise the subtree rooted at node i . Thus, replacing the subtree rooted at node i with another subtree of the same weight is readily accomplished. However, deleting the subtree rooted at node i , or replacing it with another subtree of a different weight will require adjusting the weight of all the ancestors of node i .

In preorder sequence with degrees, determining the subtree rooted at node i requires computation, since it is not obvious from the degree of node i how many descendants it has. However, once found, deleting the subtree rooted at node i or replacing it with another subtree of the same or different weight can be done locally and requires no adjustment of the degree of any other node in the tree.

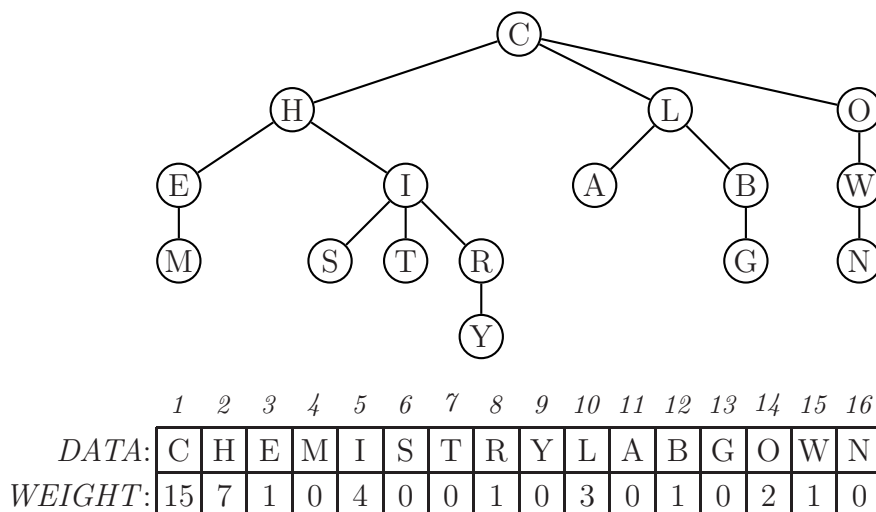
Preorder sequence may appear to be the ‘natural’ way in which to process the information in the nodes of a tree. However, when we have a function, say f , defined on the nodes of a tree such that the value of f at some node i depends on the data local to node i and on the values of f on the sons of node i (recall, for instance, the problem of calculating conveyance losses in Session 7), then the postorder representation schemes 3 and 4 are

clearly more suitable. Algorithm F (*Evaluate a locally defined function in a tree*) given in KNUTH1[1997], pp. 351–352 performs such an evaluation for a tree represented in postorder with degrees.

Some algorithms on arithmetic tree representations

1. Finding the ancestors of node i in a tree or forest represented using preorder sequence with weights

We mentioned above that when the subtree rooted at some node in a tree represented in preorder sequence with weights is deleted or replaced by another subtree with a different weight, then the weight of all the ancestors of the node must be accordingly adjusted. The algorithm to find the ancestors of any given node in a tree is based on the fact that in preorder a node is immediately followed by all its descendants. Consider the tree shown below and its representation in preorder sequence with weights as implemented by the arrays *DATA* and *WEIGHT*. Take node H which is in cell 2. The descendants of H are in



cell $2 + 1 = 3$ through cell $2 + \text{WEIGHT}(2) = 9$, as you may easily verify. Thus, any node stored in a cell with index greater than 9 cannot have H as an ancestor. In general, a node in cell k (or node k , for brevity) is an ancestor of node i only if $k + \text{WEIGHT}(k) \geq i$.

The EASY procedure ANCESTORS formalizes what we just described. It accepts as input the data structure $\mathbb{PW} = [\text{DATA}(1:n), \text{WEIGHT}(1:n)]$ which represents a tree or forest on n nodes, and the index i of the node whose ancestors are to be found. The ancestors are stored in the *FATHER* array.

```

1  procedure ANCESTORS( $\mathbb{PW}, i, \text{FATHER}$ )
2  array FATHER(1: $n$ )
3  FATHER  $\leftarrow$  0     $\triangleright$  Initialize FATHER array
4   $j \leftarrow i$ 
5  for  $k \leftarrow i - 1$  to 1 by  $-1$  do
6      if  $k + \text{WEIGHT}(k) \geq i$  then [ FATHER( $j$ )  $\leftarrow$   $k$ ;  $j \leftarrow k$  ]
7  endfor
8  end ANCESTORS

```

Procedure 8.2 Finding ancestors in a tree

The figure below depicts the *FATHER* array generated by procedure ANCESTORS when invoked with $i = 9$ (i.e., node Y). The entries are read as follows: $FATHER(9) = 8$; $FATHER(8) = 5$; $FATHER(5) = 2$; $FATHER(2) = 1$; $FATHER(1) = 0$. Hence the ancestors of node Y are R, I, H and C.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
<i>FATHER</i> :	0	1	0	0	2	0	0	5	8	0	0	0	0	0	0	0

The time complexity of the procedure is $O(n)$. In the worst case node i may have all the other nodes as its ancestors.

2. Finding the degree of the nodes in a tree or forest represented using preorder sequence with weights

The algorithm to find the degree given the weight of the nodes of a forest, enumerated in preorder, is based on the following easily verifiable facts about any node, say node i , in the forest:

- (a) The degree of node i is equal to the weight of node i minus the sum of the weights of the sons of node i .
- (b) In preorder sequence, the first son of node i , if any, is node j where $j = i + 1$.
- (c) The succeeding sons of node i , if any, are readily located from the fact that if $m = WEIGHT(j)$, then the next m nodes are the descendants of node j . The next son of node i is the next node in preorder.

The EASY procedure DEGREES accepts as input the data structure $PW = [DATA(1:n), WEIGHT(1:n)]$ which represents a tree or forest on n nodes and generates the corresponding *DEGREE* array.

```

1  procedure DEGREES(PW, DEGREE)
2  array DEGREE(1:n)
3  for  $i \leftarrow 1$  to  $n$  do
4       $DEGREE(i) \leftarrow WEIGHT(i)$ 
5       $m \leftarrow WEIGHT(i)$ 
8       $j \leftarrow i + 1$   $\triangleright$  node  $j$  is first son of node  $i$  if  $m > 0$ 
6      while  $m > 0$  do
7           $DEGREE(i) \leftarrow DEGREE(i) - WEIGHT(j)$ 
8           $m \leftarrow m - (WEIGHT(j) + 1)$ 
9           $j \leftarrow j + WEIGHT(j) + 1$   $\triangleright$  node  $j$  is next son of node  $i$  if  $m > 0$ 
10     endwhile
11 endfor
12 end DEGREE

```

Procedure 8.3 Extracting degrees from weights

The time complexity of the procedure is clearly $O(n)$. Except for the root, each node is processed at most twice: first as a son, subsequently as a father.

8.6 Trees and the equivalence problem

An important problem that is encountered in certain areas of computer science is the so-called *equivalence problem*. For instance, FORTRAN provides the EQUIVALENCE statement, and the FORTRAN compiler must process a declaration such as

EQUIVALENCE (COUNT(1),QLINK(1)),(X(1,1),Y(2,10),Z(0))

Variants of the problem also arise in finding a minimum cost spanning tree for a weighted undirected graph, strongly connected components in a directed graph, and so on.

Recall from Session 2 that an *equivalence relation* is a binary relation R on a set, say S , that is *reflexive*, *symmetric* and *transitive*. As a particular example, we saw that the relation

$$x R y \quad \text{if} \quad x - y \quad \text{is an integral multiple of } n \quad x, y, n \in \mathbb{Z}, n > 0$$

is an equivalence relation. For purposes of the present discussion, let us denote the relation R by the symbol ' \equiv ' (read: 'is equivalent to'), satisfying the above properties for any elements x, y and z in S , thus:

1. Reflexivity: $x \equiv x$
2. Symmetry: if $x \equiv y$ then $y \equiv x$.
3. Transitivity: if $x \equiv y$ and $y \equiv z$, then $x \equiv z$.

Examples of equivalence relations include the relation 'is equal to' on the real numbers, the relation 'is similar to' on binary trees (see Theorem 6.1), and so on.

8.6.1 The equivalence problem

We may now define the equivalence problem as follows:

DEFINITION 8.3. The equivalence problem

Given equivalence relations of the form $i \equiv j$ for any $i, j \in S$, determine whether k is equivalent to l , for any $k, l \in S$, on the basis of the given equivalence relations.

As an example, consider the set $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Here we look at the integers $1, 2, \dots, 9$ as *labels* of objects on which an equivalence relation is defined. Suppose we are given the relations $2 \equiv 5$, $3 \equiv 8$, $1 \equiv 4$, $3 \equiv 9$, $2 \equiv 1$, $3 \equiv 6$, and $2 \equiv 7$. On the basis of these given equivalence relations, is $1 \equiv 7$? Is $5 \equiv 8$?

Scanning the given relations we find that $2 \equiv 1$, which implies that $1 \equiv 2$ (by symmetry). Further we find that $2 \equiv 7$ which implies that $1 \equiv 7$ (by transitivity). By a similar process we find that $5 \not\equiv 8$. We obtained the answers quite easily because the set S is small and the given equivalence relations are few. Now imagine that you are given, say, 200 relations on a set of, say, 500 objects. Clearly, going through such input relations, possibly repeatedly, to determine whether some object k in the set is equivalent or not to some other object l becomes an unwieldy and unreliable process. We need a more systematic method; such an algorithm is based on the following theorem.

THEOREM 8.1. An equivalence relation partitions its set into disjoint classes, called **equivalence classes**, such that two elements are equivalent if and only if they belong to the same equivalence class.

The solution to the equivalence problem lies in finding the equivalence classes generated by the given equivalence relations, and determining whether the objects k and l belong to the same equivalence class or not.

Consider again the sample problem given above. Initially, each object is in a class by itself:

$\{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\}$

Then, given the relation $2 \equiv 5$, we put 2 and 5 in the same class, thus:

$\{1\} \{2, 5\} \{3\} \{4\} \{6\} \{7\} \{8\} \{9\}$

Continuing, we process the relations $3 \equiv 8$, $1 \equiv 4$ and $3 \equiv 9$ to obtain the classes

$\{1, 4\} \{2, 5\} \{6\} \{7\} \{3, 8, 9\}$

Applying the relations $2 \equiv 1$, $3 \equiv 6$ and $2 \equiv 7$, we finally obtain the equivalence classes

$\{1, 2, 4, 5, 7\} \{3, 6, 8, 9\}$

Having found the equivalence classes into which the given equivalence relations partitioned the set S , we can readily answer the questions ‘Is $1 \equiv 7$?’ and ‘Is $5 \equiv 8$?’ by simply determining whether the objects belong to the same class or not. Thus $1 \equiv 7$ and $5 \not\equiv 8$.

Using the notion of equivalence classes it should not take too much effort, mentally or otherwise, to solve a larger problem with, say, 200 input equivalence relations on a set of, say, 500 objects; it will just be somewhat more tedious. The more interesting and challenging task is to carry out on a computer what we could readily do ‘by hand’ using pencil and paper. To this end, we need to answer the following questions:

- Q1. How do we represent equivalence classes in the memory of a computer?
- Q2. How do we determine whether two given objects belong to the same equivalence class or not?
- Q3. How do we merge equivalence classes?

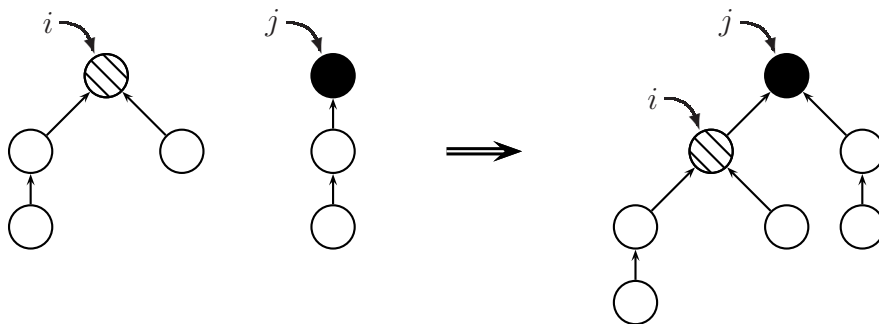
An elegant algorithm to solve the equivalence problem on a computer was put forth by M. J. Fischer and B. A. Galler in 1964. The underlying ADT is a *forest of oriented trees* with two basic operations: *find* and *union*. Following Fischer and Galler, these are the answers to the questions we just posed:

- A1. We will use oriented trees in a forest to represent equivalence classes. Each equivalence class is an oriented tree; the set of equivalence classes is the forest. Each equivalence class will have a ‘distinguished member’ which identifies the class; this is the root of the tree which represents the class. Internally, in computer memory, we will represent the forest using a single array of size n , called *FATHER*, where n is the number of objects in the set.

230 SESSION 8. Trees and forests

- A2. For each object we find the root of the tree which contains the object (the *find* operation). If the two objects have a common root, they are equivalent; otherwise, they are not.
- A3. We combine the trees representing the equivalence classes by making the root of one tree the father of the root of the other tree (the *union* operation).

Suppose we are given the equivalence relation $i \equiv j$ where i and j are distinct roots. Arbitrarily (at least for now), we will make j the father of i , as shown below. In terms of the *FATHER* array, we set $FATHER(i) \leftarrow j$.



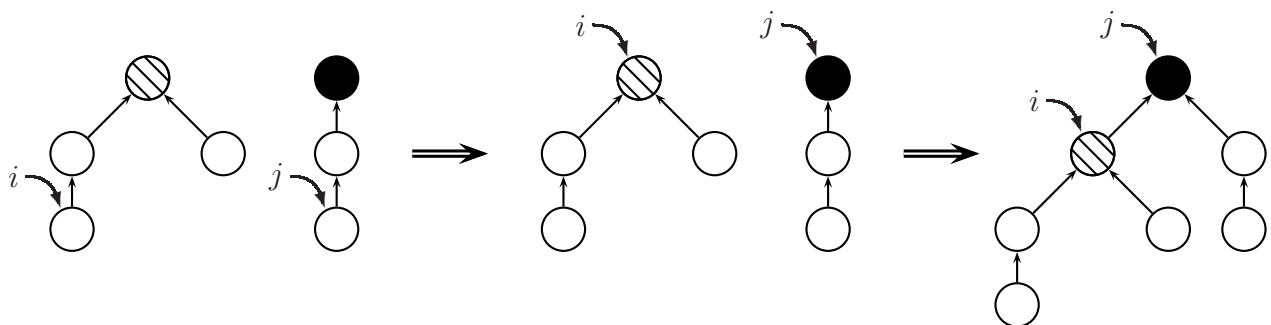
If i and/or j are not roots, then we first ‘climb up’ the trees to the roots. In terms of the *FATHER* array, climbing up the tree to find the root is easily accomplished, as shown in the following segment of EASY code.

```

while  $FATHER(i) > 0$  do
     $i \leftarrow FATHER(i)$ 
endwhile
while  $FATHER(j) > 0$  do
     $j \leftarrow FATHER(j)$ 
endwhile

```

Upon exit from the **while** loops, i points to a root and j points to a root. If the roots are distinct, we set $FATHER(i) \leftarrow j$.



Let us simulate the procedure just described for the equivalence problem given above. Study this example carefully.

	Forest of oriented trees	The <i>FATHER</i> array																		
		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9												
0	0	0	0	0	0	0	0	0												
$2 \equiv 5$		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>0</td><td>5</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	0	5	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9												
0	5	0	0	0	0	0	0	0												
$3 \equiv 8$ $1 \equiv 4$		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9												
4	5	8	0	0	0	0	0	0												
$3 \equiv 9$		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>0</td><td>0</td><td>0</td><td>0</td><td>9</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	0	0	0	0	9	0
1	2	3	4	5	6	7	8	9												
4	5	8	0	0	0	0	9	0												
$2 \equiv 1$		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>0</td><td>4</td><td>0</td><td>0</td><td>9</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	0	4	0	0	9	0
1	2	3	4	5	6	7	8	9												
4	5	8	0	4	0	0	9	0												
$3 \equiv 6$		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>0</td><td>4</td><td>0</td><td>0</td><td>9</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	0	4	0	0	9	6
1	2	3	4	5	6	7	8	9												
4	5	8	0	4	0	0	9	6												
$2 \equiv 7$		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>7</td><td>4</td><td>0</td><td>0</td><td>9</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	7	4	0	0	9	6
1	2	3	4	5	6	7	8	9												
4	5	8	7	4	0	0	9	6												

Figure 8.17 Solving the equivalence problem (initial version)

232 SESSION 8. Trees and forests

The EASY procedures EQUIVALENCE and TEST implement the algorithm presented above to solve the equivalence problem. Procedure EQUIVALENCE generates the equivalence classes on the basis of given equivalence relations on the elements of a set S and stores these in the *FATHER* array. If subsequently we want to determine whether two elements, say k and l in S are equivalent or not, we invoke procedure TEST with k and l as arguments. In these, and in all subsequent related procedures, the *FATHER* array is assumed to be global.

```
1  procedure EQUIVALENCE
▷ Given a set of objects labeled  $1, 2, \dots, n$ , the procedure finds the equivalence
▷ classes generated by input integer pairs  $i, j$  signifying  $i \equiv j$ . The pair  $0, 0$ 
▷ terminates the input. The equivalence classes are stored in the FATHER array.
2    FATHER  $\leftarrow 0$     ▷ Initialize FATHER array
3    input  $i, j$     ▷  $i \equiv j$ 
4    while  $i \neq 0$  do
5        while FATHER( $i$ )  $> 0$  do
6             $i \leftarrow$  FATHER( $i$ )
7        endwhile
8        while FATHER( $j$ )  $> 0$  do
9             $j \leftarrow$  FATHER( $j$ )
10       endwhile
11       if  $i \neq j$  then FATHER( $i$ )  $\leftarrow j$ 
12       input  $i, j$ 
13   endwhile
14   end EQUIVALENCE
```

Procedure 8.4 Generating equivalence classes (initial version)

```
1  procedure TEST( $k, l$ )
▷ Procedure returns true if  $k \equiv l$ ; else, returns false.
2    while FATHER( $k$ )  $> 0$  do
3         $k \leftarrow$  FATHER( $k$ )
4    endwhile
5    while FATHER( $l$ )  $> 0$  do
6         $l \leftarrow$  FATHER( $l$ )
7    endwhile
8    if  $k = l$  then return(true)
9        else return(false)
10   end TEST
```

Procedure 8.5 Determining whether two objects are equivalent (initial version)

8.6.2 Degeneracy and the weighting rule for *union*

To find the time complexity of procedure EQUIVALENCE we determine the number of times lines 6 and 9 are executed for an $O(n)$ sequence of input equivalence relations $i \equiv j$ on some set, say $S = \{1, 2, 3, \dots, n\}$. To this end, consider the input relations $1 \equiv 2$, $1 \equiv 3$, $1 \equiv 4$, \dots , $1 \equiv n$. Invoking procedure EQUIVALENCE on this input generates the equivalence classes depicted by the trees shown below, culminating in a single class represented by a tree of height $n - 1$. For this particular input, line 9 is never executed, but line 6 is executed $0 + 1 + 2 + \dots + n - 2 = (n - 1)(n - 2)/2$ times. The time complexity of Procedure 8.4 is therefore $O(n^2)$.

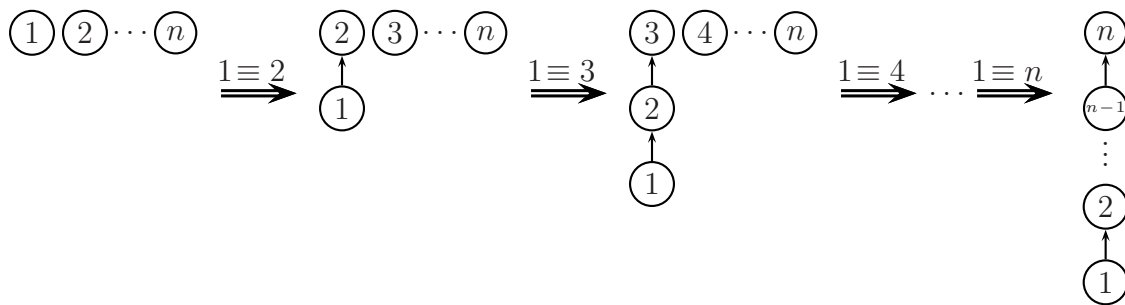
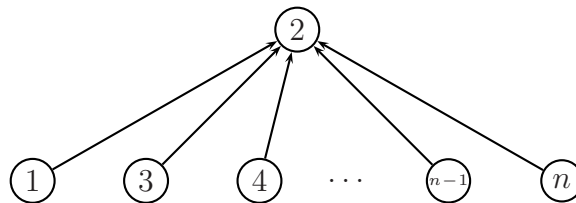


Figure 8.18 Worst case tree on n nodes ($h = n - 1$)

The reason why we obtain these degenerate trees for this particular input lies simply in always choosing the second root to be the father when two trees are merged. If, instead, we choose the root with the greater number of descendants to be the father, then for the same input $1 \equiv 2$, $1 \equiv 3$, $1 \equiv 4$, \dots , $1 \equiv n$ we will obtain the tree shown below, as you may easily verify.



Note that from any node in the tree, it now takes only one step to get to the root. The input relations used in this example have been chosen to produce the worst and best cases possible. Nonetheless, it should be clear that given some random input of equivalence relations, applying the procedure just described will yield shallower trees in the forest. We summarize this procedure in the form of a rule:

The weighting rule for the union operation

Let node i and node j be roots. If the number of nodes in the tree rooted at node i is greater than the number of nodes in the tree rooted at node j , then make node i the father of node j ; else, make node j the father of node i .

To implement the weighting rule for union, it is necessary to keep a count of the number of nodes of each tree in the forest. Since in the *FATHER* array a root node, say node i , has an entry of zero, then we can store instead the number of nodes in the tree rooted at

node i . To distinguish between counts and labels in *FATHER*, we affix a minus sign to counts. Following these conventions we initialize the *FATHER* array thus

$$FATHER(i) = -1 \quad 1 \leq i \leq n$$

since initially each tree in the forest is a root. Subsequently, $FATHER(i) = -m$ means node i is a root and that there are m nodes in the tree rooted at node i (including i itself); $FATHER(j) = p$ means node j is not a root and that its father is node p .

The EASY procedure UNION implements the weighting rule for the union operation. The arguments i and j are pointers (cursors) to the roots of the two trees to be merged.

▷ *Merges two trees with roots i and j using the weighting rule for union.*

```

1  procedure UNION( $i, j$ )
2     $count \leftarrow FATHER(i) + FATHER(j)$ 
3    if  $|FATHER(i)| > |FATHER(j)|$  then [  $FATHER(j) \leftarrow i$ ;  $FATHER(i) \leftarrow count$  ]
4                                     else [  $FATHER(i) \leftarrow j$ ;  $FATHER(j) \leftarrow count$  ]
5    end UNION

```

Procedure 8.6 Implementing the weighting rule for union

Previously, we found that if the weighting rule for union is not applied, the worst-case tree on n nodes has height $h = n - 1$, as shown in Figure 8.18. What is the height of the worst-case tree on n nodes if the weighting rule for union is applied each time that two trees are merged? The answer can be found by induction on h ; we can also arrive at the answer by construction. Figure 8.19 shows worst-case trees for $n = 1, 2, 4, 8$ and 16 with $h = 0, 1, 2, 3$ and 4 , respectively. We obtain the worst-case tree on n nodes, where n is a power of 2, by simply merging two worst-case trees on $n/2$ nodes. The height of the worst-case tree on *any* $n \geq 1$ nodes is therefore $\lceil \log_2 n \rceil$.

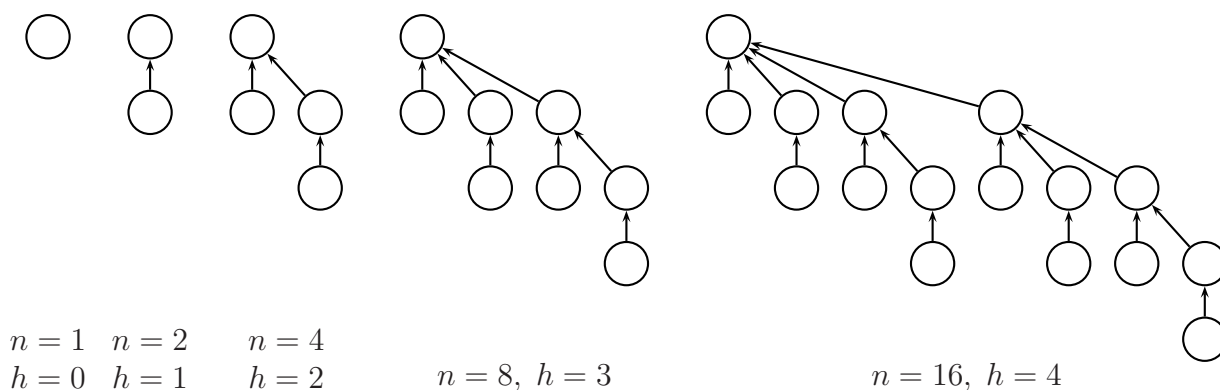
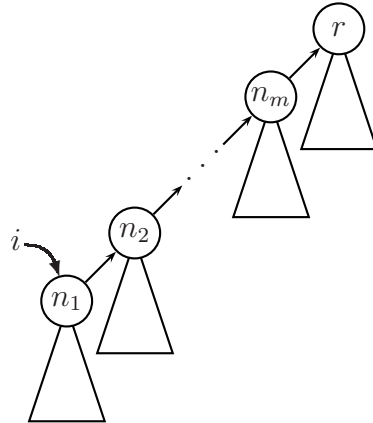


Figure 8.19 Worst-case trees on n nodes when the weighting rule for union is applied

8.6.3 Path compression: the collapsing rule for *find*

Consider the equivalence class depicted by the tree shown below. To process a relation involving node n_1 , e.g., $i \equiv j$, we have to climb up the tree to find the root r ; this takes



m steps, i.e., m executions of $i \leftarrow \text{FATHER}(i)$. If now the tree rooted at r is merged with another tree, then to process any subsequent relation involving node n_1 will take *at least* m steps to find the root of the tree which contains n_1 . To avoid such subsequent repetitive climbs we can, after finding the root r , make r the father of every node in the path from n_1 to r . This will of course involve additional computations at this point; our expectation is that this extra effort will result in reduced overall computing time. It does, as we will see shortly. We summarize the procedure in the form of a rule:

The collapsing rule for the find operation

Let n_1, n_2, \dots, n_m be nodes in the path from node n_1 to the root, say node r . Then make node r the father of nodes $n_p, 1 \leq p < m$.

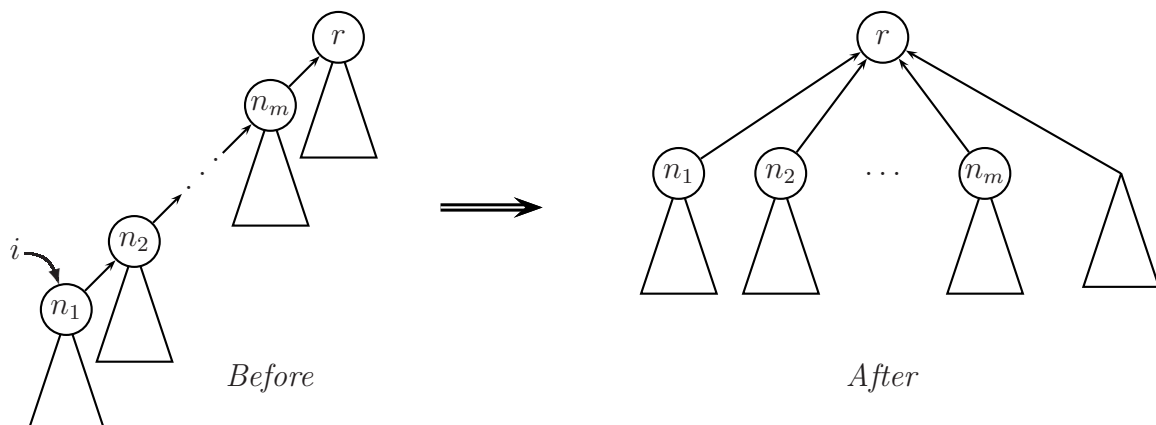


Figure 8.20 Path compression

The EASY procedure FIND implements the collapsing rule for the find operation.

▷ Finds the root of the tree containing node i and collapses the path from node i to the root

```

1  procedure FIND( $i$ )
▷ Find root
2     $r \leftarrow i$ 
3    while FATHER( $r$ ) > 0 do
4       $r \leftarrow$  FATHER( $r$ )
5    endwhile
▷ Compress path from node  $i$  to the root  $r$ 
6     $j \leftarrow i$ 
7    while  $j \neq r$  do
8       $k \leftarrow$  FATHER( $j$ )
9      FATHER( $j$ )  $\leftarrow r$ 
10    $j \leftarrow k$ 
11  endwhile
12  return( $r$ )
13  end FIND

```

Procedure 8.7 Implementing the collapsing rule for find

8.6.4 Analysis of the union and find operations

The union operation, in which we make the root of one tree the father of the root of the other tree, clearly takes $O(1)$ time. On the other hand, the cost of a find operation on node i is proportional to the length of the path from node i to its root. If the weighting rule for union is not applied, an $O(n)$ sequence of find-union operations takes, in the worst case, $O(n^2)$ time (see Figure 8.18). If, however, the weighting rule for union is applied, then an $O(n)$ sequence of find-union operations takes $O(n \log n)$ time only (see Figure 8.19). Applying the collapsing rule for the find operation reduces further the time complexity.

To appreciate the effect of path compression, we define a function $F(k)$ such that

	k	$F(k)$
$F(0) = 1$	0	1
$F(k) = 2^{F(k-1)}$	1	2
	2	4
	3	16
	4	65536
	5	2^{65536}

The function $F(k)$ grows extremely fast, as we can see from the table. Now we define the inverse function $G(n)$ such that

$$G(n) = \text{smallest integer } k \text{ such that } F(k) \geq n$$

For example, we have $G(100) = G(10000) = G(60000) = 4$; in fact, $G(n) \leq 5$ for all $n \leq 2^{65536}$. $G(n)$ is obviously an agonizingly slowly increasing function.

It can be shown that with path compression, an $O(n)$ sequence of find-union operations takes $O(n G(n))$ time. The proof is beyond the scope of the material in this book; the interested student is referred to AHO[1974], pp. 133–139. Although theoretically, $O(n G(n))$ is not linear in n , it practically is.

8.6.5 Final solution to the equivalence problem

The solution to the equivalence problem which utilizes a forest of oriented trees as the underlying data structure and as implemented by procedures EQUIVALENCE and TEST given above, is undoubtedly an elegant one. Incorporating the weighting rule for the union operation, as implemented by procedure UNION, and the collapsing rule for the find operation, as implemented by procedure FIND, makes it also unquestionably efficient.

The EASY procedures EQUIVALENCE and TEST given below (we use the same names since the earlier versions may now be discarded) implement this final solution to the equivalence problem.

```

1  procedure EQUIVALENCE
  ▷ Given a set of objects labeled 1, 2, ..., n, the procedure finds the equivalence
  ▷ classes generated by input integer pairs i, j signifying  $i \equiv j$ . The pair 0, 0
  ▷ terminates the input. The equivalence classes are stored in the FATHER array.
  ▷ The procedure uses the collapsing rule for the find operation and the weighting
  ▷ rule for the union operation.
2  FATHER ← -1      ▷ Initialize FATHER array
3  input i, j        ▷  $i \equiv j$ 
4  while i ≠ 0 do
5    i ← FIND(i)
6    j ← FIND(j)
7    if i ≠ j then call UNION(i, j)
8    input i, j
9  endwhile
10 end EQUIVALENCE

```

Procedure 8.8 Generating equivalence classes (final version)

```

1  procedure TEST(k, l)
  ▷ Procedure returns true if  $k \equiv l$ ; else, returns false.
2  k ← FIND(k)
3  l ← FIND(l)
4  if k = l then return(true)
5    else return(false)
6  end TEST

```

Procedure 8.9 Determining whether two objects are equivalent (final version)

Applying this final version of procedure EQUIVALENCE to the problem in Figure 8.17 yields the following forest and corresponding FATHER array. It is instructive to compare the two solutions depicted in Figures 8.17 and 8.21.

	Forest of oriented trees	The <i>FATHER</i> array																		
		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	9	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9												
-1	-1	-1	-1	-1	-1	-1	-1	-1												
2 ≡ 5		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>-1</td><td>5</td><td>-1</td><td>-1</td><td>-2</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	9	-1	5	-1	-1	-2	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9												
-1	5	-1	-1	-2	-1	-1	-1	-1												
3 ≡ 8 1 ≡ 4		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>-2</td><td>-2</td><td>-1</td><td>-1</td><td>-2</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	-2	-2	-1	-1	-2	-1
1	2	3	4	5	6	7	8	9												
4	5	8	-2	-2	-1	-1	-2	-1												
3 ≡ 9		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>-2</td><td>-2</td><td>-1</td><td>-1</td><td>-3</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	-2	-2	-1	-1	-3	8
1	2	3	4	5	6	7	8	9												
4	5	8	-2	-2	-1	-1	-3	8												
2 ≡ 1		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>-4</td><td>4</td><td>-1</td><td>-1</td><td>-3</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	-4	4	-1	-1	-3	8
1	2	3	4	5	6	7	8	9												
4	5	8	-4	4	-1	-1	-3	8												
3 ≡ 6		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>5</td><td>8</td><td>-4</td><td>4</td><td>8</td><td>-1</td><td>-4</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	9	4	5	8	-4	4	8	-1	-4	8
1	2	3	4	5	6	7	8	9												
4	5	8	-4	4	8	-1	-4	8												
2 ≡ 7		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>4</td><td>4</td><td>8</td><td>-5</td><td>4</td><td>8</td><td>4</td><td>-4</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	9	4	4	8	-5	4	8	4	-4	8
1	2	3	4	5	6	7	8	9												
4	4	8	-5	4	8	4	-4	8												

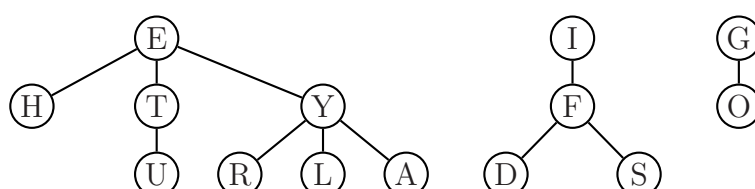
Figure 8.21 Solving the equivalence problem (final version)

Summary

- Although they share certain concepts (degree, level, height, etc) and use the same sets of terminology (botanical, spatial and familial), binary trees and trees are fundamentally different in many ways. Three important species of trees are: ordered, oriented and free. Of these three, the closest relative of a binary tree is an ordered tree.
- A forest is a set of zero or more disjoint trees. Corresponding to a forest of ordered trees is a unique binary tree, and vice versa; this relationship is called *natural correspondence*.
- Two traversal sequences are defined for ordered forests: preorder, in which a node appears immediately before its descendants; and postorder, in which a node appears immediately after its descendants.
- Representing ordered forests in computer memory using the linked design results in much wasted space as most of the link fields are merely occupied by Λ . As such, various sequential representation schemes have been devised for ordered forests. Some of these utilize links and tags, while others utilize arithmetic tree information such as degree and weight.
- The ADT used to solve the equivalence problem in this session, which consisted of a forest of oriented trees to represent sets along with the two basic set operations of find and union, is often referred to as the **disjoint set ADT**. The implementation of this ADT, in which the forest of oriented trees is stored in a single sequential array, coupled with the application of two heuristics, viz., the *weighting rule* for the union operation and the *collapsing rule* for the find operation, yields a solution to the equivalence problem whose space complexity is linear, and whose time complexity is practically linear, in the size of the problem (which is as good as it can get).

Exercises

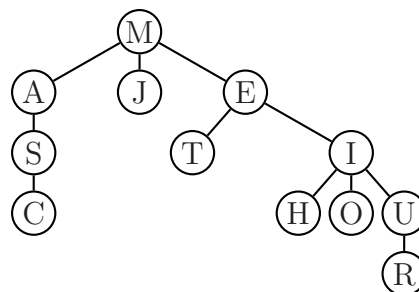
1. Draw all the
 - (a) ordered trees (14 in all)
 - (b) oriented trees (9 in all)
 - (c) free trees (3 in all)
 on five unlabeled nodes.
2. Construct the binary tree which corresponds to the ordered forest shown below.



3. How many distinct ordered trees are there on n unlabeled nodes? [Hint: Use natural correspondence.]

4. List the nodes of the tree shown in

- (a) preorder
(b) postorder
(c) level order



5. List the nodes of the ordered forest in Item 2 in: (a) preorder (b) postorder
(c) level order

6. Construct the expression trees for the following expressions.

(a) $\sqrt{\sin(x+y) \cos(x-y) + \cos(x+y) \sin(x-y)}$

(b) $-2 \ln |x-1| + 5 \ln(x^2+1)$

(c) $-\frac{x}{y} \left(\frac{2x\sqrt{xy}+1}{2y\sqrt{xy}-1} \right)$

7. Show the

- (a) preorder (b) family order (c) level order

sequential representation of the ordered forest in Item 2.

8. Show the representation of the ordered forest in Item 2 in

- (a) preorder with degrees (c) postorder with degrees (e) level order with degrees
(b) preorder with weights (d) postorder with weights (f) level order with weights

9. Construct the ordered forest whose representation in postorder with weights is

DATA: A F R I E N D L Y G H O S T
WEIGHT: 0 1 0 0 2 0 6 0 0 0 2 0 1 6

10. An ordered forest is represented in level order with degrees. Formulate a simple rule for determining the number of trees in the forest.

11. Construct the ordered forest whose representation in level order with degrees is

DATA: A W O N D E R F U L S I G H T
DEGREE: 3 1 2 1 0 1 1 0 1 0 2 0 0 0 0

12. Write an EASY procedure which accepts as input the family order sequential representation of an ordered forest F and generates the linked representation of the corresponding binary tree $B(F)$.

13. Write an EASY procedure which accepts as input the level order sequential representation of an ordered forest F and generates the linked representation of the corresponding binary tree $B(F)$.

14. Write an EASY procedure which accepts as input an ordered forest represented in preorder sequence with degrees and generates the corresponding representation in preorder sequence with weights.
15. Using a language of your choice, transcribe procedure CONVERT (Procedure 8.1) into a running program.
16. Using a language of your choice, transcribe procedure ANCESTORS (Procedure 8.2) into a running program.
17. Using a language of your choice, transcribe procedure DEGREE (Procedure 8.3) into a running program.
18. Write a recursive EASY version of procedure FIND (Procedure 8.7).
19. Show the forest and corresponding *FATHER* array which represents the equivalence classes on the set $S = (1, 2, 3, 4, 5, 6, 7, 8, 9)$ generated by the equivalence relations $9 \equiv 3$, $7 \equiv 5$, $9 \equiv 7$, $8 \equiv 4$, $9 \equiv 8$, $1 \equiv 6$ and $1 \equiv 2$. Do *not* use the weighting rule for the union operation and the collapsing rule for the find operation.
20. Show the forest and corresponding *FATHER* array which represents the equivalence classes on the set $S = (1, 2, 3, 4, 5, 6, 7, 8, 9)$ generated by the equivalence relations $9 \equiv 3$, $7 \equiv 5$, $9 \equiv 7$, $8 \equiv 4$, $9 \equiv 8$, $1 \equiv 6$ and $1 \equiv 2$. Use the weighting rule for the union operation and the collapsing rule for the find operation.
21. Using a language of your choice, transcribe procedures UNION, FIND, EQUIVALENCE and TEST (Procedures 8.6 thru 8.9) into a running program.

Bibliographic Notes

KNUTH1[1997] and STANDISH[1980] are our primary sources for the material on the sequential representation for ordered trees and forests. KNUTH1[1997], p. 350 describes some variations on the theme of preorder sequential representation that lead to better space utilization.

The equivalence problem is discussed in part in KNUTH1[1997], pp. 353–355. Procedure EQUIVALENCE (initial version) is an implementation of Algorithm E (*Process equivalence relations*) given in this reference. The weighting rule for the union operation (union by size or union by rank) and the collapsing rule for the find operation (path compression) are discussed in HOROWITZ[1976], AHO[1974], CORMEN[2001] and WEISS[1997], among others. We have adopted the analysis of the UNION and FIND procedures as given in AHO[1974], pp. 129–135, which is for a sequence of $O(n)$ FIND-UNION operations. For a more general analysis and stronger results, see CORMEN[2001], pp. 505–517 or WEISS[1997], pp. 269–277.

NOTES

SESSION 9

Graphs

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define, and give examples of, an undirected graph.
2. Define, and give examples of, a directed graph.
3. Explain important properties of, and concepts related to, undirected and directed graphs which are essential in understanding graph algorithms.
4. Represent any given graph using adjacency matrices or adjacency lists and explain the trade-offs involved in choosing one representation over another.
5. Explain DFS and BFS traversal for graphs and apply the methods to any given graph.
6. Explain how DFS and BFS are used to solve elementary graph problems such as determining whether an undirected or directed graph is acyclic, finding strongly connected components in a directed graph, finding articulation points in an undirected graph, and so on.

READINGS CORMEN[2001], pp. 525–560, 1080–1084; STANDISH[1994], pp. 405–423; AHO[1983], pp. 222–226; WEISS[1997], pp. 322–327.

DISCUSSION

In this and the next session, we will consider an extremely useful mathematical structure called a **graph**. Graphs are used to model a variety of problems across many disciplines: in engineering, physics, chemistry, operations research, economics, computer science, and so on. A graph may be used to represent an electrical circuit, a program flowchart, a communications network, airline routes, a molecular structure, a street intersection, and the like.

Informally, we can think of a graph as a collection of nodes (called *vertices*) connected by line segments (called *edges*). A vertex may represent a computer in a computer network, a city in an airline map, a task in an activity network, a street at an intersection,

and so on. An edge indicates some kind of ‘connectedness’ between two vertices, for instance a link between two computers, a route between two cities, a precedence relationship between two tasks, a permissible turn at a street intersection, and so on.

Suppose one computer breaks down in a computer network; will the rest be able to still communicate with each other? This is the problem of finding ‘articulation points’ and ‘biconnected components’ in a graph. Or, how do we find a way to travel from a starting city to a destination city through a shortest route? This is an instance of the ‘shortest path problem’ in graphs. The exam-scheduling problem which we encountered in Session 1, in which we wanted to find a conflict-free schedule with the minimum number of time slots, reduces to the so-called ‘graph coloring problem’. These are examples from a vast array of real-life problems which are modeled by graphs. Many elegant algorithms which run in polynomial time have been devised to solve some of these problems, and we will examine a number of these algorithms in this and the next session. A large number of these problems, however, have proved to be intractable, taking at least exponential time to solve.

9.1 Some pertinent definitions and related concepts

There is a great deal of terminology associated with graphs. A vital key to understanding algorithms on graphs is a clear grasp of this terminology.

1. A **graph** $G = (V, E)$ consists of a finite, nonempty set of **vertices**, V , and a finite, possibly empty set of **edges**, E . Shown below are five different graphs. For graph G_3 , the vertex set is $V = \{1, 2, 3, 4\}$ and the edge set is $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$. For graph G_5 the vertex set is $V = \{1, 2, 3, 4, 5\}$ and the edge set is $E = \{(1, 2), (1, 3), (3, 2), (4, 5)\}$.

In the rest of this session, we will use n to denote the number of vertices and e to denote the number of edges in G , i.e., $n = |V|$ and $e = |E|$.

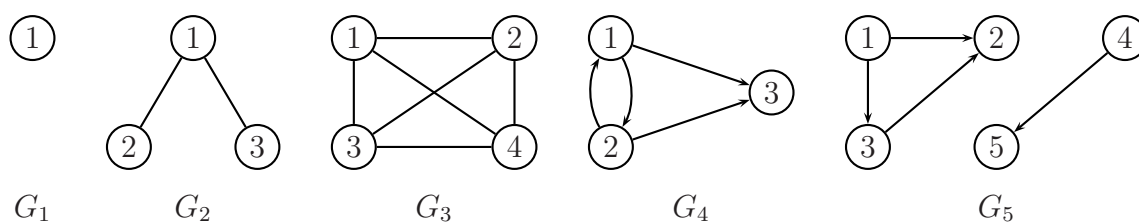


Figure 9.1 Five graphs

A **subgraph** of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. G' is an **induced subgraph** of G if E' consists of *all* the edges in E whose endpoints are in V' ; in this case, we say that G' is *induced* by V' .

2. An **undirected graph** $G = (V, E)$ is a graph in which edges are represented by an *unordered* pair (i, j) for any vertices i and j in V ; thus (i, j) and (j, i) are the *same* edge. We require that the vertices i and j be distinct, i.e., there can be no edge from a vertex to itself. Graphs G_2 and G_3 in Figure 9.1 are undirected graphs.

Let (i, j) be an edge in an undirected graph. We say that edge (i, j) is *incident on* vertices i and j . Further, we say that vertex i is *adjacent to* vertex j and vertex j is adjacent to vertex i , i.e., vertices i and j are adjacent to each other (the adjacency relation is symmetric).

3. A **directed graph** or **digraph** $G = (V, E)$ is a graph in which edges are represented by an *ordered* pair (i, j) for any vertices i and j in V , where i is the *tail* and j is the *head* of the edge. The edges (i, j) and (j, i) are two *distinct* edges. We allow for the vertices i and j to be the same, i.e., there can be an edge from a vertex to itself; such an edge is called a **self-loop**. However, in this session we will consider only directed graphs in which there are *no* self-loops; such graphs are called **simple**. Graphs G_4 and G_5 in Figure 9.1 are simple directed graphs.

Let (i, j) be an edge in a directed graph. We say that edge (i, j) is *incident from* or *leaves* vertex i and is *incident to* or *enters* vertex j . Further, we say that vertex j is *adjacent to* vertex i ; however, unless there is also an edge (j, i) , vertex i is *not* adjacent to vertex j (the adjacency relation is asymmetric). In graph G_4 of Figure 9.1, vertex 3 is adjacent to vertex 2, but vertex 2 is not adjacent to vertex 3.

4. The *maximum* number of edges in an undirected graph on n vertices is $n(n - 1)/2$, since each of the n vertices has at most $n - 1$ edges incident on it (self-loops are not allowed) and the edge (i, j) is also the edge (j, i) . Assuming that there are no self-loops, the maximum number of edges in a directed graph on n vertices is $n(n - 1)$, since each of the n vertices has at most $n - 1$ edges incident from it and the edges (i, j) and (j, i) are distinct. A graph with the full complement of edges is said to be **complete**; in a complete graph every pair of vertices are adjacent. Graph G_3 of Figure 9.1 is a complete undirected graph.

A graph in which relatively few of the possible edges are present is said to be **sparse**; a graph in which relatively few of the possible edges are absent is said to be **dense**. In terms of the parameters e and n , we say that a graph is sparse if $e \ll n^2$ and it is dense if $e \approx n^2$.

5. The **degree** of a vertex in an undirected graph is the number of edges incident on it. In a directed graph, the **out-degree** of a vertex is the number of edges incident from, or which leave, it, and the **in-degree** of a vertex is the number of edges incident to, or which enter, it; the degree of a vertex is the sum of its in-degree and out-degree. All the vertices of graph G_3 in Figure 9.1 have degree 3. Vertex 1 of graph G_4 has in-degree 1, out-degree 2 and degree 3.

Let $G = (V, E)$ be a graph with $e = |E|$ edges; the following formulas are easily verified.

$$\sum_{i \in V} \text{degree}(i) = 2e \quad (9.1)$$

Proof: Each edge contributes 1 unit to the degree of one of its endpoints, and one unit to the degree of its other endpoint. Hence each edge contributes two units to the total degree count.

$$\sum_{i \in V} \text{in-degree}(i) = \sum_{i \in V} \text{out-degree}(i) = e \quad (9.2)$$

Proof: Each directed edge contributes one unit to the out-degree of its tail and one unit to the in-degree of its head.

It follows from Eq.(9.1) that the number of vertices of odd degree in G is even.

6. A **path** from vertex i to vertex k in a graph $G = (V, E)$ is a sequence of vertices $v_0, v_1, v_2, \dots, v_{m-1}, v_m$, where $v_0 = i$ and $v_m = k$, such that $(v_0, v_1), (v_1, v_2), \dots, (v_{m-1}, v_m)$ are edges in E . The *length of a path* is the number of edges in it. A single vertex i is considered to be a path of length zero from i to itself. A path in which all vertices, except possibly the start vertex and the end vertex, are distinct is called a **simple path**.

If there is a path from vertex i to vertex k in $G = (V, E)$, we say that k is *reachable* from i . If G is undirected, this implies that i is also reachable from k . If G is directed, i is not necessarily reachable from k .

A **cycle** is a path with the same start and end vertex. A **simple cycle** is a simple path with the same start and end vertex. In a directed graph, a cycle has length at least 1 (contains at least one edge); a self-loop is a cycle of length 1. In an undirected graph, a cycle has length at least 3 (and must therefore contain at least three vertices).

In graph G_3 of Figure 9.1, $1 - 2 - 3 - 4$ is a simple path, $1 - 2 - 3 - 4 - 2$ is not a simple path, $1 - 2 - 3 - 4 - 1$ is a simple cycle, $1 - 2 - 3 - 4 - 2 - 1$ is not a simple cycle, and $1 - 2 - 1$ is *not* a cycle. In graph G_4 , $1 - 2 - 3$ is a simple path, $1 - 2 - 1 - 3$ is not a simple path, $1 - 3 - 2$ is *not* a path, and $1 - 2 - 1$ is a simple cycle.

A graph with no cycles is said to be **acyclic**. Graph G_5 in Figure 9.1 is acyclic.

7. Let $G = (V, E)$ be an undirected graph. Define the relation ' \rightarrow ' by

$$i \rightarrow k \quad \text{if } k \text{ is reachable from } i$$

for $i, k \in V$. Then ' \rightarrow ' is an equivalence relation which partitions V into equivalence classes such that two vertices are in the same class if a path connects the two. These equivalence classes are called the **connected components** of G . An undirected graph is said to be **connected** if it has only one connected component. Graph G_1 in Figure 9.2 has three connected components: $\{1, 2, 3, 4\}$, $\{5\}$ and $\{6, 7\}$. Thus, every vertex in $\{1, 2, 3, 4\}$ is reachable from every other vertex in $\{1, 2, 3, 4\}$, but not from any vertex in $\{5\}$ or in $\{6, 7\}$; vertices 6 and 7 in $\{6, 7\}$ reach each other but not 5. Graph G_2 in Figure 9.2 is a connected undirected graph; each vertex in G_2 is reachable from every other vertex in G_2 , as you may easily verify.

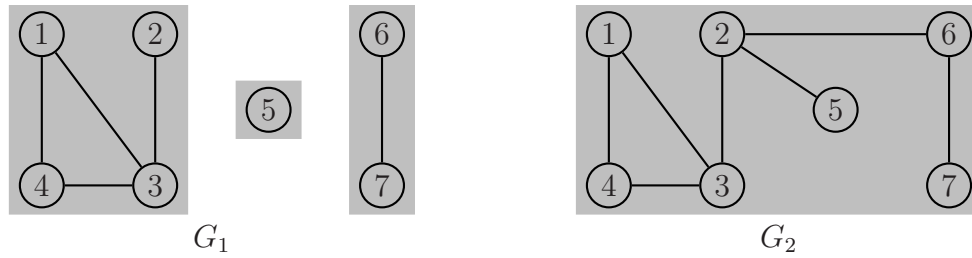


Figure 9.2 Connected components of an undirected graph

8. Let $G = (V, E)$ be a directed graph. Define the relation ' \rightleftharpoons ' by

$$i \rightleftharpoons k \quad \text{if } k \text{ is reachable from } i \text{ and } i \text{ is reachable from } k$$

for $i, k \in V$. Then ' \rightleftharpoons ' is an equivalence relation which partitions V into equivalence classes such that two vertices are in the same class if they are mutually reachable. These equivalence classes are called the **strongly connected components** of G . A directed graph is said to be **strongly connected** if it has only one strongly connected component. Graph G_1 in Figure 9.3 has three strongly connected components: $\{1, 2, 3, 4\}$, $\{5\}$ and $\{6, 7\}$. All pairs of vertices in $\{1, 2, 3, 4\}$ are mutually reachable; vertex 5 is reachable from each vertex in $\{1, 2, 3, 4\}$, but none of the vertices in $\{1, 2, 3, 4\}$ is reachable from 5. Vertices 6 and 7 in $\{6, 7\}$ are reachable each from the other; they are also reachable from 5, but 5 is not reachable from them. Graph G_2 in Figure 9.3 is a strongly connected directed graph; each vertex in G_2 is reachable from every other vertex in G_2 , as you may easily verify.

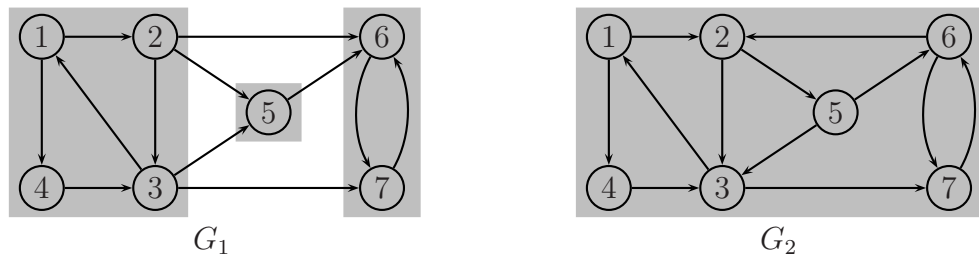


Figure 9.3 Strongly connected components of a directed graph

9. A vertex i of a connected, undirected graph $G = (V, E)$ is an **articulation point** if deleting vertex i and all edges incident on i disconnects G , i.e., G becomes two or more connected components. A connected undirected graph with no articulation points is said to be **biconnected**. Graph G_1 in Figure 9.4 has two articulation points, viz., vertices 2 and 5. Deleting vertex 2 disconnects G_1 into $G_{11} = (\{3\}, \{\})$ and $G_{12} = (\{1, 4, 5, 6, 7, 8\}, \{(1, 4), (1, 5), (4, 5), (5, 6), (5, 7), (5, 8), (6, 7), (6, 8)\})$. Deleting vertex 5 disconnects G_1 into $G_{11} = (\{1, 2, 3, 4\}, \{(1, 2), (1, 4), (2, 3), (2, 4)\})$ and $G_{12} = (\{6, 7, 8\}, \{(6, 7), (6, 8)\})$. Graph G_2 in Figure 9.4 is a biconnected graph; deleting any vertex in G_2 and all edges incident on it leaves the graph connected, as you may easily verify.

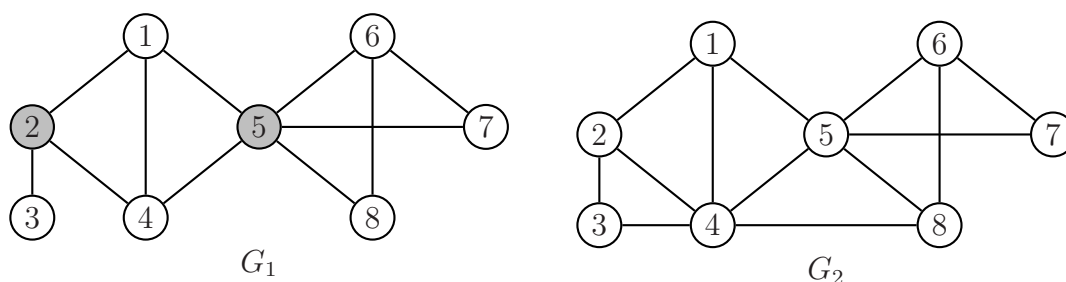


Figure 9.4 Articulation points of a connected undirected graph and biconnectivity

10. A **weighted graph**, also called a **network**, is a graph with *weights* or *costs* assigned to its edges. For instance, if the vertices of a graph represent cities, the cost of an edge connecting two vertices may represent the amount of the fare by bus, the length of the travel time by plane, and so on. In general, weights or costs are any real number; however, in this and the next session, we will assume that they are non-negative.

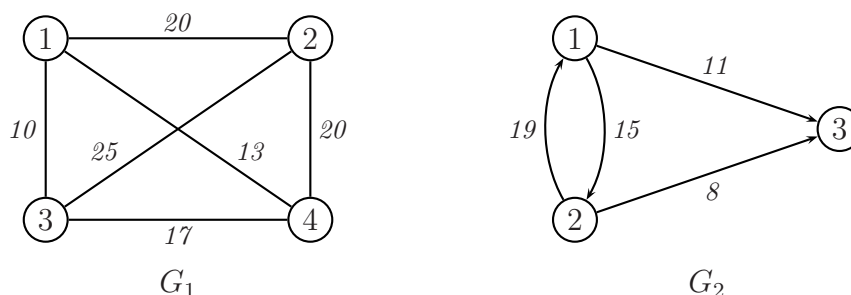


Figure 9.5 Two weighted graphs

11. A **spanning tree** is a tree which connects all the vertices of a graph. If the graph is weighted, the cost of a spanning tree is the sum of the weights or costs of the branches (edges) in the spanning tree. A **minimum cost spanning tree** is a spanning tree with minimum cost. For a given weighted graph, it is not necessarily unique. Figure 9.6 shows four of the 16 different spanning trees for graph G_1 in Figure 9.5. Trees T_3 and T_4 are minimum cost spanning trees.

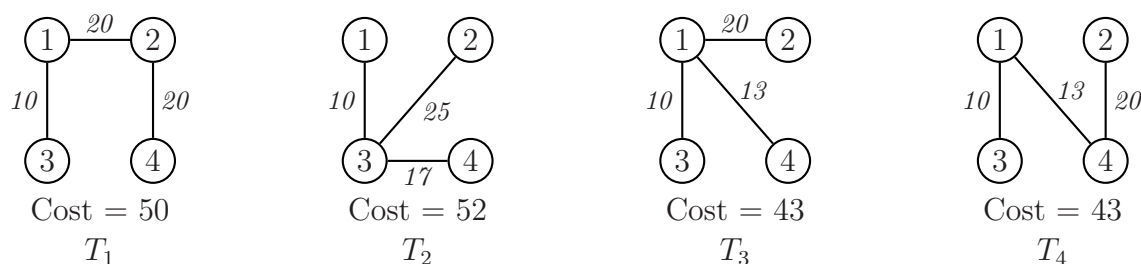


Figure 9.6 Four spanning trees for the undirected graph G_1 in Figure 9.5

9.2 Representation of graphs

There are two common ways of representing a graph $G = (V, E)$, viz., the **adjacency matrix representation** and the **adjacency lists representation** of G . As with other ADT's, the types of operations to be performed on the graph are a primary consideration in choosing which representation to use. Another important consideration is the number of edges relative to the number of vertices in the graph, i.e., whether G is sparse or dense.

9.2.1 Adjacency matrix representation of a graph

A simple and straightforward way of representing a graph $G = (V, E)$ on $n = |V|$ vertices, labeled $1, 2, \dots, n$, is by using an n by n matrix, say A , whose elements are defined by

$$A(i, j) = \begin{cases} 1 & \text{if edge } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Figure 9.7 shows the adjacency matrices A_1 and A_2 for an undirected graph G_1 and a directed graph G_2 , respectively. We take note of the following observations pertaining to

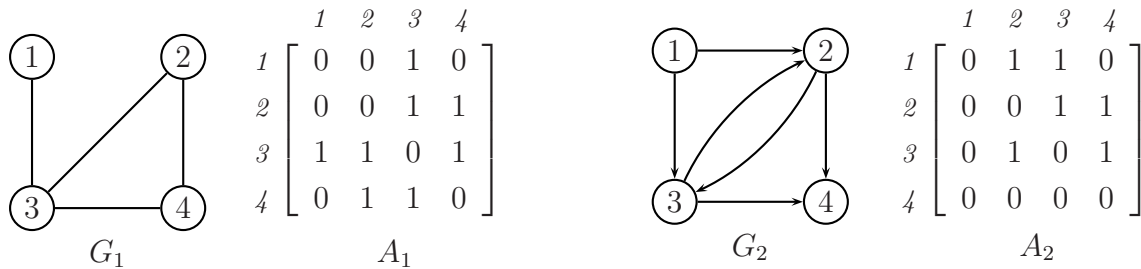


Figure 9.7 Adjacency matrix representation of a graph

the matrices A_1 and A_2 , or to an adjacency matrix A in general.

- The adjacency matrix A for a graph $G = (V, E)$ on n vertices requires $O(n^2)$ space, since space must be allocated for all possible edges whether present or not. Although one bit suffices to indicate a 0 or a 1, most programming languages do not have a facility for declaring an array of bits with the array elements accessed using the built-in array indexing mechanisms of the language; in such a case, we may use instead an array of logical variables where each occupies one byte.
- Given any two vertices $i, j \in V$, it takes $O(1)$ time to determine whether or not there is an edge (i, j) in E .
- The diagonal elements are all 0's since there are no self-loops (they are not allowed in undirected graphs and, although allowed in directed graphs, are not relevant to our present study).
- The adjacency matrix for an undirected graph is symmetric since (i, j) and (j, i) are one and the same edge. In some applications it may suffice to retain the upper (or lower) triangular part only (and leave the rest as 'don't cares').

- (e) The number of 1's in row i of the adjacency matrix of a directed graph is the out-degree of vertex i . The number of 1's in column j is the in-degree of vertex j .

A weighted graph is represented by its **cost adjacency matrix**, say C , with elements defined by

$$C(i, j) = \begin{cases} \text{cost of edge } (i, j) & \text{if edge } (i, j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

In some applications it may be more appropriate to use 0 in lieu of ∞ to indicate a nonexistent edge. Figure 9.8 shows the cost adjacency matrices C_1 and C_2 for a weighted undirected graph G_1 and a weighted directed graph G_2 , respectively. Note that C_1 is symmetric; in certain applications it may suffice to retain the upper (or lower) triangular part only (and leave the rest as ‘don’t cares’).

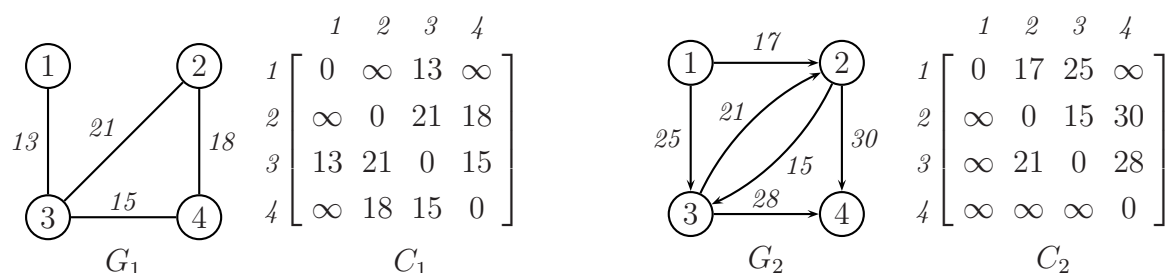


Figure 9.8 Cost adjacency matrix representation of a weighted graph

9.2.2 Adjacency lists representation of a graph

As we have pointed out earlier, the adjacency matrix representation of a graph $G = (V, E)$ on n vertices requires $O(n^2)$ space. Further, even simple operations such as inputting the matrix elements for G or determining the number of edges in G , also take $O(n^2)$ time. When the number of edges is much smaller than n^2 , a representation in which only those edges that are actually present in G are represented can lead to savings in both allocated space and processing time.

In the adjacency lists representation of a graph $G = (V, E)$ on n vertices, an array of size n , say $LIST$, is maintained such that for any vertex i in V , $LIST(i)$ points to the list of vertices adjacent to i . Each such adjacency list is maintained as a linked list, where each node in the list has node structure, say, $(VRTX, NEXT)$. If no vertices are adjacent to vertex i , $LIST(i)$ is set to null. Figure 9.9 shows the adjacency lists representation of an undirected graph G_1 and a directed graph G_2 , respectively. We take note of the following observations pertaining to these representations.

- (a) There are n adjacency lists for a graph $G = (V, E)$ on n vertices, one for each vertex in V . The adjacency list for vertex i consists of all vertices adjacent to vertex i .

- (b) An undirected edge (i, j) is represented by a node in the adjacency list for vertex i and another node in the adjacency list for vertex j . For instance, the shaded nodes in L_1 of Figure 9.9 represent the edge $(1, 3)$ in G_1 . The number of nodes comprising the adjacency lists of an undirected graph G is equal to twice the number of edges in E .
- (c) A directed edge (i, j) is represented by a node in the adjacency list for vertex i (and in this list only). For instance, the shaded node in L_2 of Figure 9.9 represents the edge $(2, 3)$ in G_2 . The number of nodes comprising the adjacency lists of a directed graph G is equal to the number of edges in E .
- (d) The adjacency lists representation of a graph G on n vertices and e edges requires $O(n + e)$ space.
- (e) Given any two vertices $i, j \in V$, it takes $O(n)$ time to determine whether or not there is an edge $(i, j) \in E$, since an adjacency list can be $O(n)$ long.

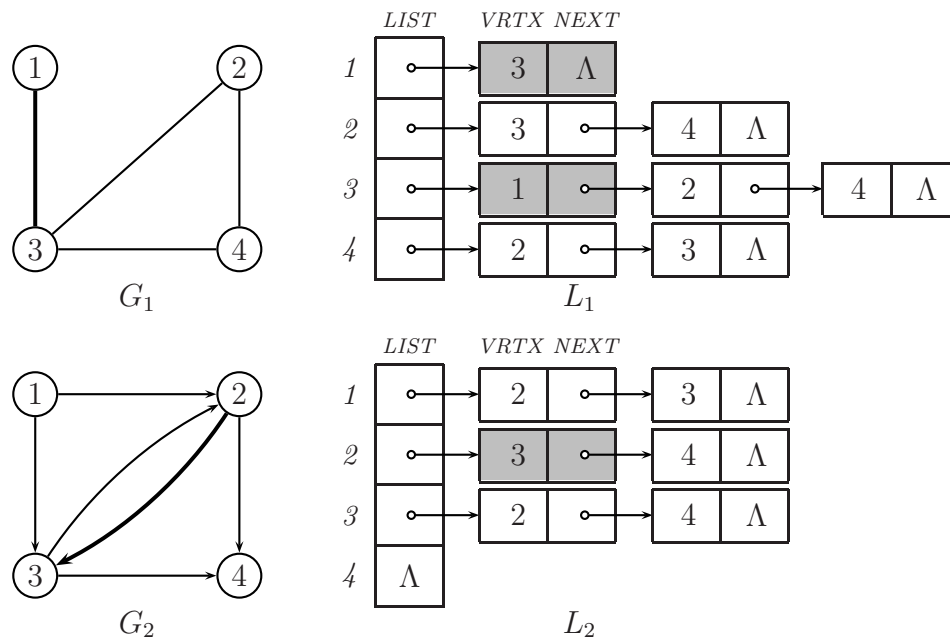


Figure 9.9 Adjacency lists representation of a graph

For weighted graphs, we could use the node structure $(VRTX, COST, NEXT)$ and store the weight or cost of the edge in the $COST$ field of the node which represents the edge, as depicted in Figure 9.10. If more information is associated with each edge in a graph, we simply add more fields in which to store the additional information. For instance, we may add a new field, say $LINK$, to link the two nodes which represent the same edge (i, j) in an undirected graph. Graph traversals classify the edges of a graph into one of four types; this information may be stored in a field, say $TYPE$. Additional information pertaining to the vertices can be stored in an array or arrays parallel to $LIST$. Clearly, the adjacency lists representation of a graph is quite robust.

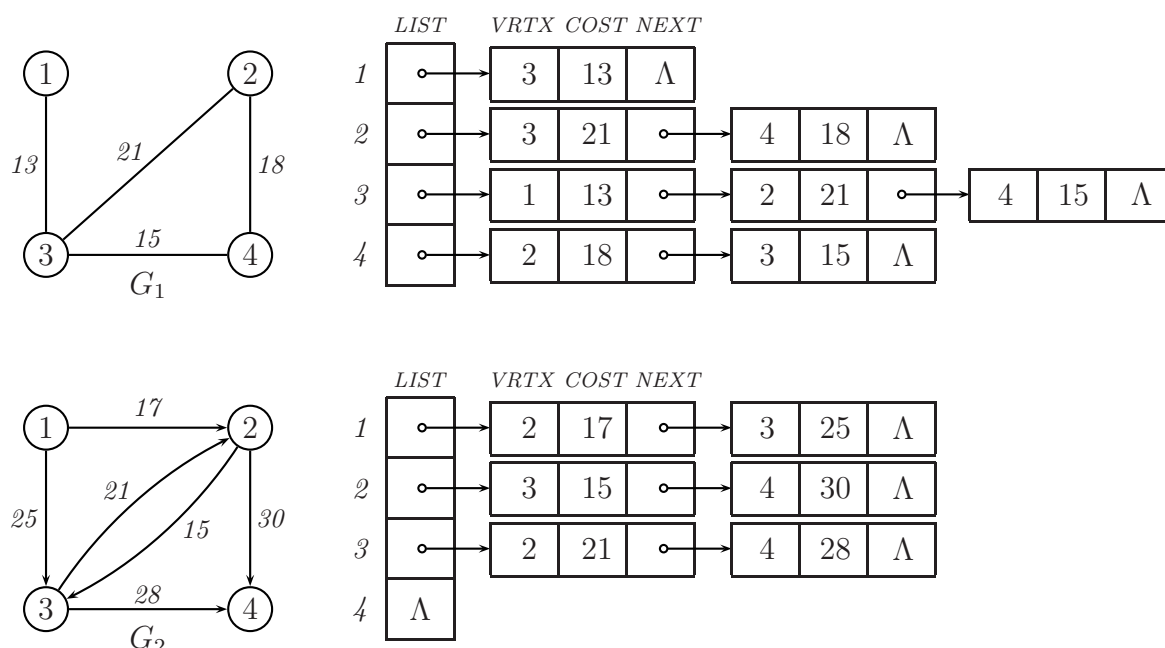


Figure 9.10 Adjacency lists representation of a weighted graph

9.3 Graph traversals

To traverse a graph $G = (V, E)$ is to **explore its edges** to **discover its vertices**. A graph traversal yields important information about the structure of a graph, and it serves as a ‘template’ for many important algorithms on graphs. A graph traversal differs in many important respects from that of a binary tree or a forest.

- A graph does not have the equivalent of a root node from which a traversal can naturally commence. This means that traversal of a graph can start at any vertex. However, not all the vertices in the graph may be reachable from this start vertex, leaving certain vertices undiscovered. Hence, traversal must be restarted from another, as yet undiscovered, vertex. This procedure is repeated until all vertices in the graph are discovered.
- There is no natural order among the vertices adjacent to some discovered vertex to indicate which one is discovered next. Which vertex is actually discovered next depends largely on the way the graph is represented.
- In a graph, a discovered vertex may be adjacent to several vertices from which it could be encountered again. It is therefore necessary to tag such a vertex as already discovered so that it is not discovered anew!

There are two general methods to traverse a graph, namely, **depth first search** (or **DFS**) and **breadth first search** (or **BFS**). The word ‘search’ in these names should explain our use of the terms *explore* and *discover* in describing these traversal algorithms.

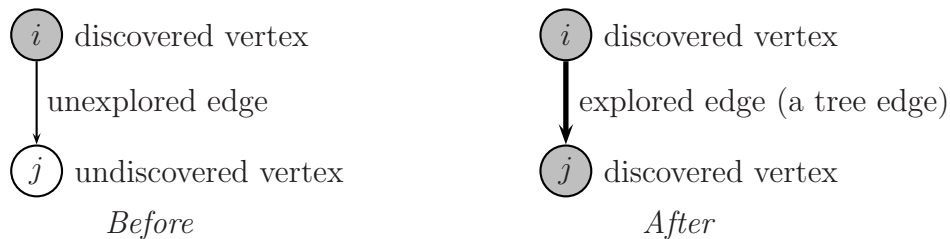
During DFS or BFS a vertex is either *undiscovered* or *discovered* and an edge is either *unexplored* or *explored*. A vertex is discovered only once (on first encounter), but it may

be encountered several more times during the search. A directed edge is explored only once (on the first and only encounter). An undirected edge is encountered twice; it is explored on first encounter, and ignored (no operation) on the second encounter.

To explore an edge (i, j) is to travel through it from the discovered vertex i to the still undiscovered, or already discovered, vertex j . In the former case, (i, j) is a *discovery edge*; in the latter, (i, j) is a *non-discovery edge*. A discovery edge is called a *tree edge*; a non-discovery edge is called either a *forward edge*, a *back edge* or a *cross edge*; why and how we classify edges this way will be explained shortly.

Figure 9.11 depicts the process of exploration and discovery during depth first search or breadth first search of a graph, directed or undirected.

Case 1. Unexplored edge leads to a still undiscovered vertex



Case 2. Unexplored edge leads to an already discovered vertex

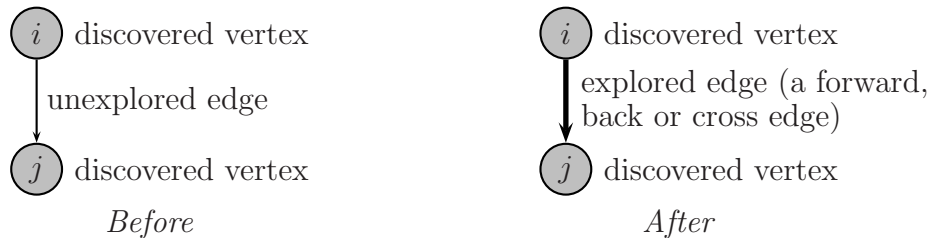


Figure 9.11 Exploring an edge to discover a vertex

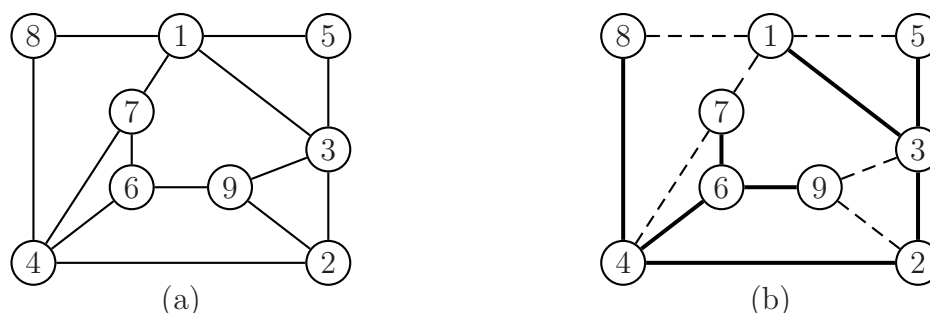
9.3.1 Depth first search

In DFS of a graph $G = (V, E)$ the search begins from some start vertex, say s , which is the first vertex to be discovered. Subsequently, an edge incident from s is explored to discover another vertex. The search then continues by exploring, each time, an edge incident from the *most recently* discovered vertex; thus the search proceeds farther and farther from the start vertex and ‘deeper’ into the graph. If all the edges leaving the most recently discovered vertex, say j , are found to have been already explored (i.e., the search has reached a ‘dead end’), then DFS *backtracks* to the vertex, say i , from which j was discovered, and explores an edge leaving i to discover another vertex, if any. This process of searching in the deeper direction, backtracking when a dead end is reached, then searching deeper into the graph again, continues until all vertices reachable from the start vertex s are discovered. The search started from s terminates after DFS backtracks all the way back to s and finds all edges incident from s already explored. If there are still

undiscovered vertices after the search initiated from s terminates, then a new search is started from any one of these undiscovered vertices. This entire process is repeated until all vertices in G are discovered.

We now illustrate DFS, in its most elementary form, with two examples. As a rule, when confronted with several candidate vertices to be discovered, we will choose the vertex with the smallest numerical label.

Example 9.1. DFS of an undirected graph

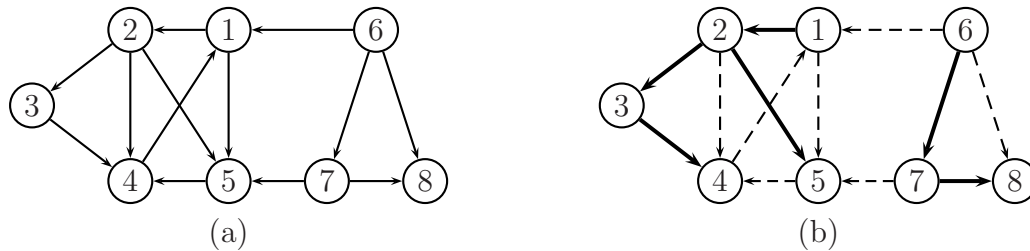


- a. Select vertex 1 as the start vertex and consider it as discovered.
- b. Explore edge (1, 3) to discover vertex 3.
- c. Explore edge (3, 2) to discover vertex 2.
- d. Explore edge (2, 4) to discover vertex 4.
- e. Explore edge (4, 6) to discover vertex 6.
- f. Explore edge (6, 7) to discover vertex 7.
- g. Explore edge (7, 1); vertex 1 already discovered.
- h. Explore edge (7, 4); vertex 4 already discovered.
- i. All edges incident on vertex 7 already explored; backtrack to vertex 6.
- j. Explore edge (6, 9) to discover vertex 9.
- k. Explore edge (9, 2); vertex 2 already discovered.
- l. Explore edge (9, 3); vertex 3 already discovered.
- m. All edges incident on vertex 9 already explored; backtrack to vertex 6.
- n. All edges incident on vertex 6 already explored; backtrack to vertex 4.
- o. Explore edge (4, 8) to discover vertex 8.
- p. Explore edge (8, 1); vertex 1 already discovered.
- q. All edges incident on vertex 8 already explored; backtrack to vertex 4.
- r. All edges incident on vertex 4 already explored; backtrack to vertex 2.
- s. All edges incident on vertex 2 already explored; backtrack to vertex 3.
- t. Explore edge (3, 5) to discover vertex 5.
- u. Explore edge (5, 1); vertex 1 already discovered.
- v. All edges incident on vertex 5 already explored; backtrack to vertex 3.
- w. All edges incident on vertex 3 already explored; backtrack to vertex 1.
- x. All edges incident on vertex 1 already explored; DFS terminates.

Figure 9.12 Depth first search of an undirected graph

In Figure 9.12(b), the solid lines are the discovery edges and the dashed lines are the non-discovery edges. The graph consisting of all the vertices and the discovery edges is the **predecessor subgraph** generated by DFS when initiated from vertex 1 of the given undirected graph in Figure 9.12(a).

Example 9.2. DFS of a directed graph



- a. Select vertex 1 as the start vertex and consider it as discovered.
- b. Explore edge (1, 2) to discover vertex 2.
- c. Explore edge (2, 3) to discover vertex 3.
- d. Explore edge (3, 4) to discover vertex 4.
- e. Explore edge (4, 1); vertex 1 already discovered.
- f. All edges incident from vertex 4 already explored; backtrack to vertex 3.
- g. All edges incident from vertex 3 already explored; backtrack to vertex 2.
- h. Explore edge (2, 4); vertex 4 already discovered.
- i. Explore edge (2, 5) to discover vertex 5.
- j. Explore edge (5, 4); vertex 4 already discovered.
- k. All edges incident from vertex 5 already explored; backtrack to vertex 2.
- l. All edges incident from vertex 2 already explored; backtrack to vertex 1.
- m. Explore edge (1, 5); vertex 5 already discovered.
- n. All edges incident from vertex 1 already explored; DFS from vertex 1 terminates.
- o. Select vertex 6 as the next start vertex and consider it as discovered.
- p. Explore edge (6, 1); vertex 1 already discovered.
- q. Explore edge (6, 7) to discover vertex 7.
- r. Explore edge (7, 5); vertex 5 already discovered.
- s. Explore edge (7, 8) to discover vertex 8.
- t. No edge is incident from vertex 8; backtrack to vertex 7.
- u. All edges incident from vertex 7 already explored; backtrack to vertex 6.
- v. Explore edge (6, 8); vertex 8 already discovered.
- w. All edges incident from vertex 6 already explored; DFS from vertex 6 terminates.

Figure 9.13 Depth first search of a directed graph

In Figure 9.13(b), the solid arrows are the discovery edges and the dashed arrows are the non-discovery edges. The graph consisting of all the vertices and the discovery edges is the predecessor subgraph generated by DFS when initiated from vertices 1 and 6 of the given directed graph in Figure 9.13(a).

Timestamping the vertices of a graph during DFS

With the preceding two examples as our vehicle for exposition, let us now look more deeply into depth-first search. To gain a deeper understanding of the way DFS works let us think in terms of a ‘container’ where vertices are temporarily stored. Undiscovered vertices are those which have not been placed in the container; discovered vertices are those which are in the container or those which have already been taken out of the container. A vertex is placed in the container the moment it is discovered, and it stays there for as long as there are unexplored edges leaving it; it is taken out of the container once all edges leaving it have been explored. We may therefore classify discovered vertices as either *unfinished* (those still in the container) or *finished* (those already taken out). Figure 9.14 shows the three states which every vertex in a graph undergoes during DFS; following CORMEN[2001], we color code the vertices to indicate their current state as an aid in describing the properties of DFS.

State	Status	Color
Undiscovered	All edges entering the vertex are unexplored.	white
Discovered		
Unfinished	Some edges leaving the vertex are still unexplored.	gray
Finished	All edges leaving the vertex are already explored.	black

Figure 9.14 The three successive states of a vertex during DFS

A careful study of the the examples shown in Figure 9.12 and 9.13 reveals that the order in which vertices are placed in, and subsequently taken out of, the container is *last-in first-out*; thus our container is actually a *stack*. Figure 9.15 shows the order in which the vertices of the graph in Example 9.1 are processed by DFS. A vertex is grayed at the moment it is discovered and pushed onto the stack, and it is blackened at the moment it is finished and popped from the stack. It is useful to record these critical moments when DFS discovers a vertex and when DFS finishes processing a vertex with a *timestamp*, which is simply an integer between 1 and $2n$. These timestamps are shown below each vertex in Figure 9.15; the number below a gray vertex is its *discovery time* and the number below a black vertex is its *finishing time*. For any vertex i we denote these timestamps as $d(i)$ and $f(i)$, respectively; for instance, $d(6) = 5$ and $f(6) = 10$.



Figure 9.15 DFS discovery and finishing times

Classifying the edges of a graph during DFS

In addition to timestamping each of the vertices of a graph $G = (V, E)$, DFS also classifies each of the edges of G as either a tree, forward, back or cross edge. Such a classification of the edges of a graph is particularly useful, in fact necessary, for many algorithms on graphs, as we will shortly see. To understand the basis for this classification, consider again the undirected graph in Example 9.1 and the predecessor subgraph generated by DFS when initiated from vertex 1, reproduced below as Figure 9.16(a) and (b). Suppose we take the graph in Figure 9.16(b) by the start vertex 1 and let the other vertices hang down, held together by the solid edges. What we obtain is a tree, as shown in Figure 9.16(c); such a tree is called a **depth-first tree** for the undirected graph shown in Figure 9.16(a). The tree is rooted at the vertex from where DFS is initiated, in this case, vertex 1. The edges shown as solid lines are called **tree edges** and the edges shown as dashed lines are called **back edges**. If the given undirected graph is not connected, DFS will yield a **depth-first forest** consisting of trees rooted at the vertices from where DFS is started.

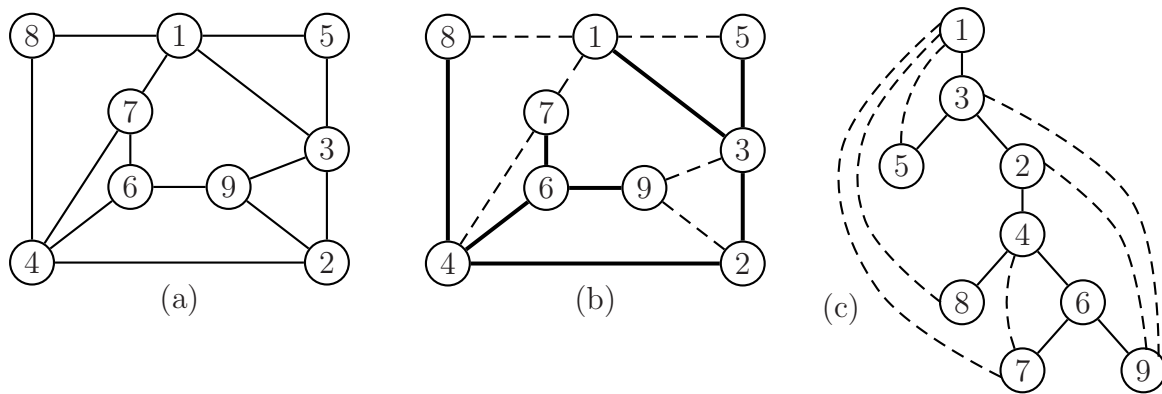


Figure 9.16 A depth-first tree for a connected undirected graph

Consider now the directed graph in Example 9.2 and the predecessor subgraph generated by DFS when initiated from vertices 1 and 6, reproduced below as Figure 9.17(a) and (b). This time we obtain a depth-first forest of two trees rooted at 1 and 6, and four types of edges, as shown in Figure 9.17(c). The edges shown as solid lines are called **tree edges**, as before. The edges shown as dashed lines are called **back edges** (B) and **forward edges** (F); the edges shown as dotted lines are called **cross edges**.

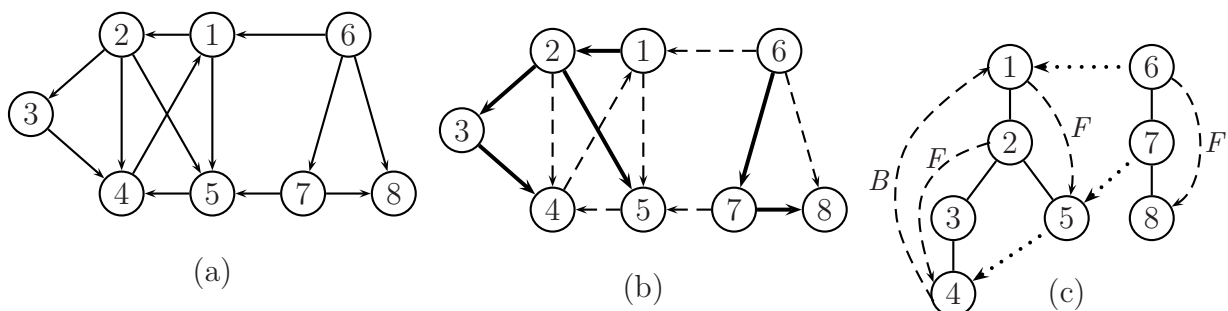


Figure 9.17 A depth-first forest for a directed graph

In terms of the depth-first tree or forest generated by DFS, these various types of edges may be characterized as follows:

- (a) A tree edge is a branch in a depth-first tree.
- (b) A back edge (i, j) is an edge connecting vertex i to an *ancestor* j in a depth-first tree.
- (c) A forward edge (i, j) is a non-tree edge connecting vertex i to a *descendant* j in a depth-first tree.
- (d) A cross edge (i, j) is an edge connecting vertices i and j where i is neither an ancestor nor a descendant of j . (For instance, i and j might be brothers, cousins, nephew-uncle, and so on.)

Let vertex i be a discovered vertex and let edge (i, j) be an unexplored edge. Depending on the color of vertex j and on the discovery time of i relative to j , edge (i, j) becomes, after it is explored, a tree, back, forward or cross edge according to the rules stated below. These rules allow us to operationalize the classification of edges into the types enumerated above.

- (a) Edge (i, j) becomes a tree edge if vertex j is white (undiscovered) when edge (i, j) is explored from i .
- (b) Edge (i, j) becomes a back edge if vertex j is gray (discovered but unfinished) when edge (i, j) is explored from i .
- (c) Edge (i, j) becomes a forward edge if vertex j is black (finished) and $d(i) < d(j)$ when edge (i, j) is explored from i .
- (d) Edge (i, j) becomes a cross edge if vertex j is black (finished) and $d(i) > d(j)$ when edge (i, j) is explored from i .

(9.3)

Identifying descendants in a depth-first forest

Consider again Figure 9.15 which depicts the DFS discovery and finishing times for the undirected graph $G = (V, E)$ in Figure 9.12 of Example 9.1. The figure is reproduced below with arrows pictorially depicting when a vertex is discovered and when it is finished.

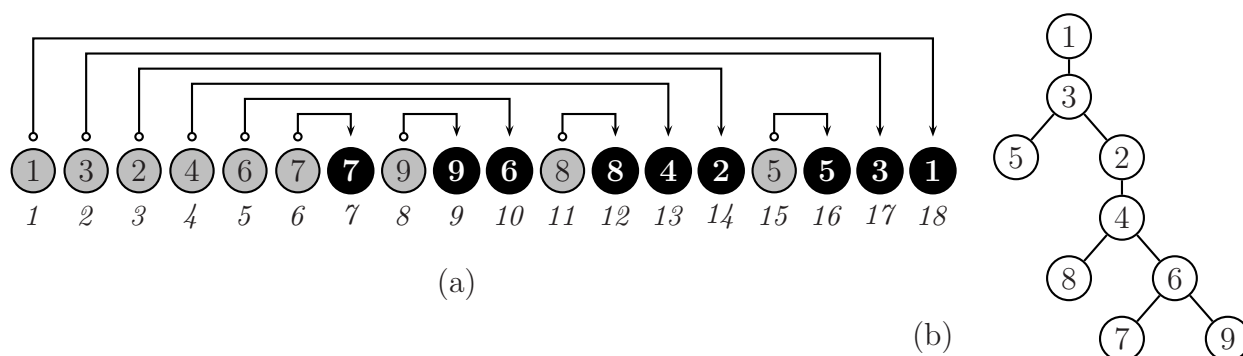
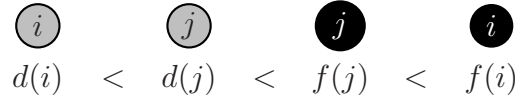


Figure 9.18 Nesting of DFS discovery and finishing times and corresponding depth-first tree

Note that arrows do not cross each other; this is because in DFS the last vertex to be discovered is the first to finish. Thus, if I_1 is the interval $[d(i), f(i)]$ and I_2 is the interval $[d(j), f(j)]$ for any two vertices i and j in G , then either I_1 contains I_2 entirely, or I_1 and I_2 are disjoint intervals. (If I_2 contains I_1 , we swap the vertex names to get the former case.) Figure 9.19 summarizes these two cases.

Case 1. Vertex j is discovered after vertex i , and j finishes before i
(j is a descendant of i)



Case 2. Vertex i is discovered and finishes before vertex j is discovered
(j is not a descendant of i)



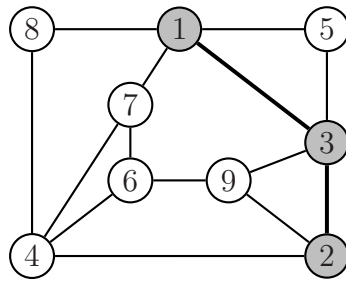
Figure 9.19 Identifying descendants and non-descendants in a depth-first forest

In Case 1, j is a descendant of i ; all vertices that are discovered after i is discovered and before i finishes are descendants of i . In case 2, vertex j is discovered after i finishes, and is therefore not a descendant of i . For instance, we find in Figure 9.18(a) that the interval $[d(2), f(2)] = [3, 14]$ contains the intervals $[4, 13]$, $[5, 10]$, $[6, 7]$, $[8, 9]$ and $[11, 12]$ for vertices 4, 6, 7, 9 and 8, respectively, but not the interval $[15, 16]$ for vertex 5; thus vertices 4, 6, 7, 9 and 8 are descendants of vertex 2 but vertex 5 is not, as we can readily see in the depth-first tree in Figure 9.18(b). In particular, we find that the interval $[d(1), f(1)] = [1, 16]$ for the start vertex 1 contains the intervals for all the other vertices; all these vertices are the descendants of vertex 1 which is the root of the depth-first tree. We may cast this observation as a rule:

All the undiscovered vertices reachable from a start vertex s become descendants of s in the resulting depth-first forest. (9.4)

This rule applies to all start vertices in DFS. In Example 9.2, when DFS is initiated from vertex 1 all the other vertices are undiscovered but only vertices 2, 3, 4 and 5 are reachable from 1; these become the descendants of 1 in the resulting depth-first forest. When DFS is re-initiated from the new start vertex 6, the still undiscovered vertices are 7 and 8 and are both reachable from 6; hence they become descendants of 6, as you can verify from Figure 9.17(c).

For some other vertex $i \neq s$, not all undiscovered vertices reachable from i at the time DFS discovers i necessarily become descendants of i . To see this consider the state of the search when DFS discovers vertex 2 of the undirected graph of Example 9.1 as shown below.



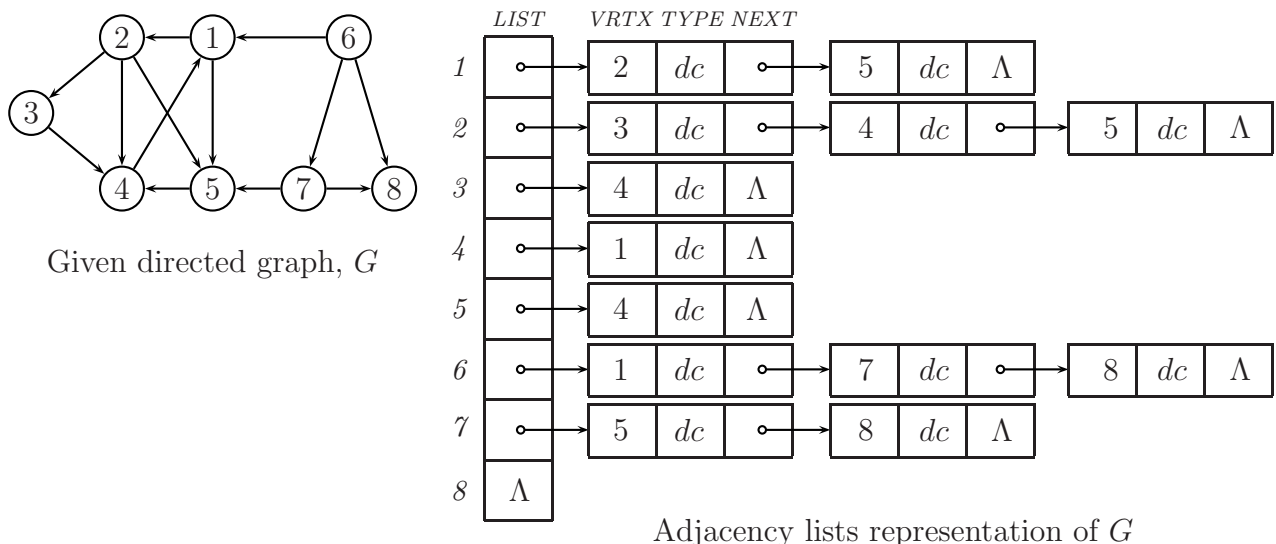
At this point in the search the still undiscovered vertices are 4, 5, 6, 7, 8 and 9, all of which are reachable from vertex 2. While vertices 4, 6, 7, 8 and 9 become descendants of vertex 2, vertex 5 does not, as you can see in the depth-first tree shown in Figure 9.18(b). What distinguishes vertex 5 from the others is that all the paths which reach vertex 5 from vertex 2 contain one or more already discovered vertices. In contrast, each of the other undiscovered vertices is reachable from vertex 2 along a path that consists entirely of undiscovered vertices. We can formalize this observation as another rule:

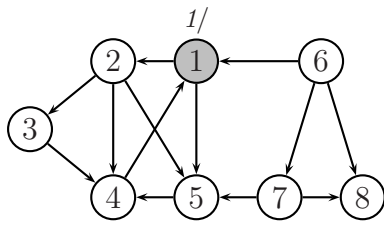
Vertex j becomes a descendant of vertex i in a depth-first forest of an undirected or directed graph if at the time DFS discovers vertex i , vertex j can be reached from vertex i along a path consisting entirely of undiscovered vertices. (9.5)

In summary, we have two ways of identifying descendants in a depth-first tree: by comparing discovery and finishing times as indicated in Figure 9.19 or by applying the two rules given above.

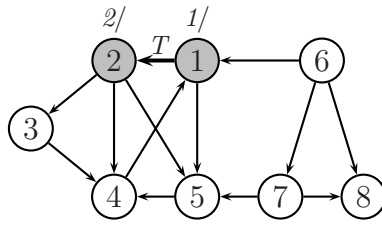
Summing up

Let us now rework Example 9.2 given earlier to illustrate how vertices are timestamped and how edges are classified during DFS. In Figure 9.20, the discovery time and finishing time for each vertex are indicated by the pair xx/xx placed alongside the vertex. The letters T , B , F or X placed on an edge indicates that the edge is a tree, back, forward or cross edge.

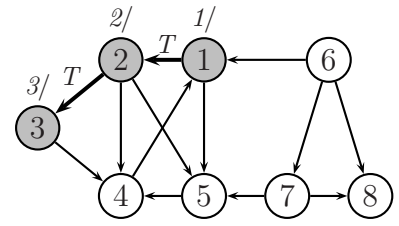




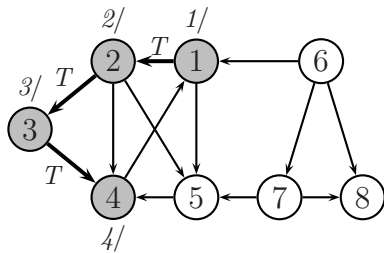
(a) Select 1 as start vertex and consider it as discovered.



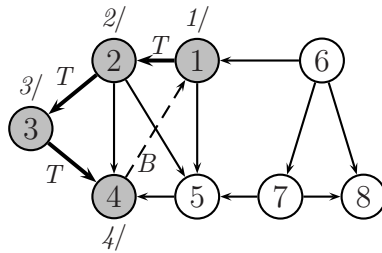
(b) Explore edge (1,2) to discover vertex 2. Edge (1,2) is a tree edge.



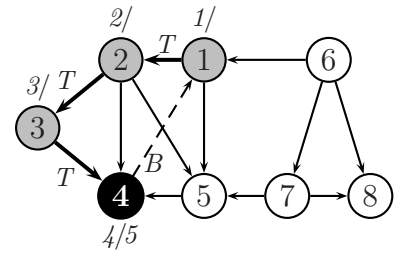
(c) Explore edge (2,3) to discover vertex 3. Edge (2,3) is a tree edge.



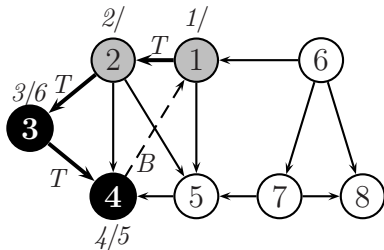
(d) Explore edge (3,4) to discover vertex 4. Edge (3,4) is a tree edge.



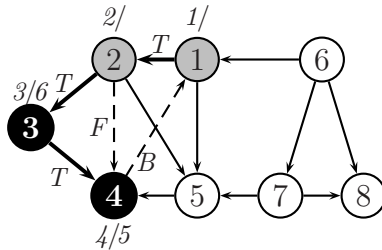
(e) Explore edge (4,1); vertex 1 is gray. Edge (4,1) is a back edge.



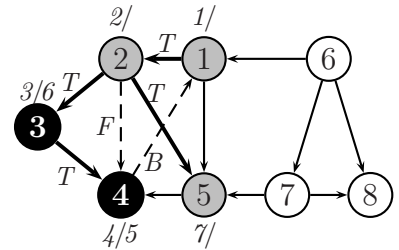
(f) All edges incident from 4 explored; 4 is finished. Backtrack to vertex 3.



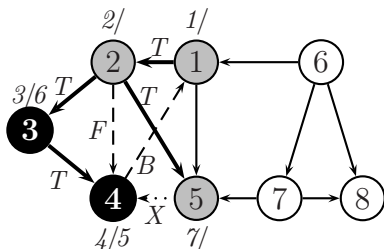
(g) All edges incident from 3 explored; 3 is finished. Backtrack to vertex 2.



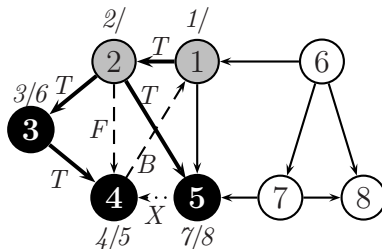
(h) Explore edge (2,4); vertex 4 is black and $d(2) < d(4)$. Edge (2,4) is a forward edge.



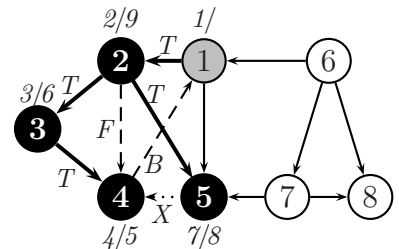
(i) Explore edge (2,5) to discover vertex 5. Edge (2,5) is a tree edge.



(j) Explore edge (5,4); vertex 4 is black and $d(5) > d(4)$. Edge (5,4) is a cross edge.

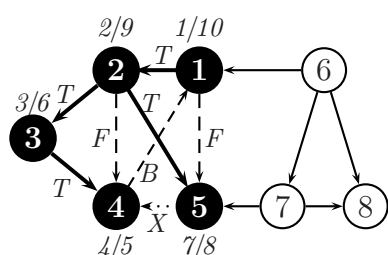


(k) All edges incident from 5 explored; 5 is finished. Backtrack to vertex 2.

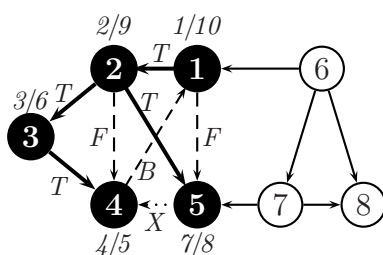


(l) All edges incident from 2 explored; 2 is finished. Backtrack to vertex 1.

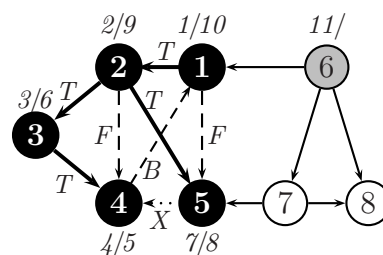
Figure 9.20 Depth-first search of a directed graph (continued on next page)



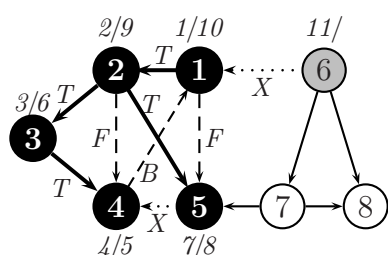
(m) Explore edge (1,5); vertex 5 is black and $d(1) < d(5)$. Edge (1,5) is a forward edge.



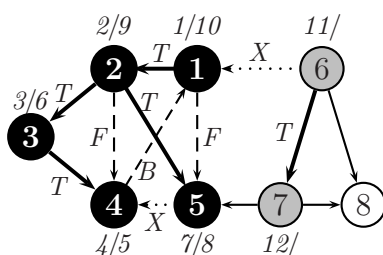
(n) All edges incident from 1 explored; 1 is finished. DFS initiated from 1 terminates.



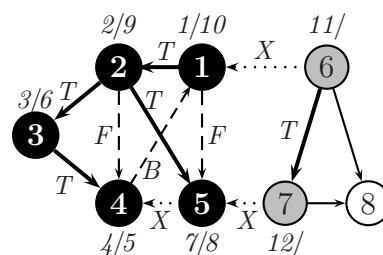
(o) Select 6 as new start vertex and consider it as discovered.



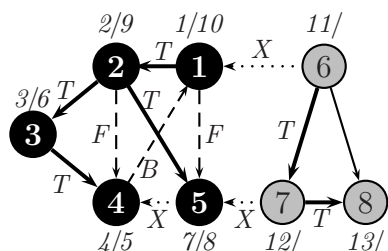
(p) Explore edge (6,1); vertex 1 is black and $d(6) > d(1)$. Edge (6,1) is a cross edge.



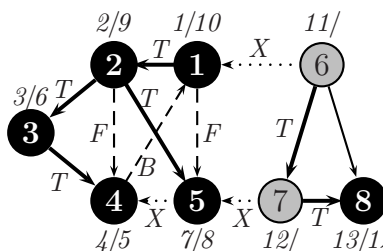
(q) Explore edge (6,7) to discover vertex 7. Edge (6,7) is a tree edge.



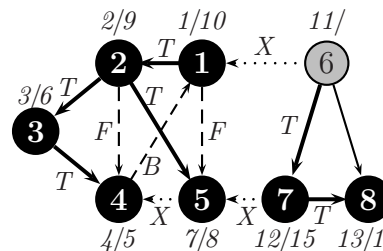
(r) Explore edge (7,5); vertex 5 is black and $d(7) > d(5)$. Edge (7,5) is a cross edge.



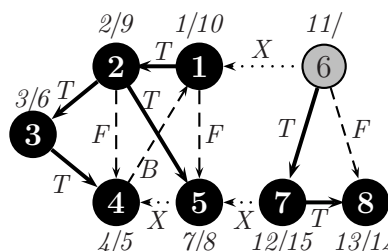
(s) Explore edge (7,8) to discover vertex 8. Edge (7,8) is a tree edge.



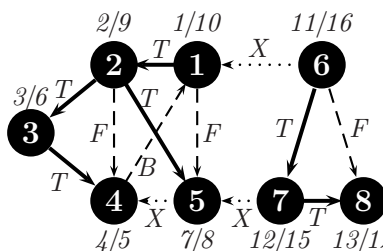
(t) No edge is incident from 8; 8 is finished. Backtrack to vertex 7.



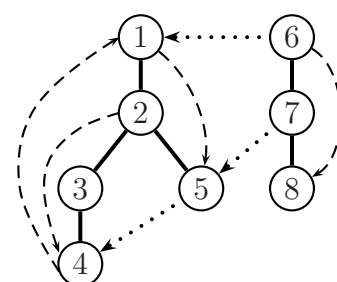
(u) All edges incident from 7 explored; 7 is finished. Backtrack to vertex 6.



(v) Explore edge (6,8); vertex 8 is black and $d(6) < d(8)$. Edge (6,8) is a forward edge.



(w) All edges incident from 6 explored; 6 is finished. DFS initiated from 6 terminates.



Depth-first forest

Figure 9.20 Depth-first search of a directed graph

EASY implementation of DFS

Procedure $\text{DFS}(\mathbb{G}, i)$ implements depth first search as described in the preceding sections. The input to the procedure is an undirected or directed graph $G = (V, E)$ and a start vertex i . The graph G is represented by the data structure $\mathbb{G} = [\text{LIST}(1:n), (\text{VRTX}, \text{TYPE}, \text{NEXT}), \text{pred}(1:n), d(1:n), f(1:n)]$. The first two components of \mathbb{G} comprise the adjacency lists representation of G , as depicted, for instance, in Figure 9.20. The remaining three components of \mathbb{G} may be viewed as ‘attributes’ of G discovered during DFS. Specifically, for any vertex i in G , $\text{pred}(i)$ is the father of i in the depth-first forest, or 0, if i is a root.

The output of the procedure consists of:

1. the resulting depth-first forest for G encoded in the pred array
2. the discovery and finishing times of each vertex in V encoded in the d and f arrays, respectively
3. the type of each edge in E encoded in the TYPE field of the node which represents the edge in the adjacency lists for G

This information about G is returned as part of the data structure \mathbb{G} .

```

1  procedure DFS( $\mathbb{G}, i$ )
2     $\text{color}(i) \leftarrow \text{gray}$ 
3     $d(i) \leftarrow \text{time} \leftarrow \text{time} + 1$ 
4     $\alpha \leftarrow \text{LIST}(i)$ 
5    while  $\alpha \neq \Lambda$  do
6       $j \leftarrow \text{VRTX}(\alpha)$ 
7      case
8        :  $\text{color}(j) = \text{white}$  : [  $\text{TYPE}(\alpha) \leftarrow T$ ;  $\text{pred}(j) \leftarrow i$ ; call DFS( $\mathbb{G}, j$ ) ]
9        :  $\text{color}(j) = \text{gray}$  : if  $\text{pred}(i) \neq j$  then  $\text{TYPE}(\alpha) \leftarrow B$ 
10       :  $\text{color}(j) = \text{black}$  : if  $d(i) < d(j)$  then  $\text{TYPE}(\alpha) \leftarrow F$  else  $\text{TYPE}(\alpha) \leftarrow X$ 
11      endcase
12       $\alpha \leftarrow \text{NEXT}(\alpha)$ 
13    endwhile
14     $\text{color}(i) \leftarrow \text{black}$ 
15     $f(i) \leftarrow \text{time} \leftarrow \text{time} + 1$ 
16  end DFS

```

Procedure 9.1 Depth-first search

The search is initiated from the undiscovered (white) vertex i (line 1). In line 2, vertex i is painted gray to indicate that it is now discovered; in line 3, the global variable time is updated and is assigned as the discovery time of vertex i . Lines 4 to 13 examine the adjacency list of vertex i . For each vertex j adjacent to vertex i , DFS classifies edge (i, j) as a tree (T), back (B), forward (F) or cross (X) edge according to Rules (9.3); the

edge type is stored in the *TYPE* field of the node representing edge (i, j) in the adjacency list of vertex i . If vertex j is discovered while exploring edge (i, j) (line 8), vertex i is recorded as j 's predecessor, after which the search continues from j (the most recently discovered vertex) by recursively calling DFS. The test in line 9 prior to classifying edge (i, j) as a back edge is necessary if (i, j) is an undirected edge (since the edge may have been explored earlier as (j, i) and classified as a tree edge); the test is irrelevant if (i, j) is a directed edge. Line 10 applies only to a directed graph; this case will never arise in an undirected graph. After DFS finishes examining i 's adjacency list, vertex i is painted black to indicate that all edges incident from it have already been explored (line 14). Finally, the timestamp *time* is updated once again and is assigned as the finishing time of vertex i (line 15).

Procedure DFS is invoked by a 'driver' procedure, as in DFS_DRIVER given below, which initializes the global variable *time* to 0, the global array *color* to *white* and the *pred* array to 0 (lines 2–4). The *TYPE* field need not be initialized when the adjacency list representation of the graph is generated for input to DFS_DRIVER; in Figure 9.20 this is indicated by the entry *dc* (which means *don't care*). Since initiating DFS from some vertex i in $G = (V, E)$ may not explore all the edges in E and discover all the vertices in V , DFS is invoked from *each undiscovered* vertex in G (lines 5–7).

```

1  procedure DFS_DRIVER( $\mathbb{G}$ )
2     $time \leftarrow 0$             $\triangleright$  global variable
3     $color \leftarrow white$      $\triangleright$  global array
4     $pred \leftarrow 0$ 
5    for  $i \leftarrow 1$  to  $n$  do
6      if  $color(i) = white$  then call DFS( $\mathbb{G}, i$ )
7    endfor
8    end DFS_DRIVER

```

Procedure 9.2 A sample calling procedure for procedure DFS

For instance, invoking DFS_DRIVER(\mathbb{G}) on the graph of Figure 9.20 returns the following components of \mathbb{G} :

	1	2	3	4	5	6	7	8
$pred(1:8)$	0	1	2	3	2	0	6	7
$d(1:8)$	1	2	3	4	7	11	12	13
$f(1:8)$	10	9	6	5	8	16	15	14

Analysis of DFS

During depth-first search of a graph $G = (V, E)$ on $n = |V|$ vertices and $e = |E|$ edges, procedure DFS is invoked only on white vertices (in line 6 of DFS_DRIVER or line 8 of DFS); i.e., it is called exactly n times. For each such call, say on vertex i , DFS explores all the edges incident from i , or equivalently, all the nodes in the adjacency list of vertex i . The total number of nodes comprising the adjacency lists for a directed graph is e and for an undirected graph is $2e$. Hence, DFS of a graph represented using adjacency lists takes $O(n + e)$ time. A version of procedure DFS for a graph represented using an adjacency matrix would take $O(n^2)$ time, since even non-existent edges are represented in the matrix.

9.3.2 Breadth first search

In BFS of a graph $G = (V, E)$ the search begins from some start vertex, say s , which is the first vertex to be discovered. Subsequently, every edge incident from s is explored to discover all the vertices adjacent to s , i.e., vertices that are one edge away from s . The search then continues by exploring edges incident from these latter vertices to discover, this time, all the vertices that are two edges away from s . Thus the search moves steadily forward across a wide front discovering vertices that are l edges away from s before discovering vertices that are $l + 1$ edges away from s , until all vertices reachable from s are discovered. Unlike in DFS, there is no backtracking in BFS; the search terminates once the farthest vertices reachable from s are discovered.

As in DFS, a BFS initiated from some start vertex s may not discover all the vertices in a graph. In such a case, a new search is started from any one of the undiscovered vertices; this process is repeated until all vertices in the graph are discovered.

Akin to DFS, we record the vertex, say i , from which vertex j is discovered, thus:

$$\text{pred}(j) = i$$

Unlike in DFS, we do not record the discovery time of j ; we record instead the distance, say $l(j)$, measured in number of edges, from the start vertex s to vertex j . With $l(s) = 0$, we have

$$l(j) = l(i) + 1 \quad \text{if } j \text{ is discovered from } i$$

As in DFS (see Figure 9.14), each vertex in a graph passes through three stages during breadth first search, as indicated in Figure 9.21. This time unfinished (gray) vertices are given the more descriptive name *fringe vertices*; fringe vertices constitute the frontier between undiscovered and finished vertices. As in DFS, we can think in terms of a container in which vertices are stored: undiscovered vertices have yet to be placed in the container, fringe vertices are those in the container and finished vertices are those no longer in the container. Since BFS discovers vertices in their order of increasing distance from the start vertex s (*all* those which are l edges away from s before *any* which is $l + 1$ edges away), the container behaves as a *queue* rather than as a *stack*.

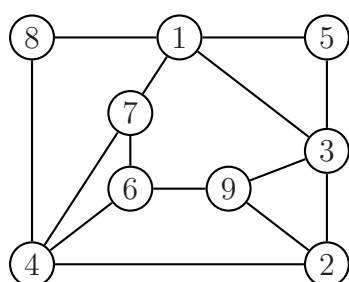
State	Status	Color
Undiscovered	All edges entering the vertex are unexplored.	white
Discovered		
Fringe	Some edges leaving the vertex are still unexplored.	gray
Finished	All edges leaving the vertex are already explored.	black

Figure 9.21 The three successive states of a vertex during BFS

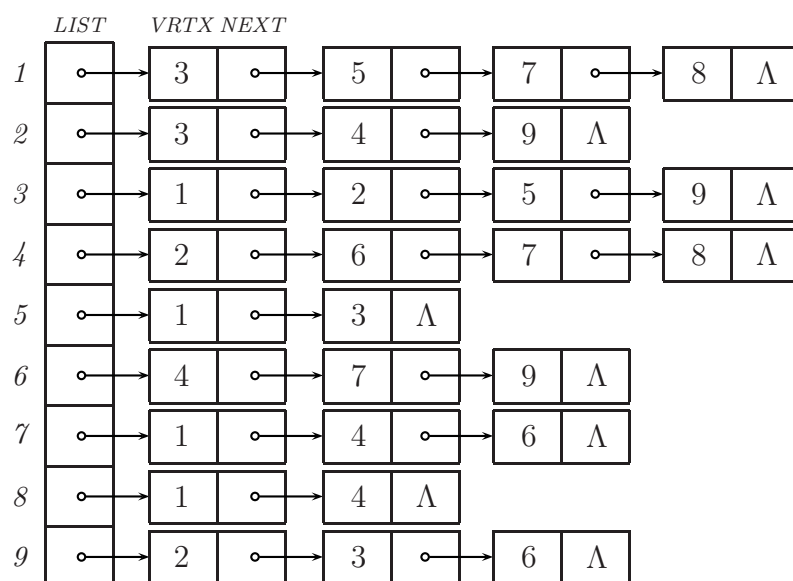
In a manner akin to DFS, BFS generates a forest, called a **breadth-first forest**, consisting of trees rooted at the vertices from where BFS is initiated. If all vertices in the graph are discovered from the start vertex s , the forest is a single tree called a **breadth-first tree**. Edges in a breadth-first forest are also classified as tree, back or cross edges (there are no forward edges); the three types are similarly defined as in a depth-first forest.

The following two examples illustrate BFS in action.

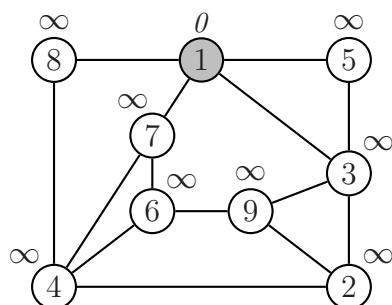
Example 9.3. Breadth-first search of an undirected graph



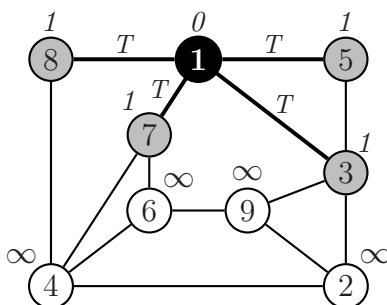
Given undirected graph G



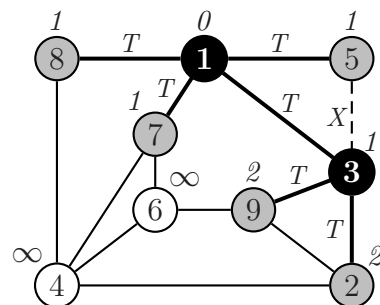
Adjacency-lists representation of G



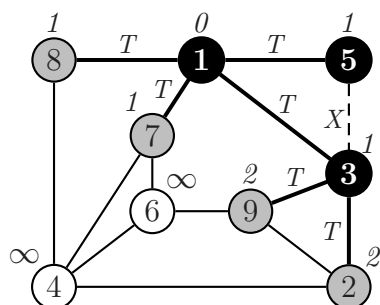
(a) Select 1 as start vertex and consider it as discovered.



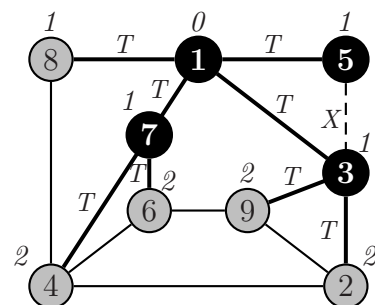
(b) Explore edges (1,3), (1,5), (1,7) and (1,8) to discover vertices 3, 5, 7 and 8. All edges incident on 1 explored; 1 is now finished.



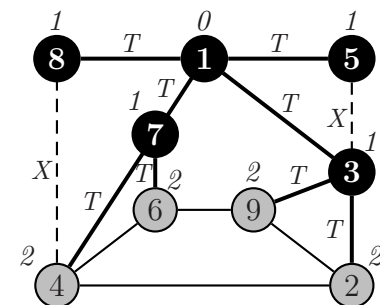
(c) Explore (3,2) to discover 2. Explore (3,5); 5 already discovered. Edge (3,5) is a cross edge. Explore (3,9) to discover 9. All edges incident on 3 explored; 3 is now finished.



(d) All edges incident on 5 already explored; 5 is finished.



(e) Explore (7,4) to discover 4. Explore (7,6) to discover 6. All edges incident on 7 explored; 7 is now finished.



(f) Explore (8,4); 4 already discovered. Edge (8,4) is a cross edge. All edges incident on 8 explored; 8 is now finished.

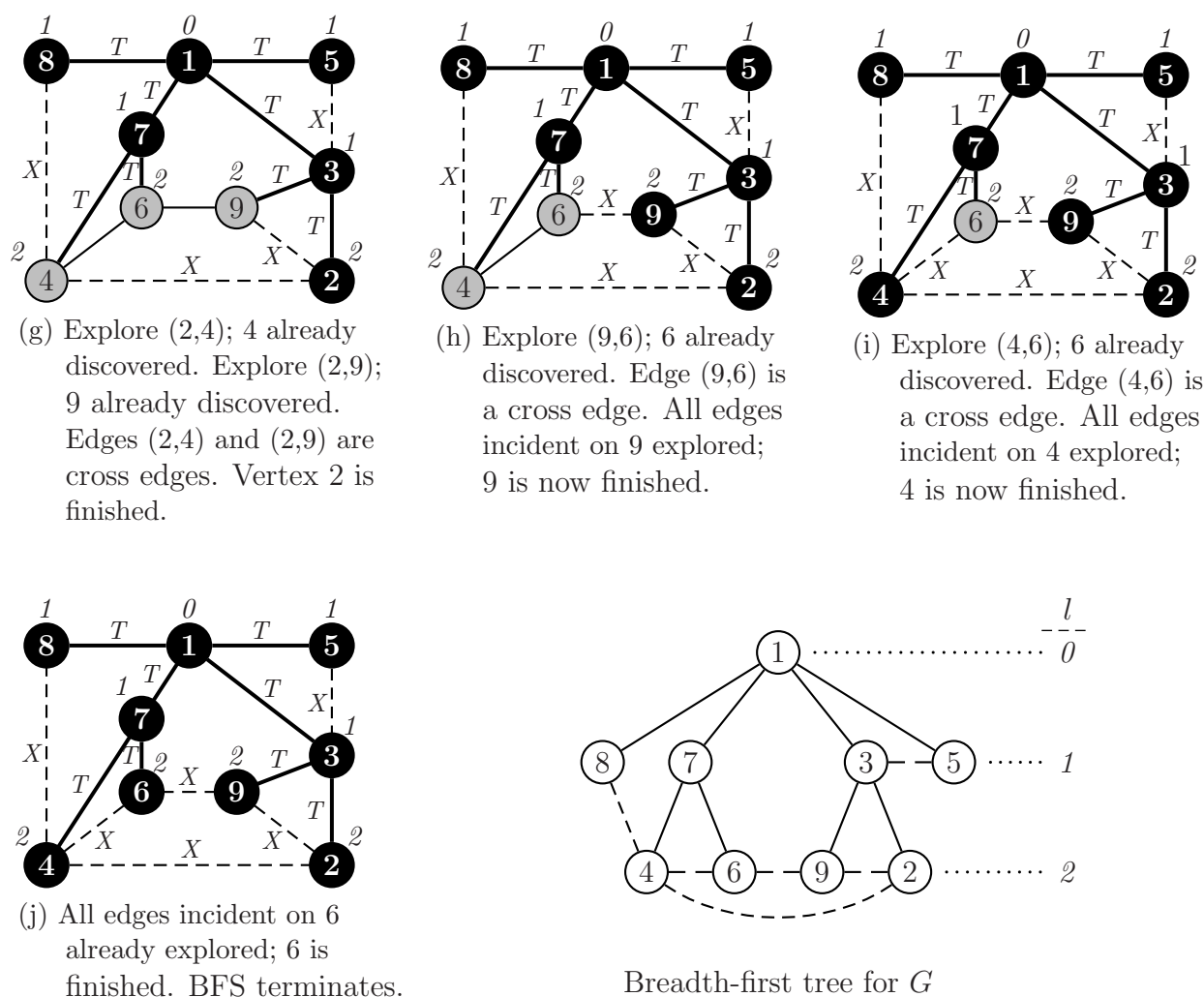


Figure 9.22 Breadth-first search of an undirected graph

We take note of the following observations pertaining to the above example.

1. There is only one tree in the breadth-first forest because the given undirected graph G is connected.
2. The number placed alongside a vertex is its distance from the start vertex s . Undiscovered vertices are initially at a distance ∞ from s ; discovered vertices are at a distance $l = 0, 1$ or 2 . Note that l is actually the *level* of a vertex in the resulting breadth-first tree. For any vertex i , $l(i)$ is the distance along a shortest path from s to i ; thus a breadth-first tree is a shortest-paths tree rooted at s for G .
3. There are only two types of edges in a breadth-first forest for an undirected graph: tree edges and cross edges. If j is undiscovered when edge (i, j) is explored from i , then edge (i, j) becomes a tree edge (j is i 's son in the breadth-first tree). If j is already discovered when edge (i, j) is explored from i , then edge (i, j) becomes a cross edge (j is i 's brother, cousin or uncle in the tree).

Example 9.4. Breadth first search of a directed graph

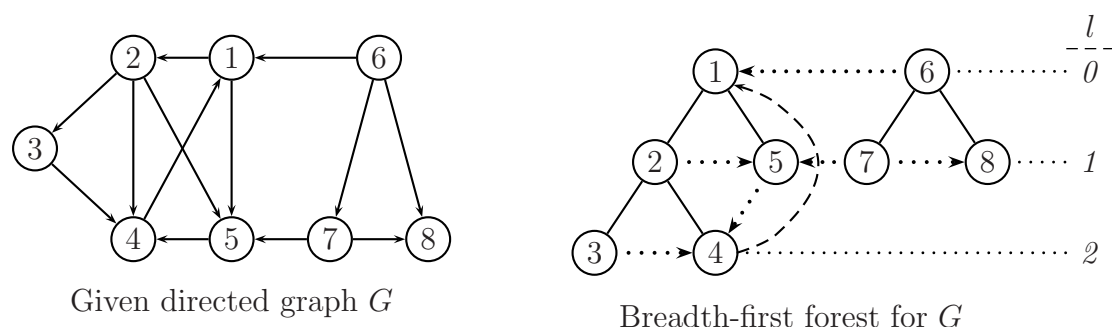


Figure 9.23 Breadth-first search of a directed graph

Breadth-first search of the directed graph G yields the breadth-first forest shown, as you may easily verify. We note the following:

1. BFS started from vertex 1 discovers vertices 2 thru 5 only; a new BFS initiated from 6 discovers the remaining vertices. Thus the resulting breadth-first forest consists of two trees rooted at 1 and 6 respectively.
2. The value of $l(i)$ for any vertex i in a tree is the distance along a shortest path from the start vertex (root) to vertex i . The trees rooted at 1 and 6 are shortest-paths trees for G .
3. There are three types of edges in a breadth-first forest of a directed graph: tree, back (shown as dashed lines) and cross edges (shown as dotted lines). Since BFS discovers vertices from one 'generation' to the next, there cannot be forward edges in a breadth-first tree. The only way vertex j can be a descendant of vertex i is if j is a son of i , in which case edge (i, j) is a tree edge. It is of course possible for j to be an ancestor of i ; then edge (i, j) is a back edge.

EASY implementation of BFS

Procedure $\text{BFS}(G, s)$ implements breadth first search as described in the preceding sections. The input to the procedure is an undirected or directed graph $G = (V, E)$ and a start vertex s . The graph G is represented by the data structure $\mathbb{G} = [\text{LIST}(1:n), (\text{VRTX}, \text{NEXT}), \text{pred}(1:n), l(1:n)]$. The first two components of \mathbb{G} comprise the adjacency lists representation of G , as depicted, for instance, in Figure 9.22. The remaining two components of \mathbb{G} may be viewed as 'attributes' of G discovered during BFS, and these comprise the output. Specifically, the resulting breadth-first forest is encoded in the pred array, and the length of the shortest path from the root to each vertex in a tree is stored in the l array.

Procedure BFS uses a queue, \mathbb{Q} , as the container for the fringe vertices. Any of the implementations of \mathbb{Q} given in Session 5 may be used.

```

1  procedure BFS( $\mathbb{G}, s$ )
2  call InitQueue( $\mathbb{Q}$ )
3   $color(s) \leftarrow gray$ 
4  call ENQUEUE( $\mathbb{Q}, s$ )
5   $l(s) \leftarrow 0$ 
6  while not IsEmptyQueue( $\mathbb{Q}$ ) do
7      call DEQUEUE( $\mathbb{Q}, i$ )
8       $\alpha \leftarrow LIST(i)$ 
9      while  $\alpha \neq \Lambda$  do
10          $j \leftarrow VRTX(\alpha)$ 
11         if  $color(j) = white$  then [  $color(j) \leftarrow gray$ ;  $pred(j) \leftarrow i$ 
12                                      $l(j) \leftarrow l(i) + 1$ ; call ENQUEUE( $\mathbb{Q}, j$ ) ]
13          $\alpha \leftarrow NEXT(\alpha)$ 
14     endwhile
15      $color(i) \leftarrow black$ 
16 endwhile
17 end BFS

```

Procedure 9.3 Breadth-first search

The search is initiated from the undiscovered (white) vertex s (line 1). In line 3, vertex s is painted gray to indicate that it is now discovered and in line 4 it is placed in the container \mathbb{Q} of gray vertices. In line 5, the distance from s to itself is recorded as zero.

Lines 6 thru 16 performs the search for all vertices reachable from s . The search continues for as long as there are gray vertices in the queue, i.e., vertices with still unexplored edges which may lead to still undiscovered vertices; the search from s terminates once the queue becomes empty. In line 7 the vertex at the front of the queue, say vertex i , is dequeued (but not blackened yet) so that its adjacency list can be processed (lines 8–14). BFS now discovers, in turn, *all* the vertices adjacent to i that are still undiscovered. For each such vertex, say j , the discovery is manifested by coloring vertex j gray, recording i as its predecessor and computing its distance from s as that of its predecessor plus 1, after which it is inserted at the rear of the queue (lines 11–12). Once processing of the adjacency list of vertex i is done, i is blackened (line 15) to indicate that it is now a finished vertex. Subsequently the front element of the queue, if any, is dequeued and processed similarly.

Unlike procedure DFS, procedure BFS does not classify edges in G during the search. If such a classification is desired, the appropriate code should be incorporated.

Procedure BFS is invoked by a ‘driver’ procedure, as in BFS_DRIVER given below, which initializes the global array *color* to *white*, the *pred* array to 0 and the *l* array to ∞ (lines 2–4). As in DFS, initiating BFS from some start vertex s may not discover all the vertices in G ; thus BFS is invoked from *each undiscovered* vertex in G (lines 5–7).

```

1  procedure BFS_DRIVER( $\mathbb{G}$ )
2     $color \leftarrow white$   $\triangleright$  global array
3     $pred \leftarrow 0$ 
4     $l \leftarrow \infty$ 
5    for  $s \leftarrow 1$  to  $n$  do
6      if  $color(s) = white$  then call BFS( $\mathbb{G}, s$ )
7    endfor
8  end BFS_DRIVER

```

Procedure 9.4 A sample calling procedure for procedure BFS

Analysis of BFS

During breadth-first search of a graph $G = (V, E)$ on $n = |V|$ vertices and $e = |E|$ edges, a vertex is enqueued only once (when it is discovered) and dequeued only once (when its adjacency list is processed) after which it is blackened to indicate it is finished. Since each enqueue and dequeue operation takes $O(1)$ time (see Session 5), enqueueing and dequeuing n vertices take $O(n)$ time. For each dequeued vertex i , BFS explores all the edges incident from i , or equivalently, all the nodes in the adjacency list of vertex i . The total number of nodes comprising the adjacency lists for a directed graph is e and for an undirected graph is $2e$. Since exploring an edge takes $O(1)$ time (lines 10–12 of procedure BFS), exploring all the edges takes $O(e)$ time. Initializing the $color$, $pred$ and l arrays in lines 2–4 of BFS_DRIVER takes $O(n)$ time. Hence, BFS of a graph represented using adjacency lists takes $O(n + e)$ time. A version of procedure BFS for a graph represented using an adjacency matrix would take $O(n^2)$ time, since even non-existent edges are represented in the matrix.

Summary of edge types in DFS and BFS

In many applications of DFS and BFS to problems on both undirected and directed graphs, the type of an edge in a depth-first or breadth-first tree plays an important role in the algorithm to solve the problem. For handy reference in the examples which follow, the types of edges generated by DFS and BFS are summarized in the table below.

	Undirected graph	Directed graph
DFS	Tree edges Back edges	Tree edges Forward edges Back edges Cross edges
BFS	Tree edges Cross edges	Tree edges Back edges Cross edges

Figure 9.24 Types of edges in depth-first and breadth-first forests

9.4 Some classical applications of DFS and BFS

In the rest of this session we will consider five applications of depth first search and breadth first search to problems on directed and undirected graphs. A common characteristic of these problems is that they are readily solved ‘by hand’ (or ‘by sight’); the challenge is solving them on a computer.

9.4.1 Identifying directed acyclic graphs

A *directed acyclic graph*, or *dag*, is a directed graph with no cycles. Earlier in Session 5 we already encountered dags (although we didn’t call them as such) in connection with partially ordered sets. If we represent the elements of a partially ordered set as vertices, and the relation $i < j$ for any two distinct elements i and j in the set as a directed edge from vertex i to vertex j , then we obtain a dag. For instance, Figure 5.6 which depicts the prerequisite structure among the CS courses in the BSCS curriculum, reproduced below as Figure 9.25(a), is a dag.

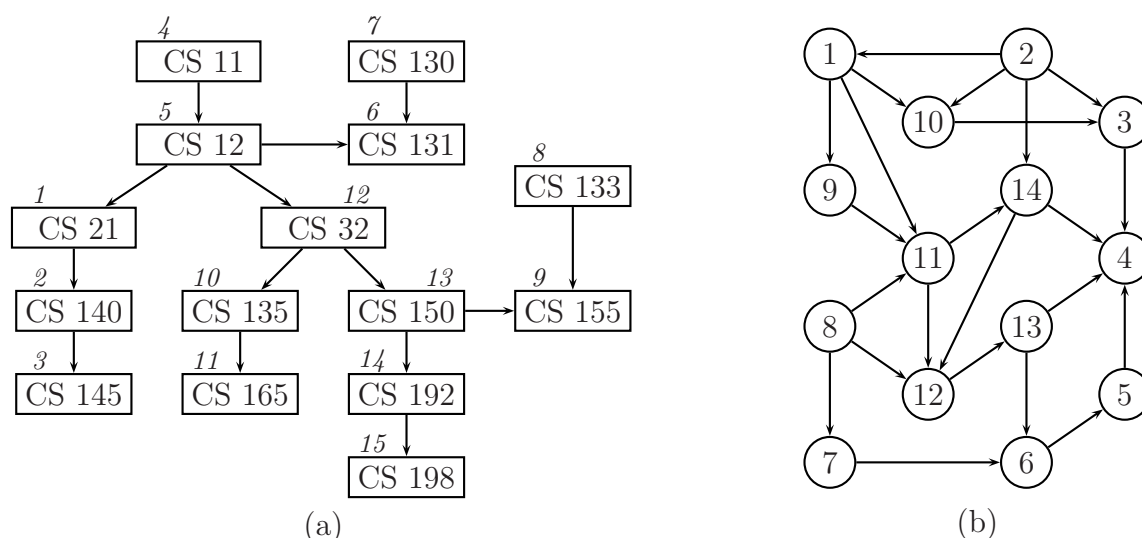


Figure 9.25 Two directed acyclic graphs

The following statement characterizes a dag:

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

To establish the truth of this statement we need to prove that:

1. If DFS yields a back edge, then there must be a cycle in G .

Proof: By definition, a back edge (i, j) connects a descendant i to an ancestor j in a depth-first tree. Since i is a descendant of j there must be a path from j to i ; therefore the edge (i, j) completes the cycle from j to j .

2. If there is a cycle in G , then DFS will yield a back edge.

Proof: Assume that there is a cycle in G and that j is the first vertex in the cycle to be discovered (which means that the rest of the vertices in the cycle are still undiscovered). Let (i, j) be the edge that precedes j in the cycle. Since there is a path from j to i consisting entirely of undiscovered vertices, i becomes a descendant of j in a depth-first tree (by Rule (9.5)). Hence (i, j) is a back edge.

The following algorithm, which is based directly on the above statement, determines whether a given directed graph G is acyclic or not.

Algorithm DAG

1. Run procedures DFS_DRIVER and DFS on G .
2. Scan the adjacency lists of G and verify whether there is a node whose *TYPE* field has been set to *B*. If there is, G is not a dag; otherwise, it is.

Alternatively, procedure DFS may be modified such that in line 9 it issues a message that G is not a dag and then returns control to the runtime system. Then Step 2 is not necessary.

9.4.2 Topological sorting

We have discussed at length the topological sorting problem in Session 5, along with an elegant algorithm from Knuth to solve the problem. (You may want to review the pertinent material at this point.) Our present concern is to consider the topological sorting problem in the context of dags and graph traversal.

A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of all the vertices in V such that if (i, j) is any edge in E , then vertex i appears before vertex j in the ordering. If we imagine the vertices to be arranged in a horizontal line, all the edges go from left to right.

Assume that we perform a DFS on a dag G , and consider the instance during the search when edge (i, j) is explored from (gray) vertex i . Now, vertex j cannot be gray because if it were gray then edge (i, j) becomes a back edge, which is not possible since G is a dag. Hence, vertex j can only be either white or black. If j is white (or undiscovered), then it becomes a descendant of i which means that j finishes before i or $f(j) < f(i)$. If j is black (or finished) then clearly $f(j) < f(i)$ also. Thus we have here a situation in which for any edge (i, j) in G we want i to appear before j in a linear order, and we find that for any such edge $f(j)$ is always less than $f(i)$. Therefore, to topologically sort a dag, we only need to perform a DFS on the dag and then arrange its vertices in their order of *decreasing* finishing times.

Given a directed acyclic graph $G = (V, E)$ on $n = |V|$ vertices and $e = |E|$ edges, the following algorithm topologically sorts the vertices of G .

Algorithm TOPOLOGICAL_SORT

1. Run procedures DFS_DRIVER and DFS on G with the following modification on procedure DFS: after vertex i is finished in line 16, push it onto a stack, say \mathbb{S} .
2. Return \mathbb{S} . (If now we pop and print the elements of \mathbb{S} until it becomes empty, we obtain a topologically sorted sequence of the vertices of G .)

Assuming that G is represented using adjacency lists, the time complexity of the algorithm is clearly that of Step 1, which is $O(n + e)$; the additional stack operations, which take $O(n)$ time, do not change the overall complexity of DFS as implemented. We note that procedure TOPOLOGICAL_SORT given in Session 5 also runs in $O(n + e)$ time, where n is the number of objects and e is the number of input relations, which correspond to the number of vertices and the number of edges, respectively, in a dag.

An example consider the dag G shown in Figure 9.25(a). Assuming that it is represented using adjacency lists as shown in Figure 9.26(a), a DFS on G using procedures DFS_DRIVER and DFS (suitably modified) yields the depth-first forest shown in Figure 9.26(b) with the discovery and finishing times indicated for each vertex. Figure 9.26(c) shows the contents of the stack \mathbb{S} when the search terminates, with vertex 8 as the top element. Note that the vertices are arranged in decreasing order of finishing times and that all edges go from left to right.

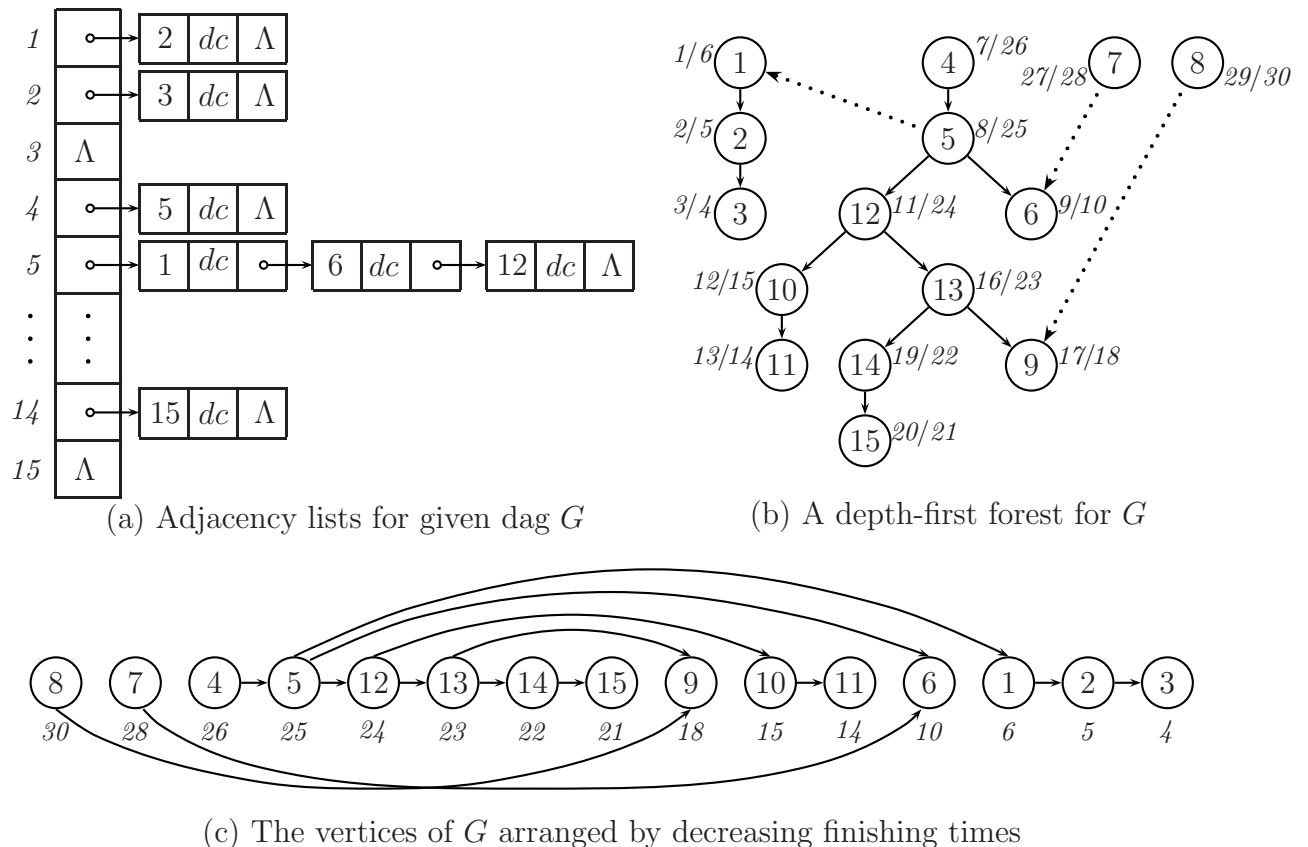


Figure 9.26 A topological sort of the directed acyclic graph of Figure 9.25(a)

9.4.3 Finding the strongly connected components of a digraph

In Section 9.1 we found that the set of vertices V of a directed graph $G = (V, E)$ can be partitioned into maximal subsets (aka equivalence classes) such that for every pair of vertices i and j in a subset, there is a path from i to j and a path from j to i . We call these subsets the *strongly connected components* of G . Any directed graph G can be decomposed into its strongly connected components; if it contains only one such component, we say that G is *strongly connected*. Figure 9.3 shows two directed graphs and their strongly connected components.

The algorithm to find the strongly connected components of a directed graph $G = (V, E)$ employs two depth-first searches, one on G , and another on its *transpose* $G^T = (V, E^R)$. The edge set E^R consists of the edges in E with the directions reversed. Figure 9.27 depicts a graph G and its transpose G^T . Note that G and G^T have the same strongly connected components, namely, $\{1,2,3,4\}$, $\{5\}$ and $\{6,7\}$; this observation applies to *any* directed graph and its transpose.

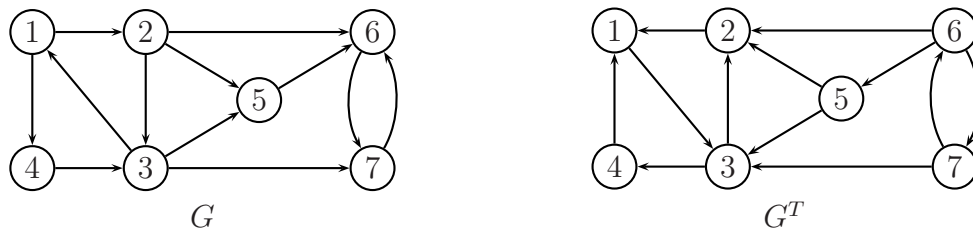


Figure 9.27 A directed graph and its transpose

The EASY procedure TRANSPOSE generates the transpose of a given directed graph G , represented by the data structure $\mathbb{G} = [LIST(1:n), (VRTX, NEXT), n]$. The transpose, denoted \mathbb{GT} is similarly represented. To indicate whose structure component is referenced in a particular EASY statement, the component is qualified by \mathbb{G} or \mathbb{GT} .

```

1  procedure TRANSPOSE( $\mathbb{G}, \mathbb{GT}$ )
2   $\mathbb{GT}.LIST \leftarrow \Lambda$ 
3  for  $i \leftarrow 1$  to  $\mathbb{G}.n$  do
4     $\alpha \leftarrow \mathbb{G}.LIST(i)$ 
5    while  $\alpha \neq \Lambda$  do
6       $j \leftarrow \mathbb{G}.VRTX(\alpha)$ 
7      call GETNODE( $\beta$ )
8       $\mathbb{GT}.VRTX(\beta) \leftarrow i$ 
9       $\mathbb{GT}.NEXT(\beta) \leftarrow \mathbb{GT}.LIST(j)$ 
10      $\mathbb{GT}.LIST(j) \leftarrow \beta$ 
11   endwhile
12 endfor
13 end TRANSPOSE

```

Procedure 9.5 Generating the transpose of a directed graph

In line 2, the adjacency lists for G^T are initialized to null. Subsequently, the adjacency list for each vertex i in G (line 3) is scanned, with the variable α marching down the list, each time pointing to the node which represents some edge (i, j) in G (lines 4–6). For each such edge, a new node is created with i stored in the *VRTX* field (lines 7–8), after which the node is inserted at the *head* of the adjacency list for vertex j in G^T (lines 9–10). Assuming that the call to GETNODE takes $O(1)$ time, it is clear that processing an edge takes constant time, or $O(e)$ time to process e edges. Initializing the n adjacency lists takes $O(n)$ time, hence the time complexity of procedure TRANSPOSE is $O(n + e)$.

With the necessary procedures in place, here now is the algorithm: Given a directed graph $G = (V, E)$, the following algorithm finds the strongly connected components of G .

Algorithm STRONGLY_CONNECTED_COMPONENTS

1. Run procedures DFS_DRIVER and DFS on G .
2. Run procedure TRANSPOSE on G to generate G^T .
3. Run procedures DFS_DRIVER and DFS on G^T with the following modification on procedure DFS_DRIVER: the loop in lines 5–7 should choose the start vertex or vertices in order of decreasing finishing times, as found in Step 1. That is, the initial start vertex, say s , is the vertex with highest finishing time; if DFS started from s terminates with some vertices still undiscovered, restart the search from the undiscovered vertex with highest finishing time. Repeat this process until all vertices are discovered.
4. Output the vertices of each tree in the depth-first forest generated in Step 3 as a separate strongly connected component.

The procedures invoked in steps 1 to 3 each take $O(n + e)$ time, as we have previously shown; step 4 clearly takes $O(n)$ time. Hence, algorithm STRONGLY_CONNECTED_COMPONENTS has time complexity $O(n + e)$, provided both G and G^T are represented using adjacency lists. A proof of the correctness of this algorithm is found in AHO[1983], p. 226 from where the algorithm is adapted; a more rigorous proof is given in CORMEN[2001], pp. 554–557.

As an example, consider finding the strongly connected components of the directed graph $G = (V, E)$ in Figure 9.27. Depending on how the adjacency lists of G are constituted and on where we initiate the search, we obtain this or that depth-first forest in Step 1 of the algorithm. Three such forests are shown in Figure 9.28, along with a

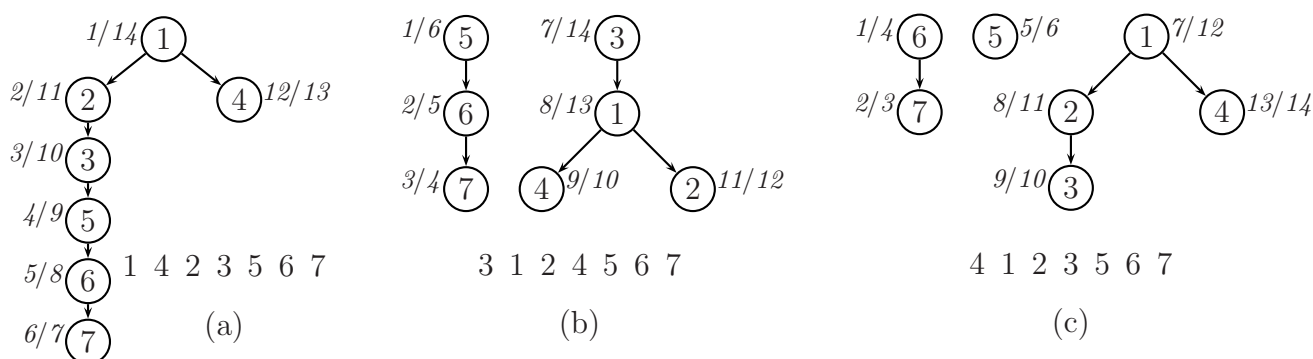


Figure 9.28 Three depth-first forests for G of Figure 9.27

listing of the vertices in their order of decreasing finishing times. If now we perform DFS on G^T , choosing the start vertex, or vertices, according to the sequences listed, we obtain the depth-first forests shown in Figure 9.29, as you may easily verify. Note that the trees which comprise each of the forests shown consists of the *same* set of vertices, viz., $\{1,2,3,4\}$, $\{5\}$ and $\{6,7\}$. These are the strongly connected components of $G = (V, E)$, as indicated in

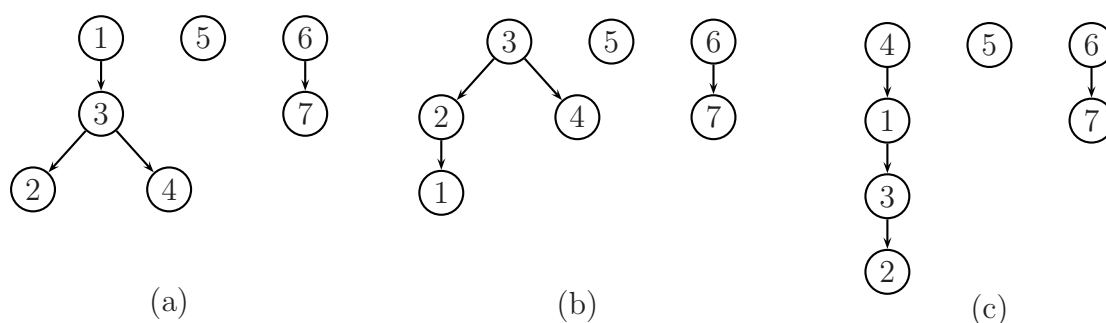


Figure 9.29 Depth-first forests for G^T of Figure 9.27

Figure 9.30(a), along with the edges in E whose endpoints are the vertices which comprise the component. Each vertex in V belongs to a strongly connected component, but some edges in E do not belong to any component. These edges are called *cross-component* edges, since they connect a vertex in one component to a vertex in another component. In Figure 9.30(a) the cross-component edges are $(2,5)$, $(2,6)$, $(3,5)$, $(3,7)$ and $(5,6)$.

If we collapse each strongly connected component into a single vertex we obtain a directed acyclic graph called the **component graph**, G^C of G , as depicted in Figure 9.30(b). An edge connects two vertices in G^C if there is a cross-component edge joining the two components represented by the two vertices. The component graph G^C is always a dag; if there were a cycle in G^C , then the components comprising the cycle should have been one strongly connected component in the first place, since every pair of vertices in a cycle are mutually reachable.

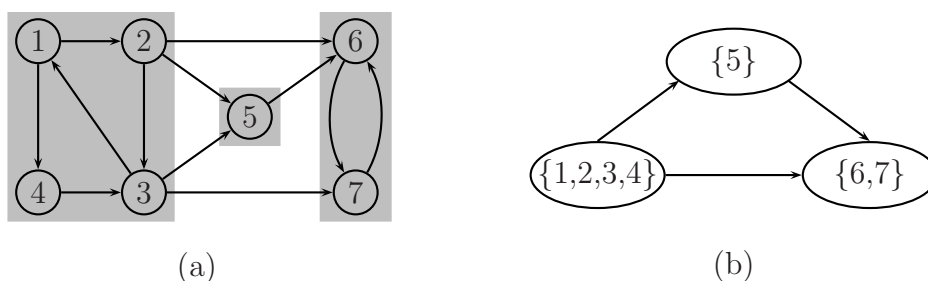


Figure 9.30 Strongly connected components of G and component graph G^C

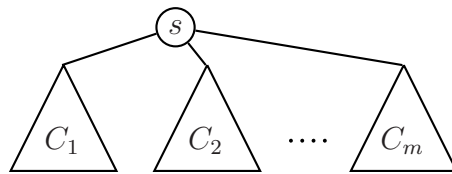
To summarize: It does not matter from which vertex or vertices we start DFS of the given directed graph G in Step 1 of algorithm STRONGLY_CONNECTED_COMPONENTS. However, in performing DFS of G^T in Step 3, we must choose the start vertex or vertices in order of decreasing finishing times of the depth-first tree found in Step 1.

9.4.4 Finding articulation points and biconnected components of a connected undirected graph

Early on in Section 9.1 we defined an *articulation point* of a connected undirected graph $G = (V, E)$ as some vertex, say i , such that removing vertex i and all the edges incident on it breaks up G into two or more pieces. If G represents a communications network where a vertex in G corresponds to a ‘site’ and an edge in G corresponds to a ‘link’ in the network, then failure at the site which corresponds to vertex i will disrupt communication among the remaining sites, i.e., there will be sites which can no longer communicate with each other. A connected undirected graph with no articulation point is said to be *biconnected*. If the network in the above example is biconnected, then failure at any single site will not disrupt communication in the rest of the network.

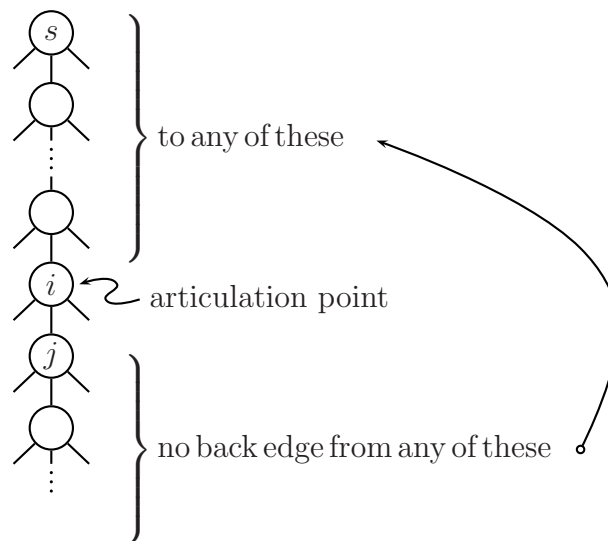
The algorithm to find the articulation points, if any, of a connected undirected graph $G = (V, E)$ is based on depth first search. Let T be a depth-first tree for G , and let vertex s be its root. We characterize the articulation points of G as follows:

1. Vertex s is an articulation point if and only if it has two or more sons.



Since there are no cross edges in a depth-first tree for an undirected graph, removing the root s clearly disconnects G into the separate components C_1, C_2, \dots, C_m .

2. Vertex $i \neq s$ is an articulation point if and only if there is a tree edge (i, j) in T for which there is no back edge (k, l) where k is a descendant of j (k may be j itself) and l is a proper ancestor of i .



Removing i clearly disconnects j and its descendants from the rest of the graph.

Figure 9.31 Identifying the articulation points of a connected undirected graph

The first case is easily handled. To test whether the root s is an articulation point we simply scan the *pred* array generated by DFS and check whether there are two or more vertices whose predecessor (father) is s .

To implement the test for the second case, we define for each vertex j a quantity, denoted $low(j)$, such that

$$low(j) = \text{minimum}[val1, val2, val3]$$

where

$$\begin{aligned} val1 &= d(j) = \text{DFS discovery time of vertex } j \\ val2 &= \text{lowest } d(l) \text{ among all back edges } (j, l) \\ val3 &= \text{lowest } low(k) \text{ among all tree edges } (j, k) \end{aligned} \tag{9.6}$$

Then, vertex $i \neq s$ is an articulation point if and only if vertex i has a son, say vertex j , such that $low(j) \geq d(i)$. As shown in Figure 9.31, the absence of a back edge from j or from any proper descendant of j to some proper ancestor of i , which is what the condition $low(j) \geq d(i)$ implies, disconnects j and all its descendants from the rest of the graph if vertex i is deleted.

It is clear from Eq.(9.3) that to find $low(j)$ for any vertex j we must first find the *low* values of all the sons of j , if any, in order to get $val3$. This is readily accomplished by traversing the depth-first tree in *postorder*, or equivalently, by processing the vertices in order of *increasing finishing times*.

Given a connected undirected graph $G = (V, E)$, the following algorithm formalizes what we described at length in the preceding paragraphs.

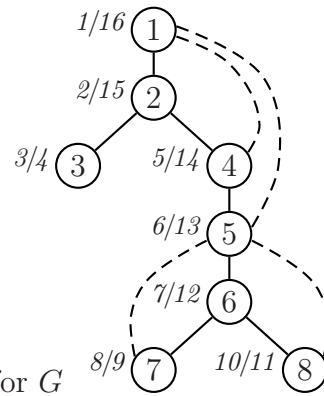
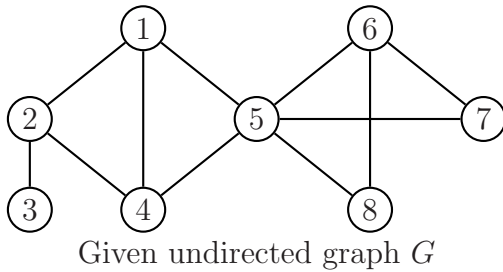
Algorithm ARTICULATION_POINTS

1. Perform a depth first search of G and record the predecessor, discovery time and finishing time of each vertex.
2. Generate the *low* array using Eq.(9.3), considering the vertices in order of increasing finishing times.
3. Find the articulation points of G , if any, as follows:
 - (a) See if there are two or more vertices whose predecessor is the root; if there are, the root is an articulation point, otherwise it is not.
 - (b) For all vertices i other than the root, see if i has a son j with $low(j)$ greater than or equal to the discovery time of i . If there is such a son, then vertex i is an articulation point; otherwise, it is not.

A recursive implementation of this algorithm can carry out *all* three steps, excluding Step 3(a), in *one* ‘pass’ through G . Since in DFS descendants finish before ancestors, computing the *low* array in postorder comes about most naturally, as we will see shortly.

For now, let us apply the algorithm ‘by hand’ to find the articulation points of graph G_1 in Figure 9.4, reproduced below as graph G . A depth-first tree for G , with vertex 1 as root, is shown along with the discovery and finishing times of each vertex. Computing the *low* values in order of increasing finishing times of the vertices, we obtain:

$$\begin{aligned}
 low(3) &= \min[d(3), na, na] = \min[3, na, na] = 3 \\
 low(7) &= \min[d(7), d(5), na] = \min[8, 6, na] = 6 \\
 low(8) &= \min[d(8), d(5), na] = \min[10, 6, na] = 6 \\
 low(6) &= \min[d(6), na, low(7)] = \min[7, na, 6] = 6 \\
 low(5) &= \min[d(5), d(1), low(6)] = \min[6, 1, 6] = 1 \\
 low(4) &= \min[d(4), d(1), low(5)] = \min[5, 1, 6] = 1 \\
 low(2) &= \min[d(2), na, low(4)] = \min[2, na, 1] = 1 \\
 low(1) &= \min[d(1), na, low(2)] = \min[1, na, 1] = 1
 \end{aligned}$$



In these computations ‘ na ’ means ‘not applicable’, either because there is no back edge (j, l) or there is no tree edge (j, k) . Finally we find the articulation points of G , if any.

Vertex 1: Vertex 1 (the root) has only one son; it is not an articulation point.

Vertex 2: $low(3) = 3 > d(2) = 2$; vertex 2 is an articulation point.

Vertex 3: Vertex 3 has no son; it cannot be an articulation point.

Vertex 4: $low(5) = 1 < d(4) = 5$; vertex 4 is not an articulation point.

Vertex 5: $low(6) = 6 = d(5) = 6$; vertex 5 is an articulation point.

Vertex 6: $low(7) = 6 < d(6) = 7$; vertex 6 is not an articulation point.

Vertex 7: Vertex 7 has no son; it cannot be an articulation point.

Vertex 8: Vertex 8 has no son; it cannot be an articulation point.

Hence the articulation points of the given graph G are vertices 2 and 5, as you may easily verify by inspection.

Since G has articulation points, it is *not* biconnected. Rather, it can be decomposed into its **biconnected components**. Figure 9.32 shows the biconnected components of G , viz., BC_1 , BC_2 and BC_3 .

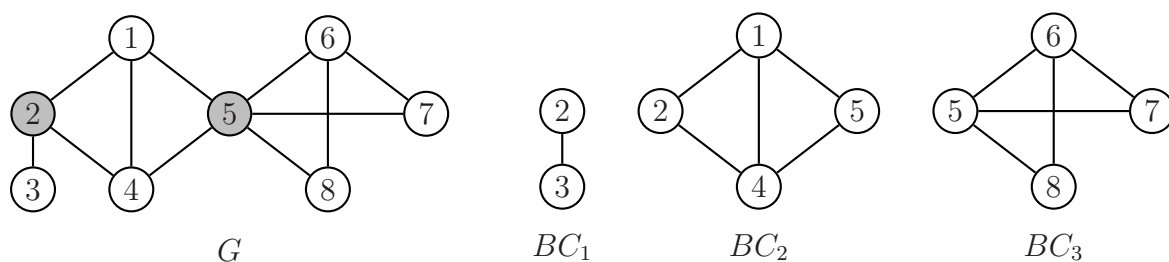


Figure 9.32 The articulation points and biconnected components of G

We take note of the following properties of these biconnected components:

1. They have no articulation points.
2. There is *at most* one vertex that is common to any two biconnected components, and this vertex is an articulation point of G .

These properties are true of the biconnected components of *any* connected, undirected graph.

Procedure ARTICULATION_POINTS(\mathbb{G}, i) finds the articulation points, if any, of a connected undirected graph $G = (V, E)$ using depth first search as the underlying template. The graph G is represented by the data structure $\mathbb{G} = [LIST(1:n), (VRTX, NEXT), pred(1:n), d(1:n), low(1:n), tag(1:n)]$. The first two components of \mathbb{G} comprise the adjacency lists representation of G . The remaining four components of \mathbb{G} may be viewed as ‘attributes’ of G discovered during the search. Specifically, $tag(i) = 1$ means vertex i is an articulation point; 0 means it is not. The procedure uses the function MIN which returns the minimum of its arguments.

```

1  procedure ARTICULATION_POINTS( $\mathbb{G}, i$ )
2     $color(i) \leftarrow gray$ 
3     $d(i) \leftarrow time \leftarrow time + 1$ 
4     $low(i) \leftarrow d(i) \quad \triangleright val1$ 
5     $\alpha \leftarrow LIST(i)$ 
6    while  $\alpha \neq \Lambda$  do
7       $j \leftarrow VRTX(\alpha)$ 
8      case
9        :  $color(j) = white$  : [  $pred(j) \leftarrow i$ ; call ARTICULATION_POINTS( $\mathbb{G}, j$ )
10                           if  $low(j) \geq d(i)$  then  $tag(i) \leftarrow 1$ 
11                            $low(i) \leftarrow MIN(low(i), low(j))$  ]  $\triangleright val3$ 
12        :  $color(j) = gray$  : if  $pred(i) \neq j$  then  $low(i) \leftarrow MIN(low(i), d(j))$   $\triangleright val2$ 
13      endcase
14       $\alpha \leftarrow NEXT(\alpha)$ 
15    endwhile
16  end ARTICULATION_POINTS

```

Procedure 9.6 Finding the articulation points of a connected undirected graph

The search is initiated from the undiscovered (white) vertex i (line 1). In line 2, vertex i is painted gray to indicate that it is now discovered; in line 3, the global variable $time$ is updated and is assigned as the discovery time of vertex i , which is then assigned as the initial value of $low(i)$ in line 4. Lines 5 to 15 examine the adjacency list of vertex i . For each vertex j adjacent to vertex i , one of two cases may obtain: (a) j is undiscovered (white) or (b) j is discovered and unfinished (gray). In the first case, j becomes a son of i (line 9), after which the search continues from j (the most recently discovered vertex) by recursively calling `ARTICULATION_POINTS`. Upon return from the call, the final value of $low(j)$ would have been computed so that it is now possible to test if vertex i (j 's father) is an articulation point (line 10). Subsequently, the current value of $low(i)$ is updated in line 11 if it turns out to be larger than that of its son j ; in turn, this value will be used later in determining whether i 's father is an articulation point.

The test in line 12, which handles case (b), is necessary since edge (i, j) may have been explored earlier as (j, i) , indicated by the condition $pred(i) = j$, in which case nothing else needs to be done about i or j (no op). Otherwise, (i, j) must be a back edge, in which case the current value of $low(i)$ is updated if it turns out to be larger than j 's discovery time.

Given below is a sample calling procedure, `FIND_ARTICULATION_POINTS(\mathbb{G})`, which initializes the global variable $time$ to 0, the global array $color$ to *white*, the $pred$ array to 0 and the tag array to 0 (lines 2–5). Since G is a connected undirected graph, the call to `ARTICULATION_POINTS` in line 6 with 1 as the start vertex (actually any vertex will do) traverses the entire graph and classifies all the vertices, except the root, as either an articulation point or not. Finally, lines 7–12 determine whether the root is an articulation point. Line 12 is necessary because the call to `ARTICULATION_POINTS($\mathbb{G}, 1$)` in line 6 always returns with $tag(1) = 1$ (because $low(1) = d(1)$).

```

1  procedure FIND_ARTICULATION_POINTS( $\mathbb{G}$ )
2     $time \leftarrow 0$             $\triangleright$  global variable
3     $color \leftarrow white$      $\triangleright$  global array
4     $pred \leftarrow 0$ 
5     $tag \leftarrow 0$ 
6    call ARTICULATION_POINTS( $\mathbb{G}, 1$ )
7     $m \leftarrow 0$             $\triangleright$  no. of subtrees of the root
8    for  $i \leftarrow 2$  to  $n$  do
9      if  $pred(i) = 1$  then  $m \leftarrow m + 1$ 
10   endfor
11   if  $m \geq 2$  then  $tag(1) \leftarrow 1$ 
12     else  $tag(1) \leftarrow 0$ 
13   end FIND_ARTICULATION_POINTS

```

Procedure 9.7 Sample calling procedure for procedure `ARTICULATION_POINTS`

The time complexity of the algorithm to find the articulation points of a connected undirected graph $G = (V, E)$ on $n = |V|$ vertices and $e = |E|$ edges, as implemented in Procedure 9.6 and 9.7, is $O(n + e)$, which is that of depth first search on which the algorithm is directly based. (See Procedures 9.1 and 9.2.)

9.4.5 Determining whether an undirected graph is acyclic

Given a pictorial representation of an undirected graph we can immediately see whether it contains cycles or not. The task becomes somewhat less trivial if we are presented instead with its adjacency lists or adjacency matrix representation, as a computer would be. For instance, is the undirected graph G shown below acyclic?

$$G = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

A connected undirected graph with no cycles is called a **free tree**; an example is shown in Figure 9.33(a). Two important, and readily verifiable, properties of a free tree are:

1. A free tree on n vertices contains exactly $n - 1$ edges.
2. If an edge is added to a free tree, it becomes a graph with a cycle.

Properties 1 and 2 imply that any undirected graph with n vertices and $e \geq n$ edges must contain a cycle; an example is shown in Figure 9.33(b). However it does not follow that a graph with less than n edges is acyclic. An undirected graph which is not connected may have fewer than n edges and still have a cycle; an example is shown in Figure 9.33(c).

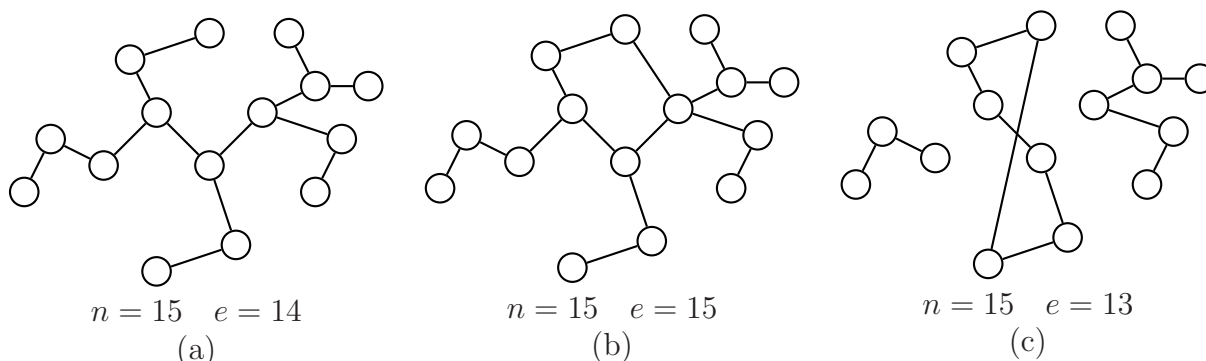


Figure 9.33 Cycles in undirected graphs

The foregoing observations imply that we cannot determine whether an undirected graph is acyclic by counting vertices and edges, unless we have prior information that the graph is connected. To solve this problem we turn once more to graph traversal, this time *breadth first search*. The following statement characterizes an acyclic undirected graph:

An undirected graph G contains no cycles if and only if a breadth first search of G yields no cross edges.

Earlier we saw that there are only two types of edges in a breadth-first forest of an undirected graph, viz., tree edges and cross edges. A forest consisting only of tree edges obviously cannot contain cycles; it is the presence of cross edges that indicate the presence of cycles. Consider, for instance, Figure 9.22 from Section 9.3.2, reproduced below for quick reference. Note that every cross edge, say (i, j) , forms a simple cycle with tree edges leading to vertices i and j from their closest common ancestor, which is their father if i and j are brothers, their grandfather if i and j are cousins, or i 's father and j 's grandfather if i and j are uncle-nephew.

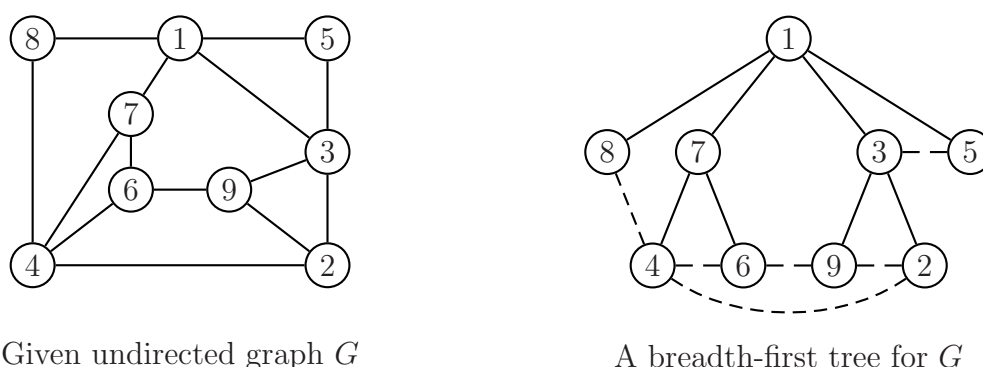


Figure 9.34 Cross edges indicate cycles in undirected graphs

Recall that in breadth first search of an undirected graph, edge (i, j) becomes a cross edge if vertex j is already gray (discovered but unfinished) when edge (i, j) is explored from vertex i . Thus we can use Procedures 9.3 and 9.4 of Section 9.3.2 to test whether an undirected graph G is acyclic by simply adding line 12a to procedure $\text{BFS}(\mathbb{G}, s)$, as indicated below:

```

11      if color(j) = white then [ color(j) ← gray; pred(j) ← i
12                                l(j) ← l(i) + 1; call ENQUEUE(Q, j) ]
12a      else [ output 'G contains a cycle'; stop ]

```

Since for an undirected graph on n vertices it takes no more than n edges to form a cycle, the above procedures, as modified, terminate in $O(n)$ time.

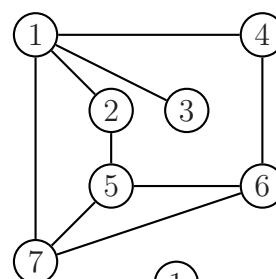
Summary

- More than any of the ADT's which abound in the CS literature, graphs are used to model a large number of real-life problems. Elegant algorithms which run in polynomial time have been devised to solve some of these problems; a large number, however, have turned out to be intractable with no efficient algorithms known to date for their solution.

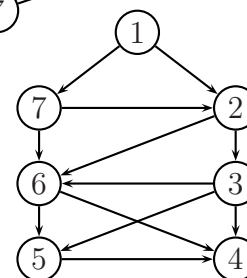
- There is a great deal of terminology associated with graphs. A vital key to understanding algorithms on graphs is a clear grasp of this terminology.
- There are basically two ways of representing graphs in computer memory — adjacency matrix representation and adjacency lists representation. While the former is easier to work with, graphs encountered in practice are typically sparse making the latter the more suitable representation to use. All of the EASY procedures in this session take as input a graph represented using adjacency lists.
- To traverse a graph is to *explore its edges to discover its vertices*. The two basic traversal algorithms on graphs are *depth first search* (DFS) and *breadth first search* (BFS). DFS performs the search along a narrow front (akin to a person trying to find his way out of a maze); BFS performs the search over a wide front (akin to a search party spreading out in search of something).
- Some classical applications of DFS and BFS are: identifying dags, topological sorting, finding the strongly connected components of a directed graph, finding the articulation points and biconnected components of an undirected graph, and determining whether an undirected graph is acyclic.

Exercises

1. There are $n \geq 2$ people at a party and each one shakes hands with every one else. What is the total number of handshakes?
2. Show how to arrange 22 rooms, where each room has at most 3 doors, such that it is possible to go from one room to any other room by passing through at most 6 doors.
3. Extend the problem in Item 2 to 32 rooms.
4. Show the
 - (a) adjacency matrix
 - (b) adjacency lists
 representation of the undirected graph shown.



5. Show the
 - (a) adjacency matrix
 - (b) adjacency lists
 representation of the directed graph shown.



6. Construct a depth-first forest for the undirected graph in Item 4. Indicate the discovery time and finishing time of each vertex and the type of each edge as in Figure 9.20.

7. Construct a depth-first forest for the directed graph in Item 5. Indicate the discovery time and finishing time of each vertex and the type of each edge as in Figure 9.20.
8. Construct a breadth-first forest for the undirected graph in Item 4. Indicate the level of each vertex and the type of each edge in the graph as in Figure 9.22.
9. Construct a breadth-first forest for the directed graph in Item 5. Indicate the level of each vertex and the type of each edge as in Figure 9.22.
10. Rewrite the EASY procedure DFS (Procedure 9.1) for a graph represented by its adjacency matrix. What is the time complexity of the procedure?
11. Rewrite the EASY procedure BFS (Procedure 9.3) for a graph represented by its adjacency matrix. What is the time complexity of the procedure?
12. Using a language of your choice, transcribe procedure DFS and DFS_DRIVER (Procedure 9.1 and 9.2) into a running program. Test your program using the graphs in Examples 9.1 and 9.2.
13. Using a language of your choice, transcribe procedure BFS and BFS_DRIVER (Procedure 9.3 and 9.4) into a running program. Test your program using the graphs in Examples 9.1 and 9.2.
14. Perform a topological sort of the directed acyclic graph in Figure 9.25(b).
15. Construct a
 - (a) directed graph on 12 vertices which contains exactly two strongly connected components
 - (b) a strongly connected graph on 12 vertices
16. Using a language of your choice, implement Algorithm STRONGLY_CONNECTED_COMPONENTS into a running program. Test your program using the graphs in Item 15.
17. Construct
 - (a) an undirected graph on 12 vertices which contains exactly two articulation points
 - (b) a biconnected graph on 12 vertices
18. Apply Algorithm ARTICULATION_POINTS to your graphs in Item 17.
19. Using a language of your choice, transcribe procedures ARTICULATION_POINTS (Procedure 9.6) and FIND_ARTICULATION_POINTS (Procedure 9.7) into a running program. Test your program using the graphs in Item 17.

Bibliographic Notes

CORMEN[2001] and STANDISH[1994] are our primary sources for the material in sections 9.1 and 9.2 on graph terminology and graph representation. The two very important algorithms for traversing graphs, DFS and BFS, are discussed in most textbooks on Data Structures or on Algorithms with varying levels of rigor. We have adopted the more comprehensive and rigorous treatment of these algorithms given in CORMEN[2001], pp. 531–547. The proofs of the correctness of Algorithm DAG and Algorithm TOPOLOGICAL_SORT are given as Lemma 22.11 and Theorem 22.12 in the same reference, pp. 550–551.

Algorithm STRONGLY_CONNECTED_COMPONENTS is from AHO[1983], pp. 224–226, where a correctness proof is also given. CORMEN[2001], pp. 554–557, ‘unravels the mystery’ of this algorithm and gives a more rigorous proof of its correctness.

Algorithm ARTICULATION_POINTS is from AHO[1983], pp. 245–246; a pseudocode implementation of the algorithm is found in WEISS[1997], pp. 322–327.

Finding cycles in undirected graphs using BFS is discussed in AHO[1983], p. 244.

SESSION 10

Applications of graphs

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Explain and illustrate the minimum spanning tree theorem.
2. Apply Prim's and Kruskal's algorithms to find a minimum-cost spanning tree for a given connected, weighted undirected graph.
3. Discuss the issues involved in implementing Prim's and Kruskal's algorithms on a computer.
4. Define the SSSP and APSP problems.
5. Apply Dijkstra's algorithm to solve the SSSP problem and Floyd's algorithm to solve the APSP problem.
6. Discuss the issues involved in implementing Dijkstra's and Floyd's algorithms on a computer.
7. Define the transitive closure of a graph.
8. Apply Warshall's algorithm to find the transitive closure of a graph.

READINGS CORMEN[2001], pp. 561–587, 595–599, 629–634; AHO[1983], pp. 203–213, 233–240.

DISCUSSION

In this session we will examine a number of important graph problems which arise in modeling and solving certain real-life problems, such as minimizing the cost of linking the various nodes of a communications network or finding the cheapest way to go from one city to another. These are the problems of:

1. finding a minimum-cost spanning tree for a connected, weighted, undirected graph
2. finding shortest paths in a weighted directed graph

We are interested in the algorithms to solve these problems not only for their practical usefulness, but more for the lesson to be derived from them as prime examples of two important algorithm design techniques, viz., the greedy technique and dynamic programming. The beauty of these algorithms lie in their brevity and simplicity; the challenge is implementing them efficiently on a computer.

10.1 Minimum-cost spanning trees for undirected graphs

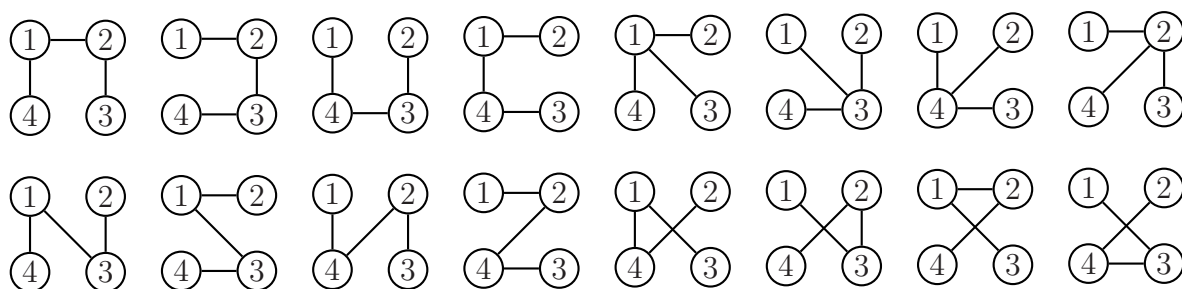
In the previous session, we have seen how depth first search or breadth first search initiated from any vertex in a connected undirected graph G can be used to generate a spanning tree for G . For the case in which *costs* or *weights* are assigned to the edges of the graph, we define the cost of a spanning tree as the sum of the costs of the edges, or branches, in the tree. An important problem on such connected, weighted, undirected graphs is finding a spanning tree of minimum cost.

For example, if we think of a communication network as a weighted graph in which vertices are nodes in the network and edges are communication links, then a minimum-cost spanning tree for the graph represents a network connecting all nodes at minimum cost.

The number of spanning trees which can be constructed for a given graph is rather large. Specifically, the number of spanning trees for a *complete* graph on n vertices is n^{n-2} . This result follows from the following theorem:

Cayley's theorem: The number of spanning trees on n distinct vertices is n^{n-2} .

Thus, for a complete graph on four vertices, the number of spanning trees is 16, as shown below; for a complete graph on ten vertices, it is 100 million!



Even for a graph that is not complete, it is reasonable to expect that the number of spanning trees is still quite large. It turns out that we need not construct all the spanning trees for the graph and compute the cost of each in order to obtain one with minimum cost. There are a number of elegant and efficient algorithms for generating a minimum-cost spanning tree for a weighted undirected graph. Among these, the two more popular are **Prim's algorithm** and **Kruskal's algorithm**.

Prim's and Kruskal's algorithms are classical applications of an algorithm design technique called the **greedy method**, which is used to solve certain optimization problems.

Following Standish, we define a greedy algorithm as ‘an algorithm in which a sequence of locally opportunistic choices succeeds in finding a global optimum’ (STANDISH[1980], p. 96). In the present case, the ‘global optimum’ is a spanning tree whose cost is minimum.

Both algorithms are based on the following theorem:

MST theorem: Let $G = (V, E)$ be a connected, weighted, undirected graph. Let U be some proper subset of V and (i, j) be an edge of least cost such that $i \in U$ and $j \in V - U$. There exists a minimum cost spanning tree T such that (i, j) is an edge in T .

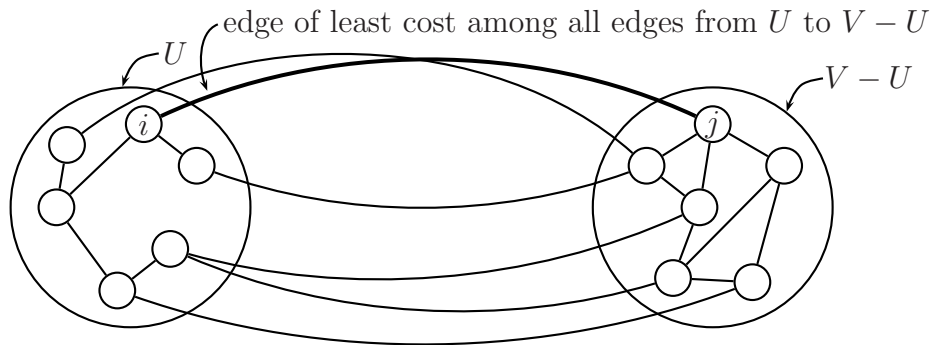


Figure 10.1 Edge (i, j) is an edge in a minimum-cost spanning tree for G

Proof: Suppose T' is a minimum-cost spanning tree for G and edge (i, j) is not in T' . Now add (i, j) to T' . Clearly a cycle is formed in T' with (i, j) as one of the edges in the cycle. Likewise, there must be some edge, say (k, l) , with $k \in U$ and $l \in V - U$ in the resulting cycle (see Figure 10.2). Since edge (i, j) is an edge of least cost among those edges with one vertex in U and the other vertex in $V - U$, then the cost of $(i, j) \leq$ cost of (k, l) . Hence, removing edge (k, l) from $T' + (i, j)$ yields a spanning tree whose cost cannot be more than the cost of T' . This is the minimum cost spanning tree T which includes the edge (i, j) .

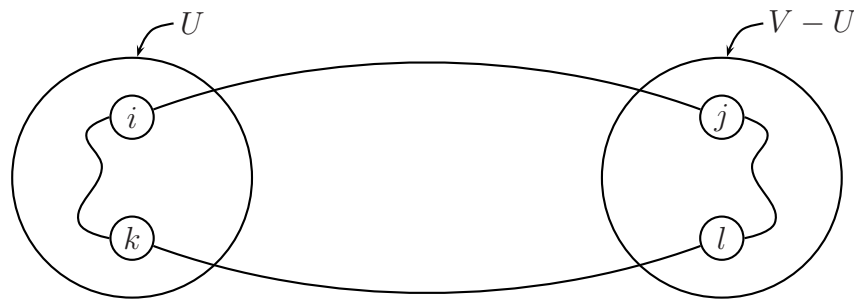


Figure 10.2 The cycle formed with the addition of edge (i, j)

10.1.1 Prim's algorithm

Let $G = (V, E)$ be a connected, weighted, undirected graph. Prim's algorithm generates a minimum cost spanning tree for G by initially choosing one vertex, any vertex, in V . It then ‘grows’ the tree by adding one vertex, and the edge which connects the vertex to the tree, at a time.

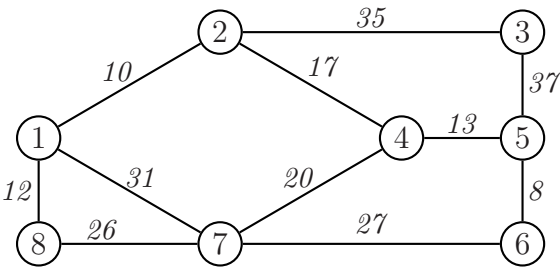
Let U denote the set of vertices already chosen and T denote the tree formed by these vertices and the edges which connect them. Initially, U and T are both empty. Prim's algorithm may be stated as follows:

Prim's algorithm

1. [Start vertex] Choose any vertex in V and place it in U .
2. [Next vertex] From among the vertices in $V - U$ choose that vertex, say j , which is connected to some vertex, say i , in U by an edge of least cost. Add vertex j to U and edge (i, j) to T .
3. [All vertices considered?] Repeat Step 2 until $U = V$. Then, T is a minimum-cost spanning tree for G .

We see from this description of Prim's algorithm that it is a direct and straightforward application of the MST theorem. To see how the algorithm actually works, and to assess which steps are crucial in implementing the algorithm on a computer, consider the following example.

Example 10.1. Prim's algorithm to find a minimum cost spanning tree



Stage	T (vertices and edges in solid black)	U	$V - U$	Edges from $V - U$ to U
1		1	2, 3, 4, 5, 6, 7, 8	(2,1) – 10 (7,1) – 31 (8,1) – 12
2		1, 2	3, 4, 5, 6, 7, 8	(3,2) – 35 (4,2) – 17 (7,1) – 31 (8,1) – 12

Stage	T (vertices and edges in solid black)	U	$V - U$	Edges from $V - U$ to U
3		1, 2, 8	3, 4, 5, 6, 7	<div>(3,2) – 35</div> <div>(4,2) – 17</div> <div>(7,1) – 31</div> <div>(7,8) – 26</div>
4		1, 2, 4, 8	3, 5, 6, 7	<div>(3,2) – 35</div> <div>(5,4) – 13</div> <div>(7,1) – 31</div> <div>(7,4) – 20</div> <div>(7,8) – 26</div>
5		1, 2, 4, 5, 8	3, 6, 7	<div>(3,2) – 35</div> <div>(3,5) – 37</div> <div>(6,5) – 8</div> <div>(7,1) – 31</div> <div>(7,4) – 20</div> <div>(7,8) – 26</div>
6		1, 2, 4, 5, 6, 8	3, 7	<div>(3,2) – 35</div> <div>(3,5) – 37</div> <div>(7,1) – 31</div> <div>(7,4) – 20</div> <div>(7,6) – 27</div> <div>(7,8) – 26</div>
7		1, 2, 4, 5, 6, 7, 8	3	<div>(3,2) – 35</div> <div>(3,5) – 37</div>

Figure 10.3 Prim's algorithm in action (continued on next page)

Stage	T (vertices and edges in solid black)	U	$V - U$	Edges from $V - U$ to U
8	<p style="text-align: center;">Cost = 115</p>	1, 2, 3, 4, 5, 6, 7, 8		

Figure 10.3 Prim's algorithm in action

We take note of the following observations pertaining to Example 10.1 of Figure 10.3.

1. The vertices and edges shown in solid black in the second column depict the minimum-cost spanning tree T as it grows, one vertex and edge, at a time. Note that the tree consists of one connected component at every stage of its growth.
2. The black vertices comprise the set U ; the gray and white vertices comprise the set $V - U$.
3. The gray vertices are the *fringe* vertices. They are connected to vertices in the tree by edges shown as dashed lines; these are the edges which span the sets $V - U$ and U in Figure 10.1. These edges are listed in the fifth column in Figure 10.3 along with their associated costs. An edge is shaded if it is an edge which connects a fringe vertex to its *nearest neighbor* in the tree. For instance, in stage 5 of Prim's algorithm, vertex 7 is connected to the tree by edges (7,1), (7,4) and (7,8); of the three, edge (7,4) has the smallest cost (vertex 4 is vertex 7's nearest neighbor) and is shaded. In particular, the edge shaded black is the edge with the smallest cost among all the shaded edges; this is the *edge of least cost* alluded to in Figure 10.1 which, according to the MST theorem, *is* an edge in a minimum-cost spanning tree for G . In stage 5 of Prim's algorithm, this is edge (6,5); in stage 6, it becomes a part of T .

We can think of Prim's algorithm as an application of breadth first search, as described in Session 9, with one simple but important modification: our container for gray, or fringe, vertices is now a *priority queue* instead of an ordinary first-in-first-out queue. We assign to each vertex i in the container a priority, which is the cost of the edge which connects i to its nearest neighbor in the tree; the vertex next discovered is the fringe vertex with highest priority, or equivalently, the fringe vertex closest to the tree. For instance, in stage 5 of Prim's algorithm in Figure 10.3, the fringe vertices in our container (priority queue) are vertices 3 (priority = 35), 6 (priority = 8) and 7 (priority = 20); of these, vertex 6 has highest priority and it is the vertex that the algorithm chooses next.

The following rule operationalizes Step 2 of Prim's algorithm:

Prim's rule: Choose the vertex on the fringe that is closest to the tree.

Program implementation of Prim's algorithm

It is clear from the above example and accompanying commentary that the crucial task in implementing Prim's algorithm on a computer is *finding the fringe vertex that is closest to the tree* in Step 2 of the algorithm. To this end, let $key(l)$ denote the priority of vertex l and consider the vertex with smallest key -value to have the highest priority. Initially each vertex in V has a priority of ∞ , except the start vertex, say s , which has a priority of $-\infty$ and which is assumed to be already in U . Further, let $pred(l)$ be j if edge (l, j) is an edge in T . With these definitions we implement Step 2 of Prim's algorithm as follows:

- 2a. Let j be the vertex most recently placed in U .
- 2b. Adjust the priority of each vertex l in $V - U$ as follows:
 - i. If vertex l is not adjacent to vertex j , retain the priority of vertex l .
 - ii. If vertex l is adjacent to vertex j and $key(l)$ is greater than the cost of edge (l, j) then set

$$key(l) = \text{cost of edge } (l, j)$$

$$pred(l) = j$$
 Otherwise, retain the current value of $key(l)$ and $pred(l)$.
- 2c. Choose the vertex in $V - U$ with highest priority (smallest key -value) and place it in U .

If now we repeat Step 2 until $U = V$, we obtain a minimum-cost spanning tree T , encoded in the array $pred$, for the connected, weighted, undirected graph $G = (V, E)$.

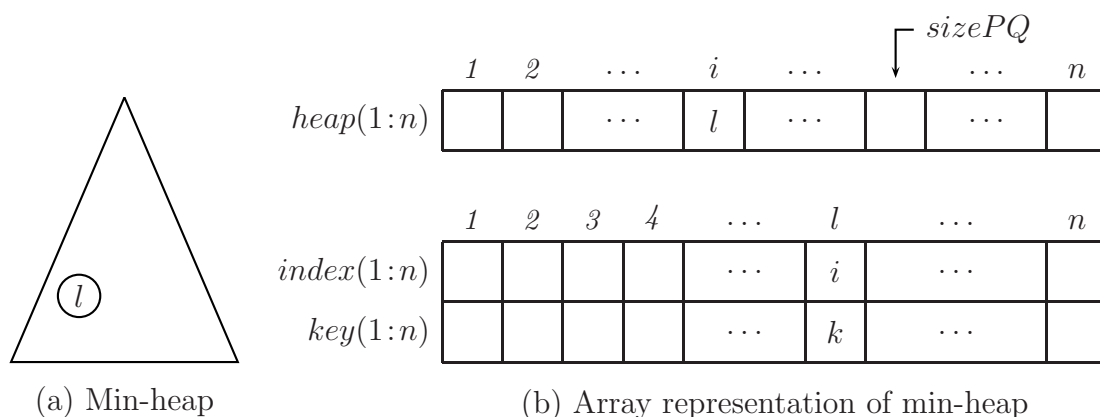
Early on in section 7.3 of Session 7, we enumerated some of the operations supported by a priority queue; two of these are:

1. *extract* (i.e., delete and return) the element with highest priority, and
2. *change* the priority of a specified element in the priority queue

These are precisely the operations we need in order to carry out Steps 2b.ii and 2c of Prim's algorithm. In order to realize both of these operations in an efficient way we will implement the priority queue using a *min-heap* represented using three parallel arrays. Specifically, we use the data structure

$$PQ = [heap(1:n), index(1:n), key(1:n), sizePQ]$$

as depicted in Figure 10.4.



n = number of vertices in G

$sizePQ$ = current size of the min-heap (or equivalently, of the priority queue)
 = n initially

l = label of a vertex in the min-heap, $1 \leq l \leq n$

i = index of the cell in array $heap$ which contains vertex l , $1 \leq i \leq sizePQ$

$k = key(l) = key(heap(i))$ = distance from l to its nearest neighbor in T

Figure 10.4 Min-heap implementation of priority queue for Prim's algorithm

We restore the heap-order property after an *extract* or *change* operation by requiring that

$$key(heap(i)) \leq \text{minimum}[key(heap(2*i)), key(heap(2*i+1))]$$

for all $1 \leq i \leq \lfloor sizePQ/2 \rfloor$, $2*i+1 \leq sizePQ$. Note that a *change* operation on a vertex, say l , *decreases* its key. This means that the subtree rooted at l remains a min-heap; however, the subtree rooted at l 's father may no longer be a min-heap. In order to restore the heap-order property we need to find the cell, say i , in the array $heap$ where l is stored. The array $index$ allows us to do this in *constant time*, since i is simply $index(l)$; if l is not the root of the heap, then l 's father is in cell $\lfloor i/2 \rfloor$.

EASY procedures to implement Prim's algorithm

The EASY procedure $PRIM(\mathbb{G}, s)$ implements Prim's algorithm as described above. The input to the procedure is a connected, weighted, undirected graph $G = (V, E)$ and a start vertex s . The graph G is represented by the data structure $\mathbb{G} = [LIST(1:n), (VRTX, COST, NEXT), pred(1:n), n]$. The first two components of \mathbb{G} comprise the cost adjacency lists representation of G . The minimum-cost spanning tree found by Prim is returned in the $pred$ array; this comprise the output.

Procedure PRIM invokes procedure InitPQ to initialize the priority queue which contains the vertices in $V - U$. Since initially every vertex is assigned a priority of ∞ except for the start vertex s which is assigned a priority of $-\infty$, any assignment of the vertices to the nodes of the min-heap is valid, provided that s is the root node.

At any stage in the computations, the fringe vertex closest to the tree is the vertex at the root of the min-heap; this is moved from $V - U$ to U by a call to procedure EXTRACT_MIN, which deletes and returns the vertex at the root of the heap and restores

the heap-order property via a call to procedure HEAPIFY. Procedures EXTRACT_MIN and HEAPIFY are suitably modified versions of the generic procedures PQ_EXTRACT and HEAPIFY given in Session 7.

After a vertex, say j , is moved to U , some vertex, say l , in $V - U$ that is adjacent to j may become closer to the tree via the edge (l, j) , requiring a change in its priority. This is accomplished by the call to procedure DECREASE_KEY, which adjusts the priority of vertex l and then restores the heap-order property.

Study the procedures carefully. A review of the material on heaps and priority queues in Session 7 would be particularly helpful.

```

1  procedure PRIM( $\mathbb{G}, s$ )
2  call InitPQ( $\mathbb{G}, \mathbb{PQ}, s$ )
3   $\mathbb{G}.pred(s) \leftarrow 0$ 
▷ Generate minimum-cost spanning tree
4  while not IsEmptyPQ( $\mathbb{PQ}$ ) do
5    call EXTRACT_MIN( $\mathbb{PQ}, j$ )
6     $\mathbb{PQ}.key(j) \leftarrow -\infty$ 
7     $\alpha \leftarrow \mathbb{G}.LIST(j)$ 
8    while  $\alpha \neq \Lambda$  do
9       $l \leftarrow \mathbb{G}.VRTX(\alpha)$ 
10     if  $\mathbb{G}.COST(\alpha) < \mathbb{PQ}.key(l)$  then [  $\mathbb{G}.pred(l) \leftarrow j$ 
11                                           call DECREASE_KEY( $\mathbb{PQ}, l, \mathbb{G}.COST(\alpha)$ ) ]
12      $\alpha \leftarrow \mathbb{G}.NEXT(\alpha)$ 
13   endwhile
14 endwhile
15 end PRIM

```

```

1  procedure InitPQ( $\mathbb{G}, \mathbb{PQ}, s$ )
2   $i \leftarrow 1$ 
3  for  $l \leftarrow 1$  to  $\mathbb{G}.n$  do
4    if  $l = s$  then [  $\mathbb{PQ}.heap(1) \leftarrow s$ ;  $\mathbb{PQ}.index(s) \leftarrow 1$ ;  $\mathbb{PQ}.key(s) \leftarrow -\infty$  ]
5    else [  $i \leftarrow i + 1$ ;  $\mathbb{PQ}.heap(i) \leftarrow l$ ;  $\mathbb{PQ}.index(l) \leftarrow i$ ;  $\mathbb{PQ}.key(l) \leftarrow \infty$  ]
6  endfor
7   $\mathbb{PQ}.sizePQ \leftarrow \mathbb{G}.n$ 
8  end InitPQ

```

```

1  procedure IsEmptyPQ( $\mathbb{PQ}$ )
2  return( $\mathbb{PQ}.sizePQ = 0$ )
3  end IsEmptyPQ

```

Procedure 10.1 Implementing Prim's algorithm using a min-heap
(continued on next page)

```

1  procedure EXTRACT_MIN(PQ, j)
2  if PQ.sizePQ = 0 then call PQ_UNDERFLOW
3      else [ j ← PQ.heap(1); PQ.heap(1) ← PQ.heap(PQ.sizePQ)
4              PQ.index(PQ.heap(1)) ← 1
5              PQ.sizePQ ← PQ.sizePQ - 1
6              call HEAPIFY(PQ, 1) ]
7  end EXTRACT_MIN

1  procedure HEAPIFY(PQ, r)
2  k ← PQ.key(PQ.heap(r))
3  l ← PQ.heap(r)
4  i ← r; j ← 2 * i
5  while j ≤ PQ.sizePQ do
6      if j < PQ.sizePQ and PQ.key(PQ.heap(j + 1)) < PQ.key(PQ.heap(j)) then j ← j + 1
7      if PQ.key(PQ.heap(j)) < k then [ PQ.heap(i) ← PQ.heap(j)
8              PQ.index(PQ.heap(j)) ← i
9              i ← j; j ← 2 * i ]
10         else exit
11  endwhile
12  PQ.heap(i) ← l
13  PQ.index(l) ← i
14  end HEAPIFY

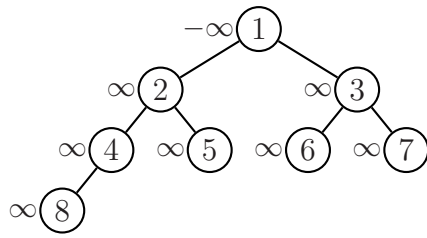
1  procedure DECREASE_KEY(PQ, l, newkey)
2  PQ.key(l) ← newkey
3  i ← PQ.index(l)
4  j ← ⌊i/2⌋
5  while i > 1 and PQ.key(PQ.heap(j)) > newkey do
6      PQ.heap(i) ← PQ.heap(j)
7      PQ.index(PQ.heap(j)) ← i
8      i ← j; j ← ⌊i/2⌋
9  endwhile
10  PQ.heap(i) ← l
11  PQ.index(l) ← i
12  end DECREASE_KEY

```

Procedure 10.1 Implementing Prim's algorithm using a min-heap

Figure 10.5 gives a partial trace of Procedure 10.1 when applied to the graph of Example 10.1. It shows the min-heap, and its internal representation, after an EXTRACT_MIN or DECREASE_KEY operation completes at selected stages of the algorithm.

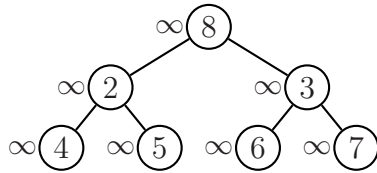
Initially: start vertex s is vertex 1



	1	2	3	4	5	6	7	8	↙ sizePQ
heap(1:8)	1	2	3	4	5	6	7	8	
index(1:8)	1	2	3	4	5	6	7	8	
key(1:8)	$-\infty$	∞	∞	∞	∞	∞	∞	∞	

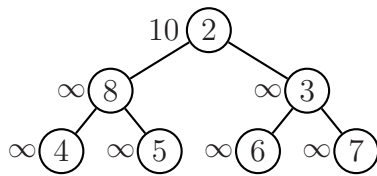
Stage 1

(a) After call to EXTRACT_MIN: vertex 1 in T



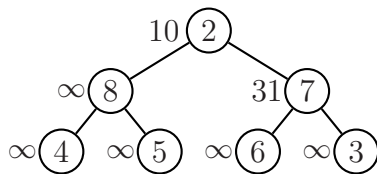
	1	2	3	4	5	6	7	8	↙ sizePQ
heap(1:8)	8	2	3	4	5	6	7	dc	
index(1:8)	dc	2	3	4	5	6	7	1	
key(1:8)	$-\infty$	∞	∞	∞	∞	∞	∞	∞	

(b) After call to DECREASE_KEY for vertex 2: $pred(2) = 1$



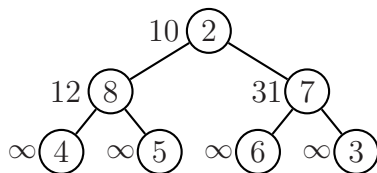
	1	2	3	4	5	6	7	8	↙ sizePQ
heap(1:8)	2	8	3	4	5	6	7	dc	
index(1:8)	dc	1	3	4	5	6	7	2	
key(1:8)	$-\infty$	10	∞	∞	∞	∞	∞	∞	

(c) After call to DECREASE_KEY for vertex 7: $pred(7) = 1$



	1	2	3	4	5	6	7	8	↙ sizePQ
heap(1:8)	2	8	7	4	5	6	3	dc	
index(1:8)	dc	1	7	4	5	6	3	2	
key(1:8)	$-\infty$	10	∞	∞	∞	∞	31	∞	

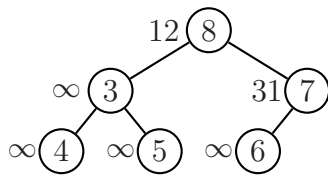
(d) After call to DECREASE_KEY for vertex 8: $pred(8) = 1$



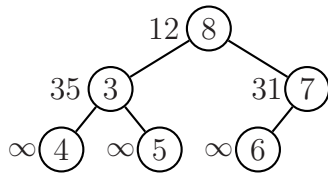
	1	2	3	4	5	6	7	8	↙ sizePQ
heap(1:8)	2	8	7	4	5	6	3	dc	
index(1:8)	dc	1	7	4	5	6	3	2	
key(1:8)	$-\infty$	10	∞	∞	∞	∞	31	12	

Figure 10.5 A partial trace of procedure PRIM for the graph of Example 10.1
(continued on next page)

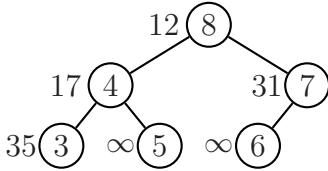
Stage 2

(a) After call to EXTRACT_MIN: edge (2,1) in T 

	↙ sizePQ							
	1	2	3	4	5	6	7	8
heap(1:8)	8	3	7	4	5	6	dc	dc
	1	2	3	4	5	6	7	8
index(1:8)	dc	dc	2	4	5	6	3	1
key(1:8)	-∞	-∞	∞	∞	∞	∞	31	12

(b) After call to DECREASE_KEY for vertex 3: $pred(3) = 2$ 

	↙ sizePQ							
	1	2	3	4	5	6	7	8
heap(1:8)	8	3	7	4	5	6	dc	dc
	1	2	3	4	5	6	7	8
index(1:8)	dc	dc	2	4	5	6	3	1
key(1:8)	-∞	-∞	35	∞	∞	∞	31	12

(c) After call to DECREASE_KEY for vertex 4: $pred(4) = 2$ 

	↙ sizePQ							
	1	2	3	4	5	6	7	8
heap(1:8)	8	4	7	3	5	6	dc	dc
	1	2	3	4	5	6	7	8
index(1:8)	dc	dc	4	2	5	6	3	1
key(1:8)	-∞	-∞	35	17	∞	∞	31	12

Do stages 3 through 6 as an exercise

Stage 7

(a) After call to EXTRACT_MIN: edge (7,4) in T 

	↙ sizePQ							
	1	2	3	4	5	6	7	8
heap(1:8)	3	dc	dc	dc	dc	dc	dc	dc
	1	2	3	4	5	6	7	8
index(1:8)	dc	dc	1	dc	dc	dc	dc	dc
key(1:8)	-∞	-∞	35	-∞	-∞	-∞	-∞	-∞

Figure 10.5 A partial trace of procedure PRIM for the graph of Example 10.1
(continued on next page)

Stage 8

 (a) After call to EXTRACT_MIN: edge (3,2) in T

$$\downarrow \text{size}PQ$$

	0	1	2	3	4	5	6	7	8
heap(1:8)		dc	dc	dc	dc	dc	dc	dc	dc

	1	2	3	4	5	6	7	8
index(1:8)	dc	dc	dc	dc	dc	dc	dc	dc
key(1:8)	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Figure 10.5 A partial trace of procedure PRIM for the graph of Example 10.1

Recovering the minimum-cost spanning tree

The minimum-cost spanning tree found by PRIM is encoded in the $pred$ array, which is returned as a component of the data structure \mathbb{G} . For our sample graph, the resulting array is shown in Figure 10.6.

	1	2	3	4	5	6	7	8
pred(1:8)	0	1	2	2	4	5	4	1

 Figure 10.6 The $pred$ array for the graph of Example 10.1

Procedure DISPLAY_MST recovers the minimum-cost spanning tree encoded in the $pred$ array by listing the edges which comprise the tree along with their respective costs. For our present example, the results are shown in Figure 10.7.

```

1  procedure DISPLAY_MST( $\mathbb{G}$ )
2     $TCost \leftarrow 0$ 
3    for  $l \leftarrow 1$  to  $n$  do
4      if  $\mathbb{G}.pred(l) \neq 0$  then [  $\alpha \leftarrow \mathbb{G}.LIST(l)$ 
5                                while  $\alpha \neq \Lambda$  do
6                                   $j \leftarrow \mathbb{G}.VRTX(\alpha)$ 
7                                  if  $\mathbb{G}.pred(l) = j$  then [ output  $l, j, \mathbb{G}.COST(\alpha)$ 
8                                                          $TCost \leftarrow TCost + \mathbb{G}.COST(\alpha)$  ]
9                                   $\alpha \leftarrow \mathbb{G}.NEXT(\alpha)$ 
10                                 endwhile ]
11  endfor
12  output  $TCost$ 
13  end DISPLAY_MST
    
```

 Procedure 10.2 Recovering the minimum-cost spanning tree encoded in $pred$ array

Edge		Cost	
(2,1)	—	10	
(3,2)	—	35	
(4,2)	—	17	
(5,4)	—	13	
(6,5)	—	8	
(7,4)	—	20	
(8,1)	—	12	Total cost = 115

Figure 10.7 Minimum-cost spanning tree for the graph of Example 10.1**Analysis of Prim's algorithm**

Let $G = (V, E)$ be a connected, weighted undirected graph on $n = |V|$ vertices and $e = |E|$ edges. We analyze Prim's algorithm as implemented in Procedure 10.1, i.e., using a binary min-heap as the container for the vertices in $V - U$, as follows.

1. Initializing the min-heap of size $\text{size}PQ = n$ in lines 3–6 of procedure InitPQ clearly takes $O(n)$ time.
2. In an *extract* operation, the vertex at the root of the heap is deleted and is replaced by the vertex stored in the rightmost leaf at the bottommost level of the heap; restoring the heap-order property may cause this vertex to migrate back to the bottommost level in $O(\log n)$ time. Exactly n EXTRACT_MIN operations are performed in lines 4–5 of PRIM, taking $O(n \log n)$ time. [Actually, the heap decreases in size after every *extract* operation, but this upper bound still applies. See Eq.(7.5) in Session 7.]
3. In a *change* operation, an element at the bottommost level of the heap whose key is changed may migrate to the root of the heap, taking $O(\log n)$ time. As we have pointed out earlier, we achieve this $\log n$ performance for DECREASE_KEY because the *index* array allows us to find in $O(1)$ time the position in the heap of the vertex whose key is decreased; this operation would otherwise take $O(n)$ time. At most e calls to DECREASE_KEY may be made in line 11 of PRIM, taking $O(e \log n)$ time.

Since any non-trivial connected undirected graph on n vertices has $e \geq n$ edges, the time complexity of Prim's algorithm as implemented in Procedure 10.1 is $O(e \log n)$.

Actually, the above analysis is overly pessimistic. When a vertex, say j , is added to U In Step 2 of Prim's algorithm, it is unlikely that all the vertices in $V - U$ which are adjacent to vertex j will have j as their new nearest neighbor, which would require a change in priority. Thus, in general, there will be much fewer than e calls to DECREASE_KEY.

10.1.2 Kruskal's algorithm

Let $G = (V, E)$ be a connected, weighted, undirected graph. Kruskal's algorithm builds a minimum-cost spanning tree for G edge by edge, with the edges considered in non-decreasing order of their cost.

Let T denote the set of edges already selected at any stage of the algorithm. Kruskal's algorithm may be stated as follows:

Kruskal's algorithm

1. [Initial edge.] Choose the edge of least cost among all the edges in E and place it in T .
2. [Next edge.] From among the remaining edges in E choose the edge of least cost, say edge (i, j) . If including edge (i, j) in T creates a cycle with the edges already in T , discard (i, j) ; otherwise, include (i, j) in T .
3. [Enough edges in T ?] Repeat Step 2 until there are $n - 1$ edges in T . Then T is a minimum-cost spanning tree for G .

As with Prim's algorithm, we see from this description of Kruskal's algorithm that it is also a straightforward application of the MST theorem. To see how the algorithm actually works, and to assess what steps are critical in implementing the algorithm on a computer, consider the following example. For the moment, ignore the 'Forest' column.

Example 10.2. Kruskal's algorithm to find a minimum-cost spanning tree

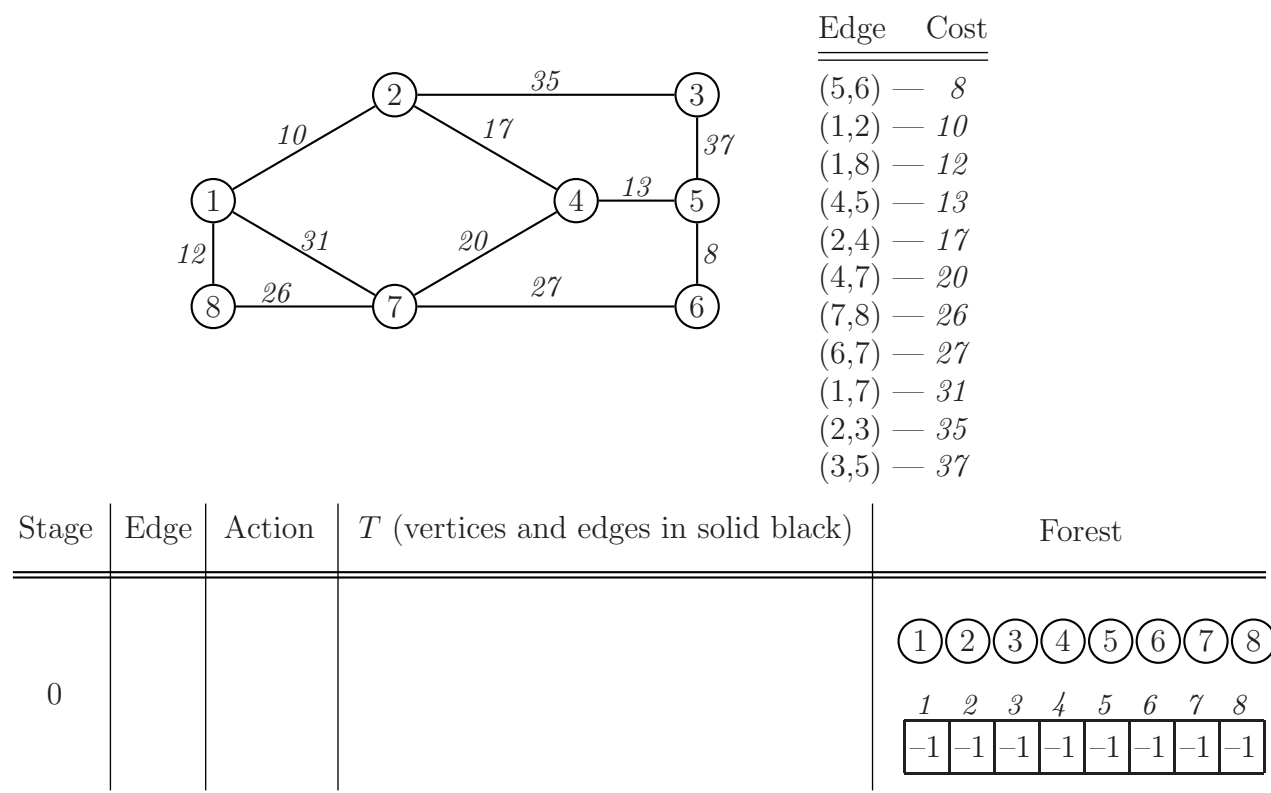


Figure 10.8 Kruskal's algorithm in action (continued on next page)

Stage	Edge	Action	T (vertices and edges in solid black)	Forest																
1	(5,6)	Accept		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>6</td><td>-2</td><td>-1</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	-1	-1	-1	-1	6	-2	-1	-1
1	2	3	4	5	6	7	8													
-1	-1	-1	-1	6	-2	-1	-1													
2	(1,2)	Accept		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>-2</td><td>-1</td><td>-1</td><td>6</td><td>-2</td><td>-1</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	2	-2	-1	-1	6	-2	-1	-1
1	2	3	4	5	6	7	8													
2	-2	-1	-1	6	-2	-1	-1													
3	(1,8)	Accept		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>-3</td><td>-1</td><td>-1</td><td>6</td><td>-2</td><td>-1</td><td>2</td></tr></table>	1	2	3	4	5	6	7	8	2	-3	-1	-1	6	-2	-1	2
1	2	3	4	5	6	7	8													
2	-3	-1	-1	6	-2	-1	2													
4	(4,5)	Accept		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>-3</td><td>-1</td><td>6</td><td>6</td><td>-3</td><td>-1</td><td>2</td></tr></table>	1	2	3	4	5	6	7	8	2	-3	-1	6	6	-3	-1	2
1	2	3	4	5	6	7	8													
2	-3	-1	6	6	-3	-1	2													
5	(2,4)	Accept		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>6</td><td>-1</td><td>6</td><td>6</td><td>-6</td><td>-1</td><td>2</td></tr></table>	1	2	3	4	5	6	7	8	2	6	-1	6	6	-6	-1	2
1	2	3	4	5	6	7	8													
2	6	-1	6	6	-6	-1	2													

Figure 10.8 Kruskal's algorithm in action (continued on next page)

Stage	Edge	Action	T (vertices and edges in solid black)	Forest																
6	(4,7)	Accept		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>6</td><td>-1</td><td>6</td><td>6</td><td>-7</td><td>6</td><td>2</td></tr></table>	1	2	3	4	5	6	7	8	2	6	-1	6	6	-7	6	2
1	2	3	4	5	6	7	8													
2	6	-1	6	6	-7	6	2													
7	(7,8)	Reject		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>6</td><td>-1</td><td>6</td><td>6</td><td>-7</td><td>6</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	2	6	-1	6	6	-7	6	6
1	2	3	4	5	6	7	8													
2	6	-1	6	6	-7	6	6													
8	(6,7)	Reject		= As is =																
9	(1,7)	Reject		 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>6</td><td>-1</td><td>6</td><td>6</td><td>-7</td><td>6</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	2	6	-1	6	6	-7	6	6
1	2	3	4	5	6	7	8													
2	6	-1	6	6	-7	6	6													
10	(2,3)	Accept	<p>Cost = 115</p>	 <table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>-8</td><td>6</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	6	6	6	6	6	-8	6	6
1	2	3	4	5	6	7	8													
6	6	6	6	6	-8	6	6													

Figure 10.8 Kruskal's algorithm in action

We take note of the following observations pertaining to Example 10.2 of Figure 10.8.

1. The input to Kruskal consists of the edges presorted by non-decreasing cost. Thus, choosing the edge of least cost in steps 1 and 2 of Kruskal's algorithm is trivially accomplished. However, if the graph is dense ($e \approx n^2/2$), presorting the edges may not be warranted. This is because the algorithm terminates, and ignores the rest of the edges, once $n - 1$ edges are accepted.
2. Unlike in Prim's algorithm, the growing tree consists of disconnected components which eventually become one connected component. In Figure 10.8 these are the vertices and edges shown in solid black which comprise T .
3. Determining whether a candidate edge will form a cycle with those already accepted is trivially accomplished when applying the algorithm 'by hand'; the task ceases to be trivial when performed on a computer. (Think about it.)

Program implementation of Kruskal's algorithm

There are essentially two questions that we need to address in implementing Kruskal's algorithm on a computer, namely:

1. Assuming that the edges are *not* presorted, how do we find the edge of least cost (call this a 'candidate edge') from among the remaining edges in E ? In Figure 10.8 above, these are the edges shown as dotted lines.
2. Having found a candidate edge, how do we determine whether or not it will form a cycle with the edges already in T ?

Needless to say, the answers to these questions will determine how efficiently Kruskal's algorithm is implemented.

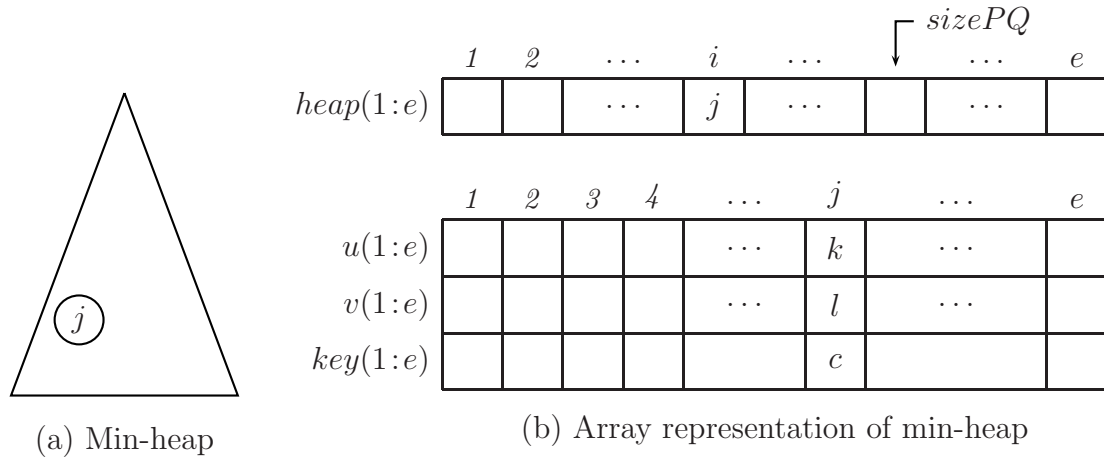
1. *Finding a candidate edge from E*

In our implementation of Prim's algorithm we used a priority queue, implemented using a min-heap, to maintain the remaining vertices in V . In like manner, we will use a min-heap to maintain, this time, the remaining edges in E . Unlike in Prim's algorithm, in which the priority of a vertex in the priority queue may change (requiring a call to `DECREASE_KEY`), in Kruskal's algorithm the priority of an edge in the priority queue never changes (obviating the need to maintain the *index* array). Consequently, we now represent the priority queue using the data structure

$$\mathbb{PQ} = [\text{heap}(1:e), u(1:e), v(1:e), \text{key}(1:e), \text{sizePQ}]$$

as depicted in Figure 10.9. At each iteration of the algorithm, the edge of least cost is the edge at the root of the heap. This is retrieved from the priority queue via a call to `EXTRACT_MIN`, which then calls on `HEAPIFY` to convert the resulting almost-heap into a heap once again. The elements of the $u - v - \text{key}$ arrays, which represent edges, neither move nor change throughout the execution of the algorithm; only the entries in

the *heap* array, which are pointers (indices) to these edges, are moved in order to maintain the heap-order property at all times.



e = number of edges in G

$sizePQ$ = current size of the min-heap (or equivalently, of the priority queue)
 = e initially

$(k, l) = (u(j), v(j))$ = edge in cell i of *heap* array

$c = key(j) = \text{cost of edge } (k, l)$

Figure 10.9 Min-heap implementation of priority queue for Kruskal's algorithm

2. Detecting a would-be cycle in T

We observed earlier in connection with Example 10.2 that Kruskal's algorithm grows a minimum-cost spanning tree T for G as a collection of disconnected components (shown in solid black) that eventually become one connected component. A candidate edge whose end vertices belong to the same component is rejected because clearly a cycle will be formed in the component if the edge were otherwise accepted. On the other hand, a candidate edge whose end vertices belong to two different components is accepted, and the two components become one.

It is immediately clear that we have here an instance of the equivalence problem discussed in section 8.6 of Session 8. Each component of the growing tree is an equivalence class and a candidate edge (i, j) is an equivalence relation $i \equiv j$. If $\text{FIND}(i) = \text{FIND}(j)$ then i and j belong to the same equivalence class, or equivalently, the edge (i, j) connects two vertices in the same component and is therefore rejected. If $\text{FIND}(i) \neq \text{FIND}(j)$ then i and j belong to two different equivalence classes, or equivalently, the edge (i, j) connects two vertices in two different components of T and is therefore included in T ; correspondingly, the two classes are merged via a call to UNION.

Go through Example 10.2 again, this time paying particular attention to the last column headed 'Forest'. Note how the notion of equivalence classes, represented using a forest of oriented trees, provides a most elegant way by which we can check whether a candidate edge will form a cycle with the edges in T in step 2 of Kruskal's algorithm.

EASY procedures to implement Kruskal's algorithm

The EASY procedure KRUSKAL(\mathbb{G}) implements Kruskal's algorithm as described above. The input to the procedure is a connected, weighted, undirected graph $G = (V, E)$, represented by the data structure $\mathbb{G} = [LIST(1:n), (VRTX, COST, NEXT), T(1:2, 1:n-1), n]$. The first two components of \mathbb{G} comprise the cost adjacency lists representation of G . The minimum-cost spanning tree found by Kruskal is returned in the T array; this comprise the output. The *FATHER* array is assumed to be global.

```

1  procedure KRUSKAL( $\mathbb{G}$ )
2     $FATHER \leftarrow -1$ 
3    call InitPQ( $\mathbb{G}, \mathbb{PQ}$ )
▷ Generate minimum-cost spanning tree
4     $m \leftarrow 0$ 
5    while  $m < \mathbb{G}.n - 1$  do
6      call EXTRACT_MIN( $\mathbb{PQ}, j$ )
7       $k \leftarrow \text{FIND}(\mathbb{PQ}.u(j))$ 
8       $l \leftarrow \text{FIND}(\mathbb{PQ}.v(j))$ 
9      if  $k \neq l$  then [  $m \leftarrow m + 1$ 
10          $T(1, m) \leftarrow \mathbb{PQ}.u(j)$ 
11          $T(2, m) \leftarrow \mathbb{PQ}.v(j)$ 
12         call UNION( $k, l$ ) ]
13    endwhile
14    end KRUSKAL

1  procedure InitPQ( $\mathbb{G}, \mathbb{PQ}$ )
2     $i \leftarrow 0$ 
3    for  $k \leftarrow 1$  to  $\mathbb{G}.n$  do
4       $\alpha \leftarrow \mathbb{G}.LIST(k)$ 
5      while  $\alpha \neq \Lambda$  do
6         $l \leftarrow \mathbb{G}.VRTX(\alpha)$ 
7        if  $k < l$  then [  $i \leftarrow i + 1$ ;  $\mathbb{PQ}.heap(i) \leftarrow i$ 
8            $\mathbb{PQ}.u(i) \leftarrow k$ ;  $\mathbb{PQ}.v(i) \leftarrow l$ ;  $\mathbb{PQ}.key(i) \leftarrow \mathbb{G}.COST(\alpha)$  ]
9         $\alpha \leftarrow \mathbb{G}.NEXT(\alpha)$ 
10     endwhile
11    endfor
12     $\mathbb{PQ}.sizePQ \leftarrow i$ 
13    for  $i \leftarrow \lfloor \mathbb{PQ}.sizePQ/2 \rfloor$  to 1 by -1 do
14      call HEAPIFY( $\mathbb{PQ}, i$ )
15    endfor
16    end InitPQ

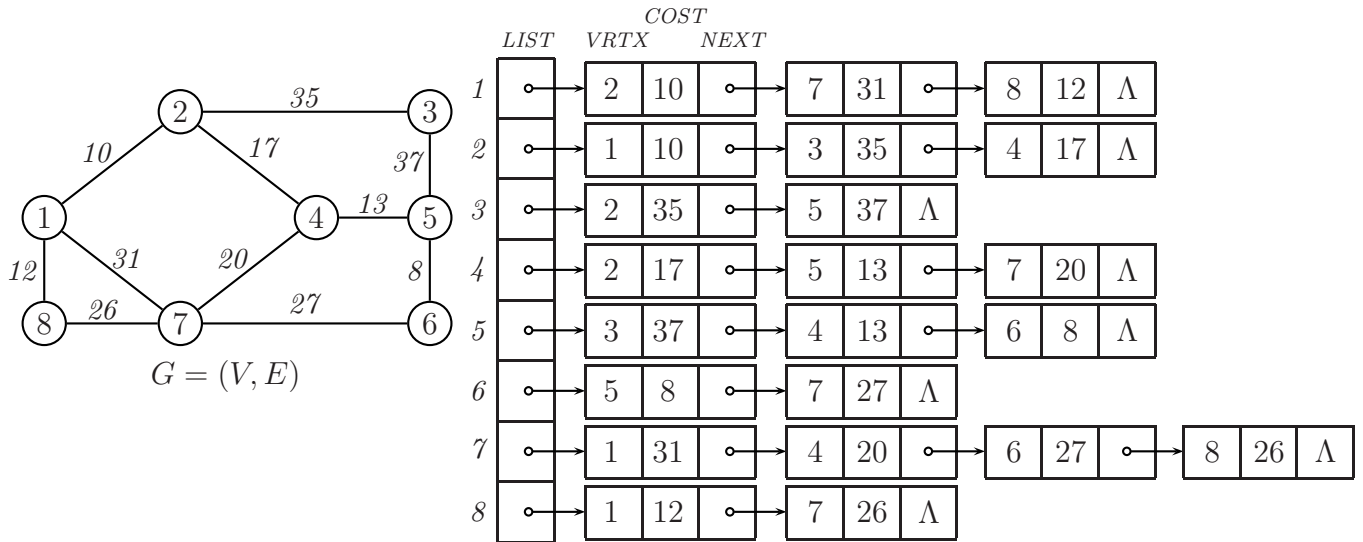
```

Procedure 10.3 Implementing Kruskal's algorithm using a min-heap

Procedures EXTRACT_MIN and HEAPIFY are as given in connection with Prim's algorithm (Procedure 10.1), except that line 4 in EXTRACT_MIN and lines 8 and 13 in HEAPIFY should be commented out. The FIND and UNION procedures are exactly as given in Session 8 (Procedure 8.6 and Procedure 8.7).

10.1 Minimum-cost spanning trees for undirected graphs 307

Figure 10.10 shows a trace of Kruskal's algorithm, as implemented in Procedure 10.3, when applied to the graph $G = (V, E)$ of Example 10.2, reproduced below along with its cost adjacency matrix.



	Priority queue, \mathbb{PQ}	T	Forest																																																												
0	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>9</td><td>1</td><td>3</td><td>8</td><td>5</td><td>6</td><td>7</td><td>2</td><td>4</td><td>10</td><td>11</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	9	1	3	8	5	6	7	2	4	10	11	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr><tr><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	<table><tr><td>①</td><td>②</td><td>③</td><td>④</td><td>⑤</td><td>⑥</td><td>⑦</td><td>⑧</td></tr></table>	①	②	③	④	⑤	⑥	⑦	⑧									
	1	2	3	4	5	6	7	8	9	10	11																																																				
	9	1	3	8	5	6	7	2	4	10	11																																																				
1	2	3	4	5	6	7																																																									
dc	dc	dc	dc	dc	dc	dc																																																									
dc	dc	dc	dc	dc	dc	dc																																																									
①	②	③	④	⑤	⑥	⑦	⑧																																																								
	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9	10	11																																																					
1	1	1	2	2	3	4	4	5	6	7																																																					
2	7	8	3	4	5	5	7	6	7	8																																																					
10	31	12	35	17	37	13	20	8	27	26																																																					
1	2	3	4	5	6	7	8																																																								
-1	-1	-1	-1	-1	-1	-1	-1																																																								
																																																												
1	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>5</td><td>3</td><td>8</td><td>11</td><td>6</td><td>7</td><td>2</td><td>4</td><td>10</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	5	3	8	11	6	7	2	4	10	dc	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr><tr><td>6</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	5	dc	dc	dc	dc	dc	dc	6	dc	dc	dc	dc	dc	dc	<table><tr><td>①</td><td>②</td><td>③</td><td>④</td><td>⑥</td><td>⑦</td><td>⑧</td></tr><tr><td></td><td></td><td></td><td></td><td>⑤</td><td></td><td></td></tr></table>	①	②	③	④	⑥	⑦	⑧					⑤					
	1	2	3	4	5	6	7	8	9	10	11																																																				
	1	5	3	8	11	6	7	2	4	10	dc																																																				
1	2	3	4	5	6	7																																																									
5	dc	dc	dc	dc	dc	dc																																																									
6	dc	dc	dc	dc	dc	dc																																																									
①	②	③	④	⑥	⑦	⑧																																																									
				⑤																																																											
	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26	Edge (5,6) accepted.	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>6</td><td>-2</td><td>-1</td><td>-1</td></tr></table>	1	2	3	4	5	6	7	8	-1	-1	-1	-1	6	-2	-1	-1
1	2	3	4	5	6	7	8	9	10	11																																																					
1	1	1	2	2	3	4	4	5	6	7																																																					
2	7	8	3	4	5	5	7	6	7	8																																																					
10	31	12	35	17	37	13	20	8	27	26																																																					
1	2	3	4	5	6	7	8																																																								
-1	-1	-1	-1	6	-2	-1	-1																																																								

Figure 10.10 A trace of procedure KRUSKAL for the graph of Example 10.2
(continued on next page)

	Priority queue, \mathbb{PQ}	T	Forest																																																																	
2	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>3</td><td>5</td><td>7</td><td>8</td><td>11</td><td>6</td><td>10</td><td>2</td><td>4</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	3	5	7	8	11	6	10	2	4	dc	dc	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table> <p>Edge (1,2) accepted.</p>	1	2	3	4	5	6	7	5	1	dc	dc	dc	dc	dc	6	2	dc	dc	dc	dc	dc																							
	1	2	3	4	5	6	7	8	9	10	11																																																									
	3	5	7	8	11	6	10	2	4	dc	dc																																																									
1	2	3	4	5	6	7																																																														
5	1	dc	dc	dc	dc	dc																																																														
6	2	dc	dc	dc	dc	dc																																																														
3	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table> <p>Edge (1,8) accepted.</p>	1	2	3	4	5	6	7	5	1	1	dc	dc	dc	dc	6	2	8	dc	dc	dc	dc	
	1	2	3	4	5	6	7	8	9	10	11																																																									
	1	1	1	2	2	3	4	4	5	6	7																																																									
2	7	8	3	4	5	5	7	6	7	8																																																										
10	31	12	35	17	37	13	20	8	27	26																																																										
1	2	3	4	5	6	7																																																														
5	1	1	dc	dc	dc	dc																																																														
6	2	8	dc	dc	dc	dc																																																														
4	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>dc</td><td>dc</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>dc</td><td>dc</td><td>dc</td></tr></table> <p>Edge (4,5) accepted.</p>	1	2	3	4	5	6	7	5	1	1	4	dc	dc	dc	6	2	8	5	dc	dc	dc	
	1	2	3	4	5	6	7	8	9	10	11																																																									
	1	1	1	2	2	3	4	4	5	6	7																																																									
2	7	8	3	4	5	5	7	6	7	8																																																										
10	31	12	35	17	37	13	20	8	27	26																																																										
1	2	3	4	5	6	7																																																														
5	1	1	4	dc	dc	dc																																																														
6	2	8	5	dc	dc	dc																																																														
5	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>2</td><td>dc</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>4</td><td>dc</td><td>dc</td></tr></table> <p>Edge (2,4) accepted.</p>	1	2	3	4	5	6	7	5	1	1	4	2	dc	dc	6	2	8	5	4	dc	dc	
	1	2	3	4	5	6	7	8	9	10	11																																																									
	1	1	1	2	2	3	4	4	5	6	7																																																									
2	7	8	3	4	5	5	7	6	7	8																																																										
10	31	12	35	17	37	13	20	8	27	26																																																										
1	2	3	4	5	6	7																																																														
5	1	1	4	2	dc	dc																																																														
6	2	8	5	4	dc	dc																																																														
6	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>2</td><td>4</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>4</td><td>7</td><td>dc</td></tr></table> <p>Edge (4,7) accepted.</p>	1	2	3	4	5	6	7	5	1	1	4	2	4	dc	6	2	8	5	4	7	dc	
	1	2	3	4	5	6	7	8	9	10	11																																																									
	1	1	1	2	2	3	4	4	5	6	7																																																									
2	7	8	3	4	5	5	7	6	7	8																																																										
10	31	12	35	17	37	13	20	8	27	26																																																										
1	2	3	4	5	6	7																																																														
5	1	1	4	2	4	dc																																																														
6	2	8	5	4	7	dc																																																														

Figure 10.10 A trace of procedure KRUSKAL for the graph of Example 10.2
(continued on next page)

	Priority queue, \mathbb{PQ}	T	Forest																																																												
7	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>10</td><td>2</td><td>4</td><td>6</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	10	2	4	6	dc	dc	dc	dc	dc	dc	dc	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>2</td><td>4</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>4</td><td>7</td><td>dc</td></tr></table> <p>Edge (7,8) rejected.</p>	1	2	3	4	5	6	7	5	1	1	4	2	4	dc	6	2	8	5	4	7	dc	<div><div>③</div><div><div>2</div><div>4</div><div>5</div><div>7</div><div>8</div></div><div>6</div><div>1</div></div>																	
	1	2	3	4	5	6	7	8	9	10	11																																																				
	10	2	4	6	dc	dc	dc	dc	dc	dc	dc																																																				
1	2	3	4	5	6	7																																																									
5	1	1	4	2	4	dc																																																									
6	2	8	5	4	7	dc																																																									
	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>2</td><td>6</td><td>-1</td><td>6</td><td>6</td><td>-7</td><td>6</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	2	6	-1	6	6	-7	6	6
1	2	3	4	5	6	7	8	9	10	11																																																					
1	1	1	2	2	3	4	4	5	6	7																																																					
2	7	8	3	4	5	5	7	6	7	8																																																					
10	31	12	35	17	37	13	20	8	27	26																																																					
1	2	3	4	5	6	7	8																																																								
2	6	-1	6	6	-7	6	6																																																								
8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>2</td><td>6</td><td>4</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	2	6	4	dc	dc	dc	dc	dc	dc	dc	dc	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>2</td><td>4</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>4</td><td>7</td><td>dc</td></tr></table> <p>Edge (6,7) rejected.</p>	1	2	3	4	5	6	7	5	1	1	4	2	4	dc	6	2	8	5	4	7	dc	= As is =																	
	1	2	3	4	5	6	7	8	9	10	11																																																				
	2	6	4	dc	dc	dc	dc	dc	dc	dc	dc																																																				
1	2	3	4	5	6	7																																																									
5	1	1	4	2	4	dc																																																									
6	2	8	5	4	7	dc																																																									
	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26																		
1	2	3	4	5	6	7	8	9	10	11																																																					
1	1	1	2	2	3	4	4	5	6	7																																																					
2	7	8	3	4	5	5	7	6	7	8																																																					
10	31	12	35	17	37	13	20	8	27	26																																																					
9	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>4</td><td>6</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	4	6	dc	dc	dc	dc	dc	dc	dc	dc	dc	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>2</td><td>4</td><td>dc</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>4</td><td>7</td><td>dc</td></tr></table> <p>Edge (1,7) rejected.</p>	1	2	3	4	5	6	7	5	1	1	4	2	4	dc	6	2	8	5	4	7	dc	<div><div>③</div><div><div>1</div><div>2</div><div>4</div><div>5</div><div>7</div><div>8</div></div><div>6</div></div>																	
	1	2	3	4	5	6	7	8	9	10	11																																																				
	4	6	dc	dc	dc	dc	dc	dc	dc	dc	dc																																																				
1	2	3	4	5	6	7																																																									
5	1	1	4	2	4	dc																																																									
6	2	8	5	4	7	dc																																																									
	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>6</td><td>6</td><td>-1</td><td>6</td><td>6</td><td>-7</td><td>6</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	6	6	-1	6	6	-7	6	6
1	2	3	4	5	6	7	8	9	10	11																																																					
1	1	1	2	2	3	4	4	5	6	7																																																					
2	7	8	3	4	5	5	7	6	7	8																																																					
10	31	12	35	17	37	13	20	8	27	26																																																					
1	2	3	4	5	6	7	8																																																								
6	6	-1	6	6	-7	6	6																																																								
10	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>6</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td><td>dc</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	6	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>5</td><td>1</td><td>1</td><td>4</td><td>2</td><td>4</td><td>2</td></tr><tr><td>6</td><td>2</td><td>8</td><td>5</td><td>4</td><td>7</td><td>3</td></tr></table> <p>Edge (2,3) accepted. Min-cost spanning tree T is complete.</p>	1	2	3	4	5	6	7	5	1	1	4	2	4	2	6	2	8	5	4	7	3	<div><div>3</div><div>1</div><div>2</div><div>4</div><div>5</div><div>7</div><div>8</div></div> <div>6</div>																	
	1	2	3	4	5	6	7	8	9	10	11																																																				
	6	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc																																																				
1	2	3	4	5	6	7																																																									
5	1	1	4	2	4	2																																																									
6	2	8	5	4	7	3																																																									
	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>2</td><td>7</td><td>8</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>6</td><td>7</td><td>8</td></tr><tr><td>10</td><td>31</td><td>12</td><td>35</td><td>17</td><td>37</td><td>13</td><td>20</td><td>8</td><td>27</td><td>26</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	1	1	1	2	2	3	4	4	5	6	7	2	7	8	3	4	5	5	7	6	7	8	10	31	12	35	17	37	13	20	8	27	26		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>-8</td><td>6</td><td>6</td></tr></table>	1	2	3	4	5	6	7	8	6	6	6	6	6	-8	6	6
1	2	3	4	5	6	7	8	9	10	11																																																					
1	1	1	2	2	3	4	4	5	6	7																																																					
2	7	8	3	4	5	5	7	6	7	8																																																					
10	31	12	35	17	37	13	20	8	27	26																																																					
1	2	3	4	5	6	7	8																																																								
6	6	6	6	6	-8	6	6																																																								

Figure 10.10 A trace of procedure KRUSKAL for the graph of Example 10.2

We take note of the following observations pertaining to the figure above.

1. The figure consists of 11 rows and three columns. A row corresponds to an iteration, or 'stage', of Kruskal's algorithm.

2. The column headed \mathbb{PQ} shows the priority queue, implemented as a min-heap and represented using four parallel arrays, at each stage of Kruskal's algorithm. Specifically, the first row (stage 0) shows the priority queue after the call to InitPQ in line 4 of KRUSKAL completes.

The entries in the *heap* array are *indices* into the $u - v - \text{key}$ arrays in which are stored the edges in E . Only the entries in the *heap* array are moved in order to maintain the heap-order property at every stage of the computations. The entries in the $u - v - \text{key}$ arrays neither move nor change throughout the execution of the algorithm.

3. The edge at the root of the heap, which is the candidate edge for next inclusion in T , is shown shaded. For instance, in stage 0 we have $\text{heap}(1) = 9$, and we find in cell 9 of $u - v - \text{key}$ the edge (5,9) with cost 8, which is the edge of least cost. In stage 1, we have $\text{heap}(1) = 1$, and we find in cell 1 of $u - v - \text{key}$ the edge (1,2) with cost 10, which is the edge of least cost among the remaining edges. And so on.
4. An entry of dc is made in the *heap* array for each edge that is removed from the priority queue.
5. The column headed T shows the spanning tree encoded in the array T as it grows one edge at a time.

In stage 1, the edge (5,6) is included in T and this is recorded with the entry $T(1,1) = 5$ and $T(2,1) = 6$ in the T array. In stage 2, the edge (1,2) is included in T and this is recorded with the entry $T(1,2) = 1$ and $T(2,2) = 2$. In stage 7, the candidate edge (7,8) is rejected, so no new entry is made into the T array.

6. The column headed 'Forest' shows the forest of oriented trees which represent the equivalence classes, encoded in the *FATHER* array, which correspond to the connected components in T . A candidate edge is accepted and included in T if its endpoints belong to two different trees in the forest. For instance, at the end of stage 3, the forest consists of five trees rooted at 2, 3, 4, 6 and 7; in stage 4 the candidate edge (4,5) is accepted because 4 and 5 belong to two different trees, one rooted at 4 and the other rooted at 6. At the end of stage 4, with the edge (4,5) included in T , the forest consists of four trees with 4 and 5 now belonging to the tree rooted at 6, a result of the weighting rule for the union operation.

At the end of stage 6, the forest consists of two trees rooted at 3 and 6; in stage 7 the candidate edge (7,8) is rejected because 7 and 8 belong to the same tree rooted at 6. At the end of stage 7 there are still two trees in the forest; however, note the change in the tree rooted at 6, a result of path compression during the find operation on 8.

7. The algorithm terminates once $n - 1$ edges have been accepted. The minimum-cost spanning tree encoded in T is returned as a component of the data structure \mathbb{G} .

Analysis of Kruskal's algorithm

Let $G = (V, E)$ be a connected, weighted, undirected graph on $n = |V|$ vertices and $e = |E|$ edges. We analyze Kruskal's algorithm as implemented in Procedure 10.3, i.e., using a binary min-heap as the container for the edges in E and the FIND-UNION procedures for detecting would-be cycles in T , as follows.

1. Initializing the forest of oriented trees, represented by the *FATHER* array, in line 3 of KRUSKAL takes $O(n)$ time. Assigning the edges in E to the nodes of a complete binary tree, represented by the *heap* array, in lines 2–11 of InitPQ takes $O(e)$ time. Converting the binary tree into a min-heap in lines 12–15 takes $O(e)$ time, as we have shown in Session 7 in connection with the analysis of heapsort [see Eq.(7.4) in Session 7].
2. In an *extract* operation on the min-heap, the edge at the root of the heap is deleted and is replaced by the edge stored in the rightmost leaf at the bottommost level of the heap; restoring the heap-order property may cause this edge to migrate back to the bottommost level in $O(\log e)$ time. In the worst case, e EXTRACT_MIN operations are performed in line 8 of KRUSKAL, taking $O(e \log e)$ time. [Actually, the heap decreases in size after every *extract* operation, but this upper bound still applies. See Eq.(7.5) in Session 7.]
3. Each edge (i, j) extracted from the heap is processed as an equivalence pair $i \equiv j$ in lines 9–12 of KRUSKAL to determine whether to accept or reject the edge. An $O(e)$ sequence of FIND-UNION operations takes $O(e G(e))$ time, where $G(e) \leq 5$ for all $e \leq 2^{65536}$ [see section 8.6 of Session 8].

Since $G(e)$ is essentially a constant for values of e for which Kruskal's algorithm will ever be used, the time complexity of Kruskal's algorithm as implemented in Procedure 10.3 is $O(e \log e)$.

We end this section with a comparison between Prim's and Kruskal's algorithms for finding a minimum-cost spanning tree, T , for a connected, weighted undirected graph, $G = (V, E)$.

1. Prim's algorithm builds T by taking in one vertex at a time, choosing that vertex which is connected by an edge of least cost to the (single) partial spanning tree, say T' , that it has built thus far. Kruskal's algorithm builds T by selecting one edge at a time, in the order of increasing edge cost, taking in an edge provided it does not create a cycle with the edges already selected. The spanning tree grows as a collection of disconnected components which eventually become one tree.
2. As implemented above, both algorithms use a priority queue, represented using a binary min-heap, to store the 'building blocks' that they use in building the tree – vertices in Prim's algorithm and edges in Kruskal's. The priority assigned to a vertex in Prim's is the cost of the edge which connects the vertex to its nearest neighbor in T' and this may change as T' grows. The priority assigned to an edge in Kruskal's is the cost of the edge, and this does not change. Thus the priority queue in the former, when implemented, must support the *change* operation (as in DECREASE_KEY above); that in the latter need not (however, see next item).
3. In addition to the priority queue ADT which both algorithms use, Kruskal's also utilizes the disjoint set ADT in order to detect cycles, adding more overhead.

10.2 Shortest path problems for directed graphs

Let $G = (V, E)$ be a directed graph with **nonnegative costs**, or *weights*, assigned to its edges. Let (u, v) be an edge in E ; we denote the cost or weight of edge (u, v) as $w((u, v))$. We define the cost (or weight or length) of a path from some vertex u to some other vertex v in V as the sum of the costs of the edges comprising the path. If there is no path from u to v , i.e., v is not reachable from u , we define the cost of the path from u to v to be ∞ . Two important path problems on such weighted graphs are:

1. the single-source shortest-paths (SSSP) problem – determine the cost of the *shortest* path from a given *source* or *start* vertex $s \in V$ to every other vertex in V
2. the all-pairs shortest-paths (APSP) problem – determine the cost of the *shortest* path between every pair of vertices in V

The classical solution to the SSSP problem is called **Dijkstra's algorithm**, which is in the same greedy class as Prim's and Kruskal's algorithms. For a graph on n vertices, we can view the APSP problem as n instances of the SSSP problem. To solve the APSP problem, therefore, we can apply Dijkstra's algorithm n times, taking each vertex in turn as the source vertex. However, there is a more direct solution to the APSP problem called **Floyd's algorithm**.

Closely related to the APSP problem is the problem of finding the *transitive closure* of a directed graph G . This time we simply want to determine whether a path exists between every pair of vertices in G . The classical solution to this problem is **Warshall's algorithm**, which actually predates Floyd's.

We will now study these three algorithms on graphs in depth.

10.2.1 Dijkstra's algorithm for the SSSP problem

The general idea behind Dijkstra's algorithm may be stated as follows: Each vertex is assigned a *class* and a *value*. A class 1 vertex is a vertex whose shortest distance from the source vertex s has already been found; a class 2 vertex is a vertex whose shortest distance from s has yet to be found. The value of a class 1 vertex is its distance from vertex s along a shortest path; the value of a class 2 vertex is its shortest distance, *found thus far*, from vertex s .

Initially, only the source vertex s is in class 1 with a value of 0; all the other vertices are in class 2 with a value of ∞ . Subsequently, vertices in class 2 migrate to class 1 as the shortest distance from s to each such vertex is found. Let $d(v)$ be the value of some vertex v when it is still in class 2 and let $\delta(v)$ be its value when v is already in class 1. By definition of a class 1 vertex, $\delta(v)$ is the cost (or weight or length) of the shortest path from s to v ; thus at all times we have $\delta(v) \leq d(v)$ for all $v \in V$. We can think of Dijkstra's algorithm as a process whereby $d(v)$ reduces to $\delta(v)$ for all vertices $v \in V$ that are reachable from s .

Now, the algorithm.

Dijkstra's algorithm

1. Place vertex s in class 1 and all other vertices in class 2.
2. Set the value of vertex s to zero and the value of all other vertices to ∞ .
3. Do the following until all vertices v in V that are reachable from s are placed in class 1:
 - a. Denote by u the vertex most recently placed in class 1.
 - b. Adjust all vertices v in class 2 as follows:
 - (i) If vertex v is not adjacent to u , retain the current value of $d(v)$.
 - (ii) If vertex v is adjacent to u , adjust $d(v)$ as follows:

$$\text{if } d(v) > \delta(u) + w((u, v)) \text{ then } d(v) \leftarrow \delta(u) + w((u, v)) \quad (10.1)$$
 - c. Choose a class 2 vertex with *minimal* value and place it in class 1.

Step 3.b.ii is called *edge relaxation*. Having found the shortest distance $\delta(u)$ from s to u , we check whether our current estimate $d(v)$ of the shortest distance from s to v can be improved by using the edge (u, v) . If indeed the path from s to v which passes through u is shorter, we take this path and set $d(v)$ to be the cost of this shorter path.

We take note of the following observations about Dijkstra's algorithm.

1. Dijkstra's algorithm as given above finds the cost of shortest path from the source vertex s to each of the other vertices $v \in V$, but it does not indicate what the intermediate vertices are, if any, in the shortest path from s to v . So that we can reconstruct such a path, we will maintain an array, say *pred*, which we initialize to zero, and subsequently update whenever we relax an edge, as shown below:

if $d(v) > \delta(u) + w((u, v))$ **then** [$d(v) \leftarrow \delta(u) + w((u, v))$; $\text{pred}(v) \leftarrow u$]

After Dijkstra's algorithm terminates, the shortest path from s to some other vertex $v \in V$ is found by simply following the chain of predecessors encoded in *pred* backwards from v to s .

2. A variant of the SSSP problem is the single-pair shortest-path (or SPSP) problem, i.e., find the shortest path from one start vertex s to one destination vertex t . This problem is not any easier to solve than the SSSP problem, and there is no known algorithm to solve this problem that runs any faster than Dijkstra's. Thus to solve the SPSP problem we apply Dijkstra's algorithm and terminate the computations once vertex t is placed in class 1. If t enters class 1 last we will have solved the SSSP problem as well!

Example 10.3. Dijkstra's algorithm to solve the SSSP problem

Figure 10.11 shows Dijkstra's algorithm in action as it finds the shortest paths from the source vertex $s = 1$ to all the other vertices in the input graph.

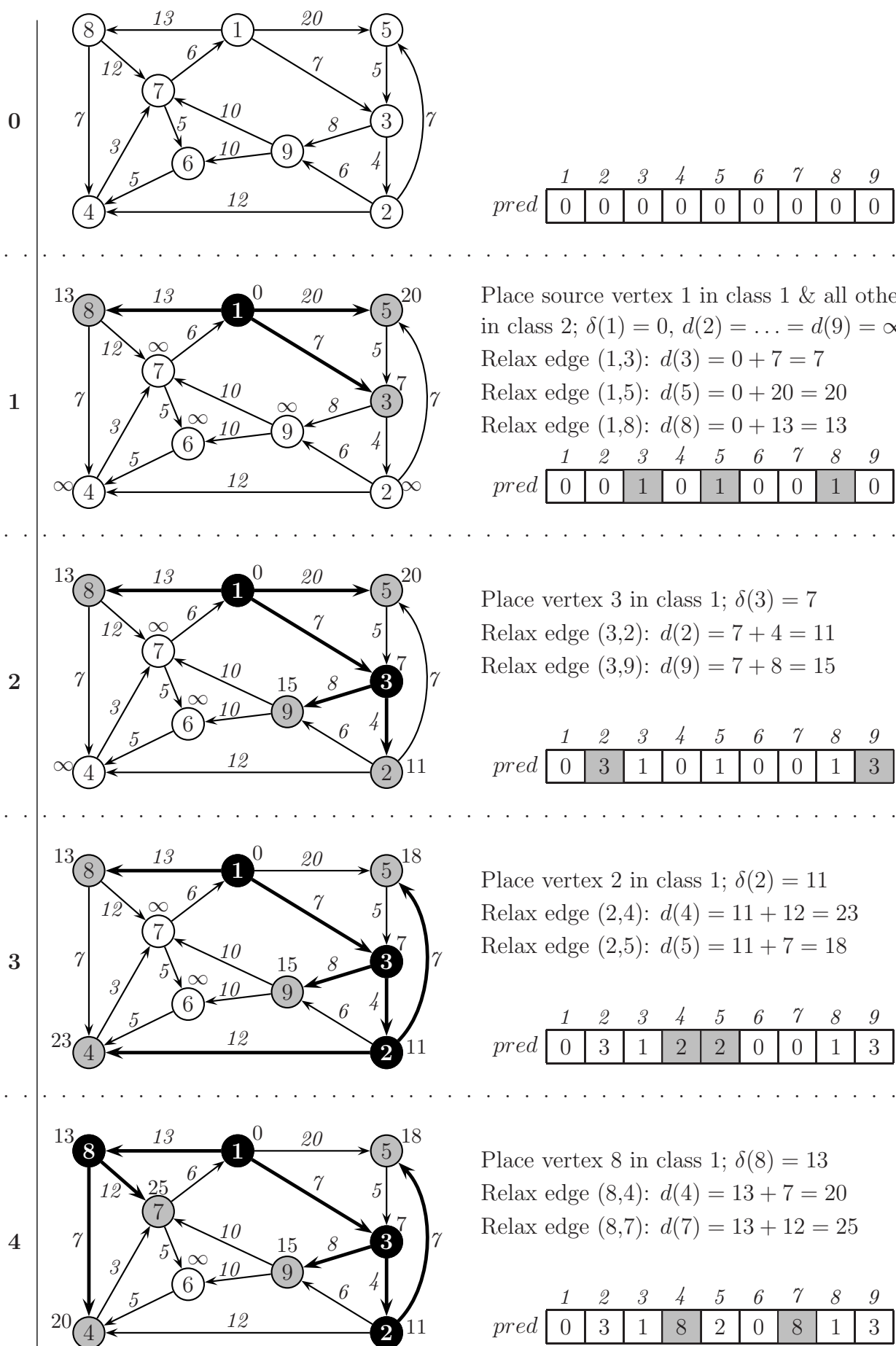


Figure 10.11 Dijkstra's algorithm in action (continued on next page)

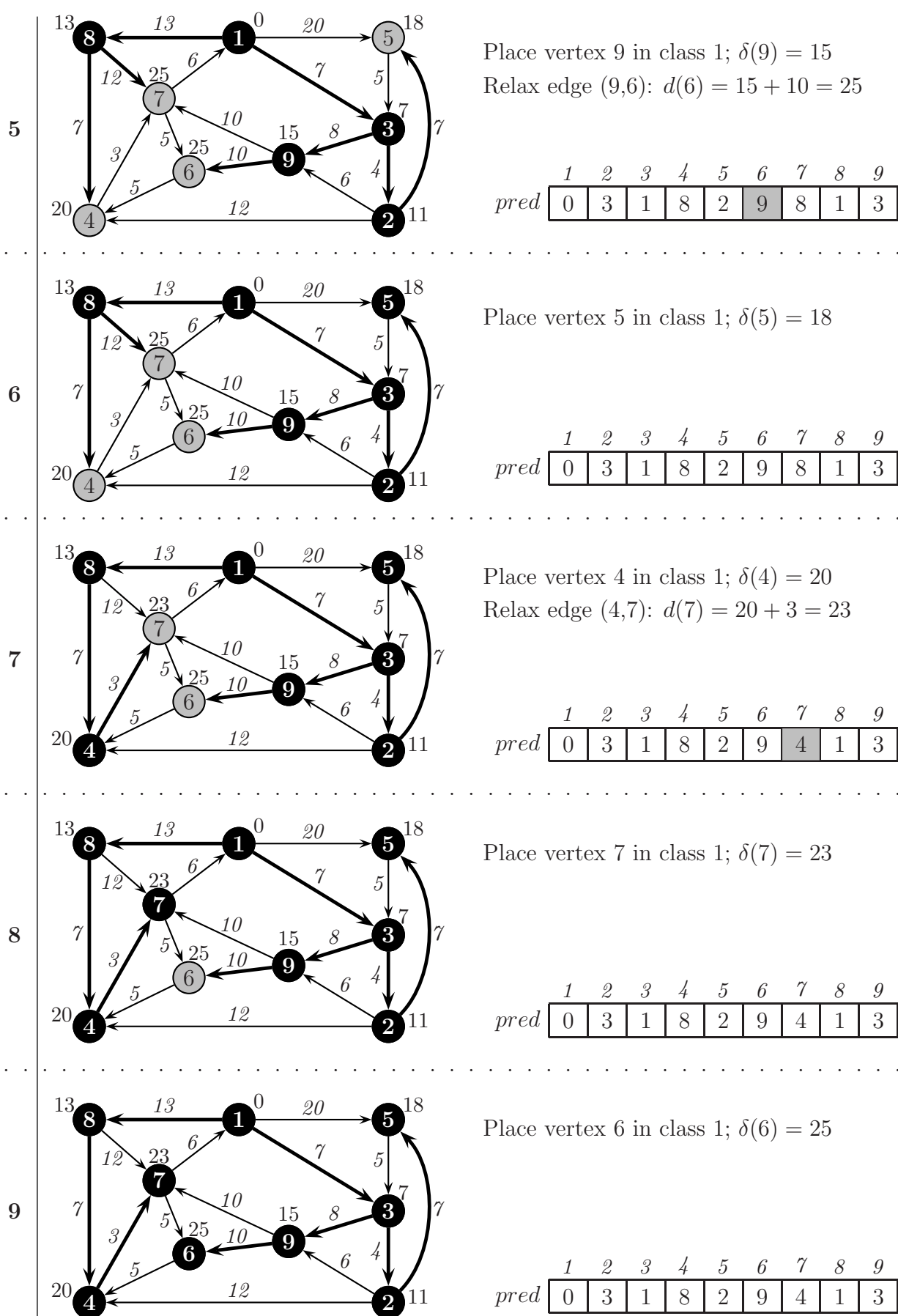


Figure 10.11 Dijkstra's algorithm in action

We take note of the following observations pertaining to Example 10.3 of Figure 10.11.

1. The figure is divided into rows, each row corresponding to a ‘stage’ of Dijkstra’s algorithm.
2. The black vertices are the class 1 vertices; the gray and white vertices are the class 2 vertices. In particular, the gray vertices are the *fringe* vertices; they are adjacent to one or more vertices in class 1. It is from among the fringe vertices that the vertex next placed in class 1 is chosen.
3. The number placed alongside a black vertex, say u , is its shortest distance $\delta(u)$ from the source vertex s ; the number placed alongside a gray vertex, say v , is its shortest distance $d(v)$ from the source vertex *found thus far*.
4. The fringe vertex with the smallest d -value is the vertex next placed in class 1 in Step 3.c of Dijkstra’s algorithm. We may cast this as a rule:

Dijkstra’s rule: *Choose the vertex on the fringe that is closest to the source vertex.*

5. The intermediate vertices, if any, in a shortest path from s to a black or gray vertex v are all black vertices.
6. The thick edges are edges in a shortest path from the source vertex, which is a final shortest path (if the end vertex is black) or a shortest path found so far (if the end vertex is gray). A thick edge leading to a gray vertex v may become thin again if a shorter path to v which does not include this edge is subsequently found. For instance, in stage 1 the edge (1,5) is thick; in stage 3, it becomes thin again for reasons explained below.
7. An edge leading from the vertex, say u , most recently placed in class 1 to a vertex, say v , in class 2 is relaxed if including the edge (u, v) shortens the path from s to v . For instance, in Stage 1 and up until the start of stage 3 the shortest path found so far from vertex 1 to vertex 5 is $1 \rightarrow 5$ with a cost $d(5) = 20$. With vertex 2 placed in class 1, a shorter path is found which includes edge (2,5), namely, $1 \rightarrow 3 \rightarrow 2 \rightarrow 5$ with a cost $d(5) = 18$. Edge (1,5) ceases to be an edge in a shortest path from s .
8. The *pred* array indicates the immediate predecessor of a vertex in a shortest path from s . If $\text{pred}(v) = u$, then edge (u, v) is thick. The values in the *pred* array may change from one stage to the next (indicated by the shaded entries) as tentative shortest paths are discarded when a still shorter path is found. At termination of Dijkstra’s algorithm the *pred* array encodes a **shortest-paths tree** rooted at the source vertex s which contains all the shortest paths from s to every other vertex $v \in V$ that is reachable from s . Figure 10.12 shows a shortest-paths tree for the graph of Example 10.3.

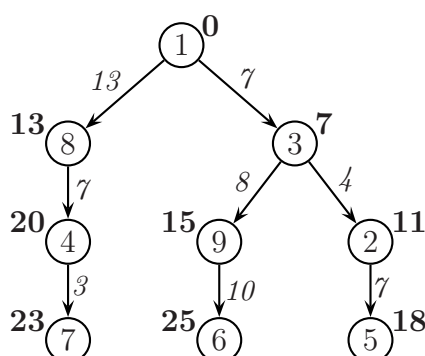


Figure 10.12 Shortest-paths tree rooted at $s = 1$ for the graph of Example 10.3

Correctness of Dijkstra's algorithm

Dijkstra's algorithm is a greedy algorithm. The greedy strategy is applied in Step 3.c when the class 2 vertex, say vertex v , with the smallest value $d(v)$ among all class 2 vertices, is placed next in class 1. That $d(v)$ is already the shortest distance from s to v , i.e., $d(v) = \delta(v)$, hinges on two facts about vertex v .

1. The shortest path from the source vertex s to vertex v passes through class 1 vertices only.
2. Step 3.b.ii correctly computes the cost of this shortest path, which is assigned as the value of vertex v .

Suppose, to the contrary, that there are class 2 vertices in a shortest path from s to v as shown in Figure 10.13. If the hypothetical path $s \rightsquigarrow x \rightsquigarrow v$ is shorter than the path $s \rightsquigarrow v$, then it must be true that $l_2 < l_1$, since l_3 cannot be negative if all edge costs are nonnegative. Now, if $l_2 < l_1$, then vertex x must have been placed in class 1 ahead of vertex v . Since this is not in fact the case, then l_1 must be shorter than l_2 , which means that the shortest path from s to v passes through class 1 vertices only. Note that this argument hinges on the assumption that edge costs are nonnegative; if there are negative costs, Dijkstra's algorithm will not work properly.

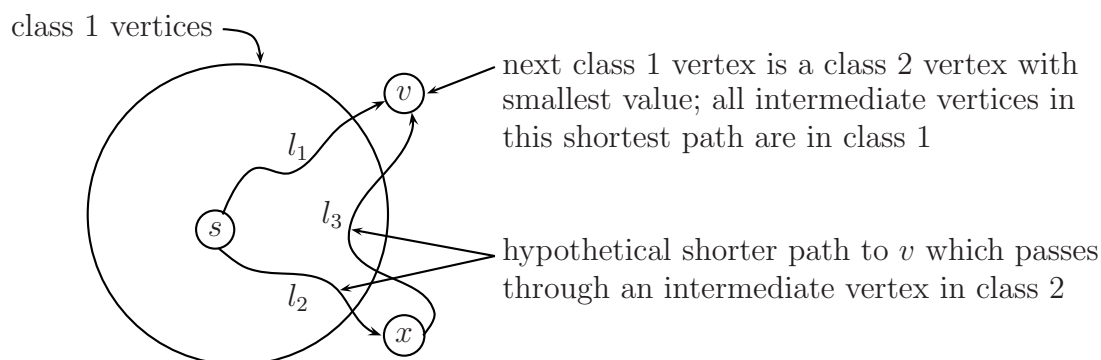


Figure 10.13 A shortest path passes through class 1 vertices only

To show that Eq.(10.1) correctly computes the shortest distance from s to v , consider Figure 10.14 which indicates the state of the computations just after u is placed in class 1 and before Eq.(10.1) is applied to vertex v in class 2. At this point, the shortest distance from s to v found so far is the shorter of $s \rightsquigarrow v$ or $s \rightsquigarrow z \rightsquigarrow v$. Let p_1 denote this path; by definition, the cost of p_1 is the current value of $d(v)$. With u in class 1, two possibly shorter paths from s to v which pass through u are $s \rightsquigarrow u \rightarrow v$ (call this p_2) and $s \rightsquigarrow u \rightarrow z \rightsquigarrow v$ (call this p_3). But p_3 cannot be shorter than p_1 , since $\delta(z) \leq \delta(u)$ (because z is an older class 1 node than u) and $w((u, z)) \geq 0$ (edge costs are nonnegative). Hence we need only to take the smaller between the cost of p_1 and p_2 as our new estimate of the shortest distance from s to v , which is precisely what Eq.(10.1) does, to wit: $d(v) = \text{minimum}[d(v), \delta(u) + w((u, v))]$.

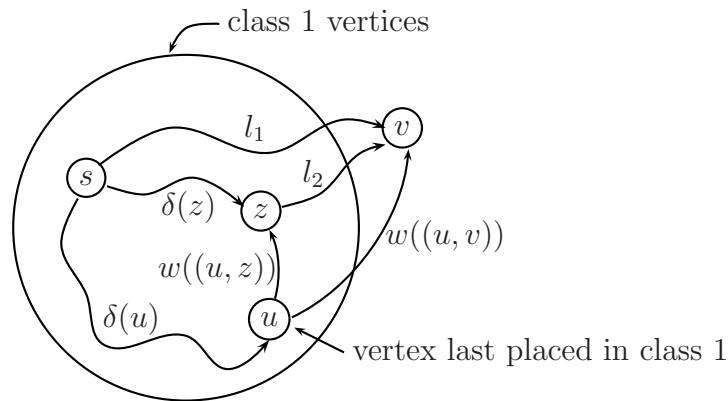


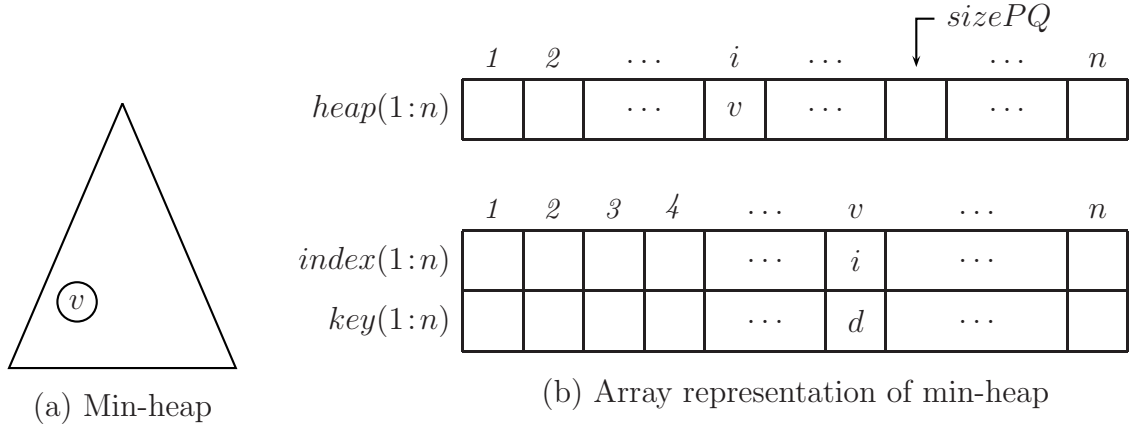
Figure 10.14 Finding the shortest path from s to v

Program implementation of Dijkstra's algorithm

The alert reader should have already noticed the striking similarities between Prim's algorithm and Dijkstra's algorithm. In terms of program implementation the only essential difference between the two is in the choice of the vertex that is placed next in U (in Prim's) or in class 1 (in Dijkstra's). In the former, it is the vertex on the fringe that is closest to T ; in the latter, it is the vertex on the fringe that is closest to s . In the implementation of Prim's algorithm as given in Procedure 10.1 we used a priority queue, represented as a binary min-heap, to maintain the vertices in $V - U$, keyed on the distance to their nearest neighbor in U . In much the same way, we will implement Dijkstra's algorithm using a priority queue, represented as a binary min-heap, to maintain the vertices in class 2, keyed on the shortest distance found thus far from the source vertex s . We will use the same data structure

$$\mathbb{PQ} = [\text{heap}(1:n), \text{index}(1:n), \text{key}(1:n), \text{sizePQ}]$$

that we utilized in implementing Prim's algorithm with the *key* array suitably modified, as shown in Figure 10.15. There are two operations on the priority queue that we will need to carry out: an *extract* operation to move a vertex from class 2 to class 1 and a *change* operation to change the priority of a vertex after an edge is relaxed to yield a shorter distance from s . These operations are implemented by the same procedures EXTRACT_MIN and DECREASE_KEY, given earlier in connection with Prim's algorithm.



n = number of vertices in G

sizePQ = current size of the min-heap (or equivalently, of the priority queue)

= n initially

v = label of a vertex in the min-heap, $1 \leq v \leq n$

i = index of the cell in array heap which contains vertex v , $1 \leq i \leq \text{sizePQ}$

$d = \text{key}(v) = \text{key}(\text{heap}(i))$ = shortest distance from s to v found thus far

Figure 10.15 Min-heap implementation of priority queue for Dijkstra's algorithm

EASY procedures to implement Dijkstra's algorithm

The EASY procedure $\text{DIJKSTRA}(\mathbb{G}, s)$ implements Dijkstra's algorithm as described above. The input to the procedure is a weighted directed graph $G = (V, E)$ and a start vertex s . The graph G is represented by the data structure $\mathbb{G} = [\text{LIST}(1:n), (\text{VRTX}, \text{COST}, \text{NEXT}), \text{pred}(1:n), \text{dist}(1:n), n]$. The first two components of \mathbb{G} comprise the cost adjacency lists representation of G . The shortest-paths tree rooted at s found by Dijkstra is returned in the array pred ; the shortest-path distances to each of the vertices $v \in V$ are returned in the array dist . These two arrays comprise the output.

Procedure DIJKSTRA invokes procedure InitPQ to initialize the priority queue which contains the class 2 vertices. Since initially every vertex is assigned a priority of ∞ except for the start vertex s which is assigned a priority of 0, any assignment of the vertices to the nodes of the min-heap is valid, provided that s is the root node.

At any stage in the computations, the fringe vertex closest to s is the vertex at the root of the min-heap; this is moved from class 2 to class 1 by a call to procedure EXTRACT_MIN , which deletes and returns the vertex at the root of the heap and restores the heap-order property via a call to procedure HEAPIFY .

After a vertex, say u , is placed in class 1, some vertex, say v , in class 2 that is adjacent to u may become closer to s via a path which includes the relaxed edge (u, v) , requiring a change in its priority. This is accomplished by the call to procedure DECREASE_KEY , which adjusts the priority of vertex v and then restores the heap-order property.

320 SESSION 10. Applications of graphs

Procedures `IsEmptyPQ`, `EXTRACT_MIN`, `HEAPIFY` and `DECREASE_KEY` which we used in implementing Prim's algorithm are once more utilized, without any modification, in implementing Dijkstra's algorithm.

```

1  procedure DIJKSTRA( $\mathbb{G}, s$ )
2  call InitPQ( $\mathbb{G}, \mathbb{PQ}, s$ )
3   $\mathbb{G}.pred \leftarrow 0$ 
4   $\triangleright$  Find shortest paths from  $s$  to every other vertex in  $V$ 
5  while not IsEmptyPQ( $\mathbb{PQ}$ ) do
6    call EXTRACT_MIN( $\mathbb{PQ}, u$ )
7    if  $\mathbb{PQ}.key(u) = \infty$  then exit
8     $\alpha \leftarrow \mathbb{G}.LIST(u)$ 
9    while  $\alpha \neq \Lambda$  do
10      $v \leftarrow \mathbb{G}.VRTX(\alpha)$ 
11      $newval \leftarrow \mathbb{PQ}.key(u) + \mathbb{G}.COST(\alpha)$ 
12     if  $\mathbb{PQ}.key(v) > newval$  then [  $\mathbb{G}.pred(v) \leftarrow u$ 
13                                     call DECREASE_KEY( $\mathbb{PQ}, v, newval$ ) ]
14      $\alpha \leftarrow \mathbb{G}.NEXT(\alpha)$ 
15   endwhile
16 endwhile
17  $\mathbb{G}.dist \leftarrow \mathbb{PQ}.key$ 
18 end DIJKSTRA

1  procedure InitPQ( $\mathbb{G}, \mathbb{PQ}, s$ )
2   $i \leftarrow 1$ 
3  for  $v \leftarrow 1$  to  $\mathbb{G}.n$  do
4    if  $v = s$  then [  $\mathbb{PQ}.heap(1) \leftarrow s$ ;  $\mathbb{PQ}.index(s) \leftarrow 1$ ;  $\mathbb{PQ}.key(s) \leftarrow 0$  ]
5    else [  $i \leftarrow i + 1$ ;  $\mathbb{PQ}.heap(i) \leftarrow v$ ;  $\mathbb{PQ}.index(v) \leftarrow i$ ;  $\mathbb{PQ}.key(v) \leftarrow \infty$  ]
6  endfor
7   $\mathbb{PQ}.sizePQ \leftarrow \mathbb{G}.n$ 
8  end InitPQ

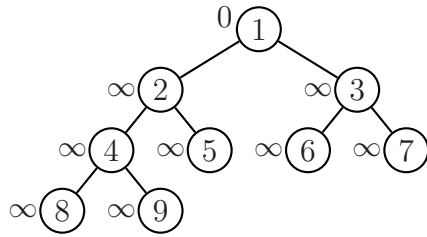
```

Procedure 10.4 Implementing Dijkstra's algorithm using a min-heap

Not all of the vertices $v \in V$ may be reachable from the source vertex s . In `DIJKSTRA` this condition is indicated when the vertex, say u , at the root of the min-heap has a priority of ∞ (line 6). This means no path exists from s to each of the vertices still in the heap, and so the loop on \mathbb{PQ} is exited. Upon exit from the loop, whether normal or premature, the *key* array contains the shortest distances from s to all the other vertices in V ; in line 16, these values are copied into the *dist* array which is returned as a component of \mathbb{G} .

Figure 10.16 depicts a partial trace of procedure `DIJKSTRA` when applied to the graph of Example 10.3. It shows the min-heap, and its internal representation, after an `EXTRACT_MIN` or `DECREASE_KEY` operation completes at selected stages of the algorithm.

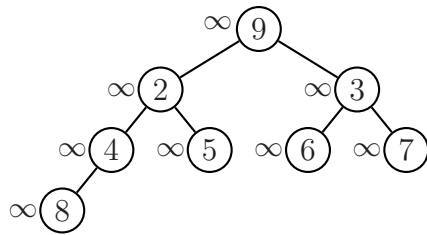
Initially: source vertex s is vertex 1



	$sizePQ \downarrow$								
	1	2	3	4	5	6	7	8	9
$heap(1:9)$	1	2	3	4	5	6	7	8	9
	1	2	3	4	5	6	7	8	9
$index(1:9)$	1	2	3	4	5	6	7	8	9
$key(1:9)$	0	∞	∞	∞	∞	∞	∞	∞	∞

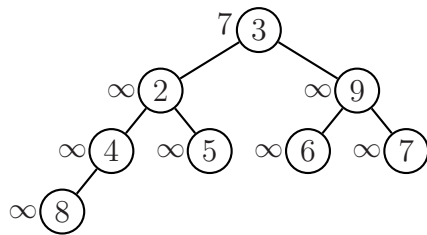
Stage 1

(a) Upon return from EXTRACT_MIN: vertex 1 placed in class 1



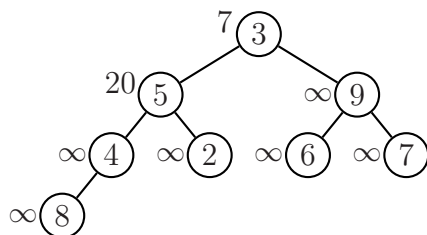
	$sizePQ \downarrow$								
	1	2	3	4	5	6	7	8	9
$heap(1:9)$	9	2	3	4	5	6	7	8	dc
	1	2	3	4	5	6	7	8	9
$index(1:9)$	dc	2	3	4	5	6	7	8	1
$key(1:9)$	0	∞	∞	∞	∞	∞	∞	∞	∞

(b) Upon return from DECREASE_KEY after edge (1,3) is relaxed: $pred(3) = 1$



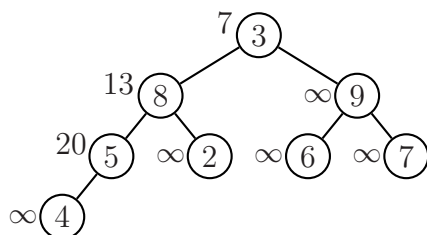
	$sizePQ \downarrow$								
	1	2	3	4	5	6	7	8	9
$heap(1:9)$	3	2	9	4	5	6	7	8	dc
	1	2	3	4	5	6	7	8	9
$index(1:9)$	dc	2	1	4	5	6	7	8	3
$key(1:9)$	0	∞	7	∞	∞	∞	∞	∞	∞

(c) Upon return from DECREASE_KEY after edge (1,5) is relaxed: $pred(5) = 1$



	$sizePQ \downarrow$								
	1	2	3	4	5	6	7	8	9
$heap(1:9)$	3	5	9	4	2	6	7	8	dc
	1	2	3	4	5	6	7	8	9
$index(1:9)$	dc	5	1	4	2	6	7	8	3
$key(1:9)$	0	∞	7	∞	20	∞	∞	∞	∞

(d) Upon return from DECREASE_KEY after edge (1,8) is relaxed: $pred(8) = 1$

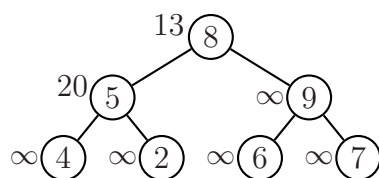


	$sizePQ \downarrow$								
	1	2	3	4	5	6	7	8	9
$heap(1:9)$	3	8	9	5	2	6	7	4	dc
	1	2	3	4	5	6	7	8	9
$index(1:9)$	dc	5	1	8	4	6	7	2	3
$key(1:9)$	0	∞	7	∞	20	∞	∞	13	∞

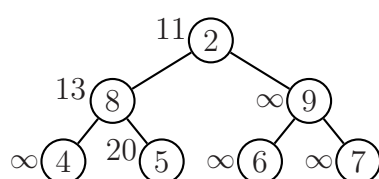
Figure 10.16 A partial trace of procedure DIJKSTRA for the graph of Example 10.3
(continued on next page)

Stage 2

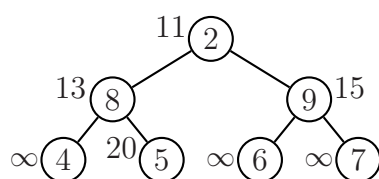
(a) Upon return from EXTRACT_MIN: vertex 3 placed in class 1



	\nwarrow sizePQ								
	1	2	3	4	5	6	7	8	9
heap(1:9)	8	5	9	4	2	6	7	dc	dc
	1	2	3	4	5	6	7	8	9
index(1:9)	dc	5	dc	4	2	6	7	1	3
key(1:9)	0	∞	7	∞	20	∞	∞	13	∞

(b) Upon return from DECREASE_KEY after edge (3,2) is relaxed: $pred(2) = 3$ 

	\nwarrow sizePQ								
	1	2	3	4	5	6	7	8	9
heap(1:9)	2	8	9	4	5	6	7	dc	dc
	1	2	3	4	5	6	7	8	9
index(1:9)	dc	1	dc	4	5	6	7	2	3
key(1:9)	0	11	7	∞	20	∞	∞	13	∞

(c) Upon return from DECREASE_KEY after edge (3,9) is relaxed: $pred(9) = 3$ 

	\nwarrow sizePQ								
	1	2	3	4	5	6	7	8	9
heap(1:9)	2	8	9	4	5	6	7	dc	dc
	1	2	3	4	5	6	7	8	9
index(1:9)	dc	1	dc	4	5	6	7	2	3
key(1:9)	0	11	7	∞	20	∞	∞	13	15

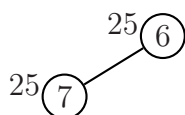
.....

Do stages 3 to 6 as an exercise.

.....

Stage 7

(a) Upon return from EXTRACT_MIN: vertex 4 placed in class 1



	\nwarrow sizePQ								
	1	2	3	4	5	6	7	8	9
heap(1:9)	6	7	dc	dc	dc	dc	dc	dc	dc
	1	2	3	4	5	6	7	8	9
index(1:9)	dc	dc	dc	dc	dc	1	2	dc	dc
key(1:9)	0	11	7	20	18	25	25	13	15

Figure 10.16 A partial trace of procedure DIJKSTRA for the graph of Example 10.3
(continued on next page)

Recovering the shortest paths found by DIJKSTRA

Procedure `DISPLAY_PATH(\mathbb{G}, s, v)` uses the *pred* and *dist* components of \mathbb{G} to construct from the former the shortest path from s to v and obtain from the latter the cost of this shortest path. If v is not reachable from s , indicated by the condition $\text{pred}(v) = 0$ or $\text{dist}(v) = \infty$, then a message ‘No path found’ is issued.

```

1  procedure DISPLAY_PATH( $\mathbb{G}, s, v$ )
2  array path(1:n)
3  len  $\leftarrow$  1
4  path(len)  $\leftarrow$   $v$ 
5  i  $\leftarrow$   $v$ 
6  while  $i \neq s$  do
7      if  $\mathbb{G}.\text{pred}(i) = 0$  then [output ‘No path found’; return]
8                          else [ $i \leftarrow \mathbb{G}.\text{pred}(i)$ ;  $\text{len} \leftarrow \text{len} + 1$ ;  $\text{path}(\text{len}) \leftarrow i$ ]
9  endwhile
10 output ‘Shortest path found:’  $\text{path}(\text{len}), \dots, \text{path}(1)$ 
11 output ‘Cost of shortest found:’  $\mathbb{G}.\text{dist}(v)$ 
12 end DISPLAY_PATH

```

Procedure 10.5 Recovering the shortest path from vertex s to vertex v

For the graph of Example 10.3, the following segment of EASY code will produce the

```

:
s  $\leftarrow$  1
for  $v \leftarrow 4$  to 7 do
    call DISPLAY_PATH( $\mathbb{G}, s, v$ )
enddo
:

```

output shown in Figure 10.18.

Shortest path found:	1 8 4
Cost of shortest path:	20
Shortest path found:	1 3 2 5
Cost of shortest path:	18
Shortest path found:	1 3 9 6
Cost of shortest path:	25
Shortest path found:	1 8 4 7
Cost of shortest path:	23

Figure 10.18 Shortest paths from vertex 1 for the graph of Example 10.3

Analysis of Dijkstra's algorithm

Let $G = (V, E)$ be weighted directed graph on $n = |V|$ vertices and $e = |E|$ edges. We analyze Dijkstra's algorithm as implemented in Procedure 10.4, i.e., using a binary min-heap as the container for the vertices in class 2, as follows.

1. Initializing the min-heap of size $sizePQ = n$ in lines 4–7 of procedure InitPQ clearly takes $O(n)$ time.
2. In an *extract* operation, the vertex at the root of the heap is deleted and is replaced by the vertex stored in the rightmost leaf at the bottommost level of the heap; restoring the heap-order property may cause this vertex to migrate back to the bottommost level in $O(\log n)$ time. At most n EXTRACT_MIN operations are performed in lines 4–5 of DIJKSTRA, taking $O(n \log n)$ time. [Actually, the heap decreases in size after every *extract* operation, but this upper bound still applies. See Eq.(7.5) in Session 7.]
3. In a *change* operation, an element at the bottommost level of the heap whose key is changed may migrate to the root of the heap, taking $O(\log n)$ time. We achieve this $\log n$ performance for DECREASE_KEY because the *index* array allows us to find in $O(1)$ time the position in the heap of the vertex whose key is decreased; this operation would otherwise take $O(n)$ time. At most e calls to DECREASE_KEY may be made in line 12 of DIJKSTRA, taking $O(e \log n)$ time.

Since any directed graph on n vertices for which the SSSP problem is non-trivially solved would have $e > n$ edges, the time complexity of Dijkstra's algorithm as implemented in Procedure 10.4 is $O(e \log n)$.

10.2.2 Floyd's algorithm for the APSP problem

Given a directed graph $G = (V, E)$ on n vertices labeled $1, 2, 3, \dots, n$ and with costs or weights assigned to the edges, Floyd's algorithm finds the cost of the shortest path between every pair of vertices in V . The basic idea behind the algorithm is to generate a series of $n \times n$ matrices $D^{(k)}, k = 0, 1, 2, \dots, n$

$$D^{(0)} \implies D^{(1)} \implies D^{(2)} \implies \dots \implies D^{(k-1)} \implies D^{(k)} \implies \dots \implies D^{(n)}$$

with elements defined as follows:

$$d_{ij}^{(k)} = \text{cost of the shortest path from vertex } i \text{ to vertex } j \text{ which goes through no intermediate vertex with a label greater than } k$$

Put another way, the *allowable* intermediate vertices in the shortest path from i to j whose cost is $d_{ij}^{(k)}$ are vertices $1, 2, \dots, k$ only. By this definition, $d_{ij}^{(0)}$ is the cost of the edge (i, j) , or ∞ if there is no such edge. Hence, $D^{(0)}$ is simply the cost adjacency matrix for the graph, as defined in section 9.2 of Session 9. With $D^{(0)}$ as the starting matrix, each successive matrix is then generated using the iterative formula

$$d_{ij}^{(k)} = \text{minimum}[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}], \quad 1 \leq i, j \leq n \quad (10.2)$$

For any given pair of vertices i and j , the iterative application of Eq.(10.2) is equivalent to systematically considering the other vertices for inclusion in the path from vertex i to vertex j . If at the k th iteration, including vertex k in the path from i to j results in a shorter path, then the cost of this shorter path is taken as the k th iteration value of the cost of the path from i to j . Clearly, the n th iteration value of this cost is the cost of the shortest path from vertex i to vertex j .

Let C be the cost adjacency matrix for a weighted, directed graph G ; the following algorithm formalizes the process just described.

Floyd's algorithm

1. [Initialize] $D^{(0)} \leftarrow C$
2. [Iterate] Repeat for $k = 1, 2, 3, \dots, n$

$$d_{ij}^{(k)} \leftarrow \text{minimum}[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}], \quad 1 \leq i, j \leq n$$

Then, $D^{(n)}$ contains the cost of the shortest path between every pair of vertices i and j in G .

Example 10.4. Floyd's algorithm to solve the APSP problem

Consider the graph shown in Figure 10.19 along with its cost adjacency matrix C , which is $D^{(0)}$. We view the entry in row i and column j , $i \neq j$, of $D^{(0)}$ as the cost of the *shortest path* from i to j in which there is no intermediate vertex in the path, i.e., the path is an edge. If there is no edge (i, j) then the cost of the path is ∞ .

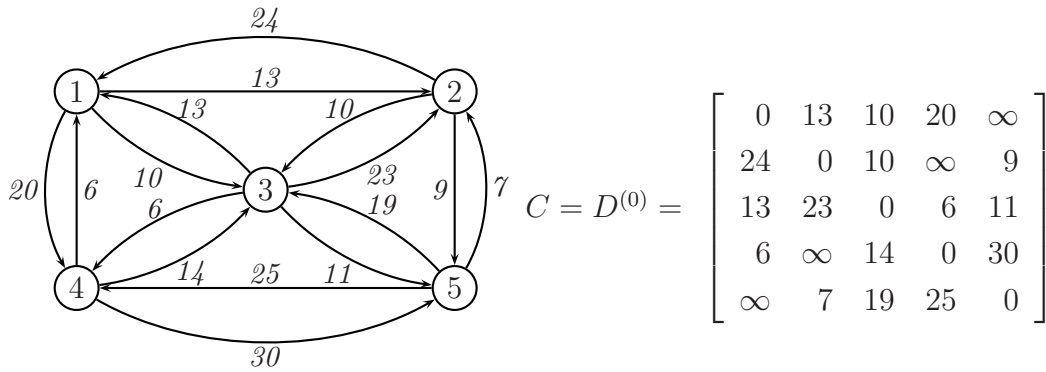


Figure 10.19 Graph of Example 10.4 and its cost adjacency matrix

To find a possibly shorter path from i to j we now allow vertex 1 to be an intermediate vertex in the path. Applying Eq.(10.2) we have, for instance

$$\begin{aligned} d_{23}^{(1)} &= \min[d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)}] = \min[10, 24 + 10] = 10 \\ d_{24}^{(1)} &= \min[d_{24}^{(0)}, d_{21}^{(0)} + d_{14}^{(0)}] = \min[\infty, 24 + 20] = 44 \\ d_{42}^{(1)} &= \min[d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)}] = \min[\infty, 6 + 13] = 19 \end{aligned} \quad D^{(1)} = \begin{bmatrix} 0 & 13 & 10 & 20 & \infty \\ 24 & 0 & 10 & 44 & 9 \\ 13 & 23 & 0 & 6 & 11 \\ 6 & 19 & 14 & 0 & 30 \\ \infty & 7 & 19 & 25 & 0 \end{bmatrix}$$

Including vertex 1 in the path from vertex 2 to vertex 3 yields a higher cost, so we retain the current shortest path with a cost of 10. Including 1 in the path from 2 to 4 yields a lower cost, so we take the path $2 \rightarrow 1 \rightarrow 4$ with a cost of 44. And so on. The matrix on the right tabulates the costs of the shortest paths found by Floyd with vertex 1 allowed as an intermediate vertex.

Next we allow vertex 2 to be an intermediate vertex also. Applying Eq.(10.2) we have, for instance

$$\begin{aligned} d_{45}^{(2)} &= \min [d_{45}^{(1)}, d_{42}^{(1)} + d_{25}^{(1)}] = \min [30, 19 + 9] = 28 \\ d_{51}^{(2)} &= \min [d_{51}^{(1)}, d_{52}^{(1)} + d_{21}^{(1)}] = \min [\infty, 7 + 24] = 31 \\ d_{53}^{(2)} &= \min [d_{53}^{(1)}, d_{52}^{(1)} + d_{23}^{(1)}] = \min [19, 7 + 10] = 17 \end{aligned} \quad D^{(2)} = \begin{bmatrix} 0 & 13 & 10 & 20 & 22 \\ 24 & 0 & 10 & 44 & 9 \\ 13 & 23 & 0 & 6 & 11 \\ 6 & 19 & 14 & 0 & 28 \\ 31 & 7 & 17 & 25 & 0 \end{bmatrix}$$

With 1 and 2 as allowable intermediate vertices, we find a shorter path from 4 to 5, namely, the path from 4 to 2 which passes thru 1 found in the previous iteration plus the edge from 2 to 5, with a total cost of 28. Including 2 in the path from 5 to 3 yields also a shorter path with a cost of 17. And so on. The matrix on the right tabulates the costs of the shortest paths found by Floyd with vertices 1 and 2 allowed as intermediate vertices.

By successively allowing vertices 3, 4 and finally, vertex 5 as intermediate vertices, we obtain shorter and shorter and finally the shortest paths between every pair of vertices in the graph, whose costs comprise the final matrix $D^{(5)}$.

Some notes regarding Floyd's algorithm

1. Floyd's algorithm is based on an algorithm design technique called **dynamic programming**, a method typically applied to optimization problems. This is a method in which an optimal solution to a problem is obtained by combining optimal solutions to subproblems in a bottom-up fashion. Solutions to smaller subproblems are recorded in a table so that they need not be recomputed when they are combined to obtain solutions to larger subproblems. The word 'programming' in 'dynamic programming' refers to this tabular procedure, not to computer programming. In the above example, the entries in the matrix $D^{(1)}$ are optimal solutions to the subproblem of finding the cost of the shortest path between every pair of vertices in G in which only vertex 1 is allowed as an intermediate vertex. Similarly, the entries in the matrix $D^{(2)}$ are optimal solutions to the larger subproblem of finding the cost of the shortest path between every pair of vertices in G in which 1 and 2 are allowed as intermediate vertices. The solutions tabulated in $D^{(2)}$ are obtained from the solutions tabulated in $D^{(1)}$, in true dynamic programming fashion, as the sample computations given above clearly show. And so on, until we finally obtain $D^{(5)}$ which contains the final smallest costs.
2. Floyd's algorithm, unlike Dijkstra's, is applicable to a graph G with negative weights or costs on the edges, provided that there are no negative-cost cycles in G .
3. Floyd's algorithm, like Dijkstra's, records only the cost of the shortest paths found, but not the actual paths.

Encoding the shortest paths

To be able to reconstruct the shortest paths found by Floyd's algorithm, we will maintain an $n \times n$ matrix $pred^{(k)}$, $k = 0, 1, 2, \dots, n$ with elements defined as follows:

$$\begin{aligned}
 pred_{ij}^{(0)} &= 0 && \text{if } i = j \text{ or if there is no edge } (i, j) \text{ in } E \\
 &= i && \text{if there is an edge } (i, j) \text{ in } E \\
 pred_{ij}^{(k)} &= pred_{kj}^{(k-1)} && \text{if including } k \text{ as an intermediate vertex yields a} \\
 &&& \text{shorter path from } i \text{ to } j \\
 &= pred_{ij}^{(k-1)} && \text{otherwise}
 \end{aligned}$$

At termination of Floyd's algorithm, $pred_{ij}^{(n)}$ is the immediate predecessor of vertex j in a shortest path from vertex i . Thus each row i of the final $pred$ matrix encodes a shortest-paths tree rooted at vertex i , in much the same way that the $pred$ array in Dijkstra's algorithm encodes a shortest-paths tree rooted at the source vertex s .

EASY procedure to implement Floyd's algorithm

The EASY procedure FLOYD(\mathbb{G}) implements Floyd's algorithm as described above. The input to the procedure is a weighted directed graph $G = (V, E)$. The graph G is represented by the data structure $\mathbb{G} = [C(1:n, 1:n), D(1:n, 1:n), pred(1:n, 1:n), n]$. The first component of \mathbb{G} is the cost adjacency matrix for G . The shortest paths found by Floyd are encoded in $pred$ and the corresponding shortest-path distances are stored in D . These two matrices comprise the output.

```

1  procedure FLOYD( $\mathbb{G}$ )
▷ Initializations
2     $\mathbb{G}.D \leftarrow \mathbb{G}.C$ 
3    for  $i \leftarrow 1$  to  $\mathbb{G}.n$  do
4      for  $j \leftarrow 1$  to  $\mathbb{G}.n$  do
5        if  $j = i$  or  $\mathbb{G}.C(i, j) = \infty$  then  $\mathbb{G}.pred(i, j) = 0$ 
6        else  $\mathbb{G}.pred(i, j) = i$ 
7      endfor
8    endfor
▷ Floyd's algorithm
9    for  $k \leftarrow 1$  to  $\mathbb{G}.n$  do
10     for  $i \leftarrow 1$  to  $\mathbb{G}.n$  do
11       for  $j \leftarrow 1$  to  $\mathbb{G}.n$  do
12          $dijk \leftarrow \mathbb{G}.D(i, k) + \mathbb{G}.D(k, j)$ 
13         if  $dijk < \mathbb{G}.D(i, j)$  then [ $\mathbb{G}.D(i, j) \leftarrow dijk$ ;  $\mathbb{G}.pred(i, j) \leftarrow \mathbb{G}.pred(k, j)$ ]
14       endfor
15     endfor
16   endfor
17 end FLOYD

```


Figure 10.20 shows the $D^{(k)}$ and $pred^{(k)}$ matrices generated by Procedure 10.5 at the end of each iteration of Floyd's algorithm for the graph of Example 10.4. The italicized values in $D^{(k)}$ are new costs of shorter paths found by including k as an intermediate vertex, as indicated by the corresponding italicized entry in $pred^{(k)}$. Note that Procedure 10.5 utilizes only one D matrix; the computations prescribed by Eq.(10.2) are done *in place* since we always have $d_{ik}^{(k)} = d_{ik}^{(k-1)}$ and $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Likewise only one matrix $pred$ is used, with its elements updated in place as needed.

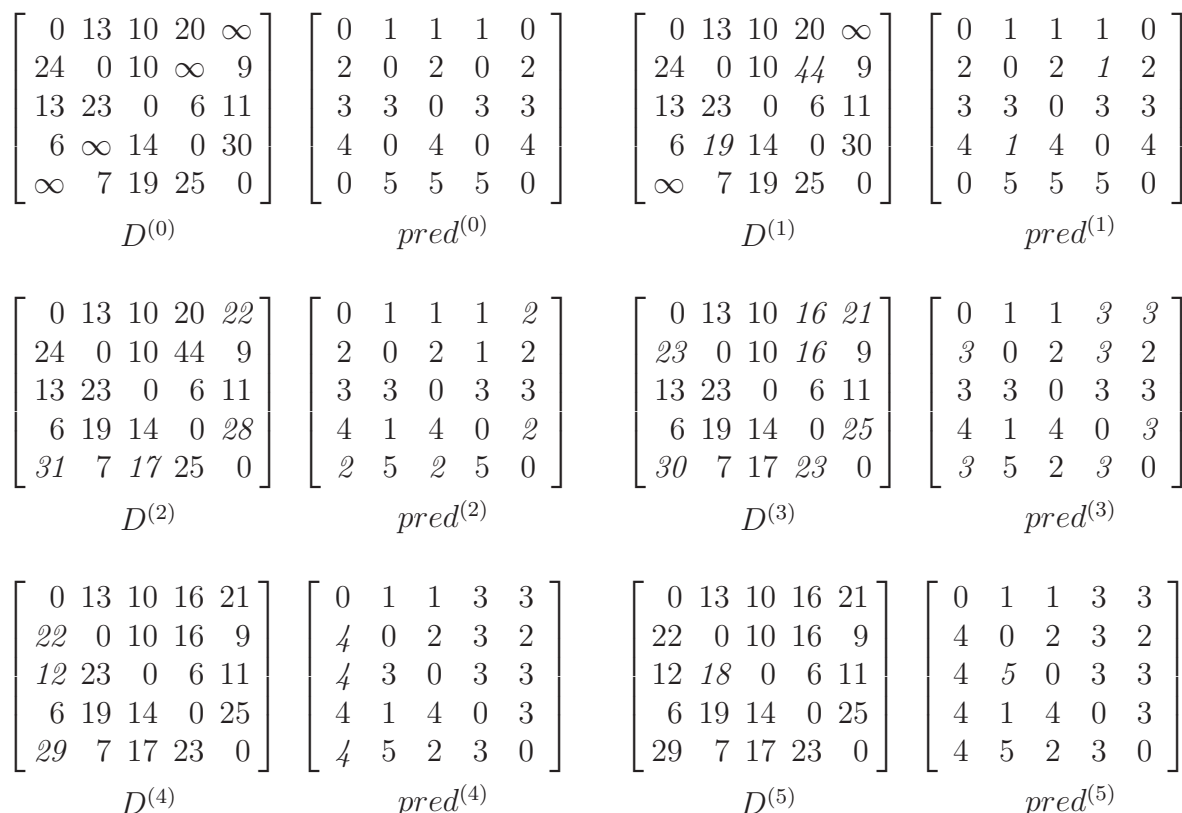


Figure 10.20 The sequence of $D^{(k)}$ and $pred^{(k)}$ matrices for the graph of Example 10.4

As we have earlier pointed out, each row i of the final $pred$ matrix encodes a shortest-paths tree rooted at i . Figure 10.21 shows the five shortest-paths trees found by Floyd for the graph of Example 10.4, as encoded in $pred^{(5)}$. The number placed alongside a vertex is the cost of the shortest path from the root to that vertex, as recorded in $D^{(5)}$.

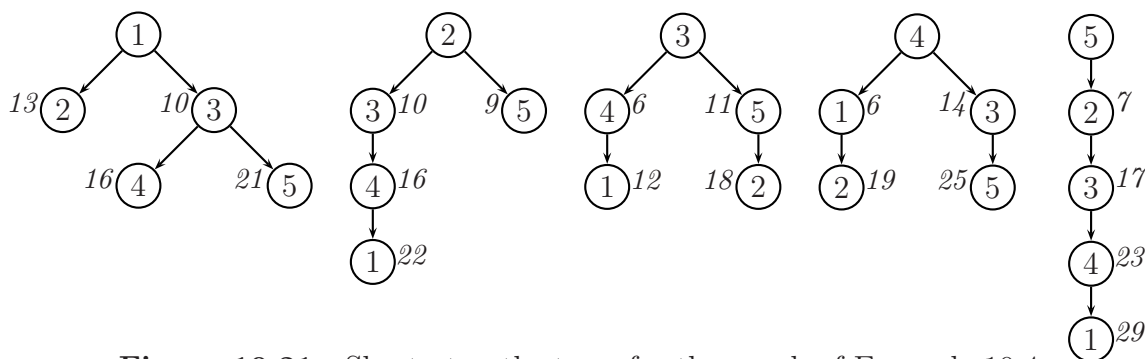


Figure 10.21 Shortest-paths trees for the graph of Example 10.4

Recovering the shortest paths found by FLOYD

Procedure `DISPLAY_PATH(\mathbb{G}, i, j)` uses the *pred* and *D* components of \mathbb{G} to construct from the former the shortest path from *i* to *j* and obtain from the latter the cost of this shortest path. If *j* is not reachable from *i*, indicated by the condition $\mathbb{G}.pred(i, j) = 0$ or $\mathbb{G}.D(i, j) = \infty$, then a message ‘No path found’ is issued.

```

1  procedure DISPLAY_PATH( $\mathbb{G}, i, j$ )
2  array path(1:n)
3  len  $\leftarrow$  1
4  path(len)  $\leftarrow$  j
5  k  $\leftarrow$  j
6  while k  $\neq$  i do
7      if  $\mathbb{G}.pred(i, k) = 0$  then [output ‘No path found’; return]
8      else [k  $\leftarrow$   $\mathbb{G}.pred(i, k)$ ; len  $\leftarrow$  len + 1; path(len)  $\leftarrow$  k]
9  endwhile
10 output ‘Shortest path found:’ path(len), ..., path(1)
11 output ‘Cost of shortest found:’  $\mathbb{G}.D(i, j)$ 
12 end DISPLAY_PATH

```

Procedure 10.7 Recovering the shortest path from vertex *i* to vertex *j*

The following segment of EASY code will output the shortest path between every pair of vertices in *G*, along with its corresponding cost in the manner shown in Figure 10.18.

```

:
for i  $\leftarrow$  1 to  $\mathbb{G}.n$  do
    for j  $\leftarrow$  1 to  $\mathbb{G}.n$  do
        if j  $\neq$  i then call DISPLAY_PATH( $\mathbb{G}, i, j$ )
    endfor
endfor
:

```

Analysis of Floyd’s algorithm

Let $G = (V, E)$ be a weighted directed graph on $n = |V|$ vertices and $e = |E|$ edges. Floyd’s algorithm as implemented in Procedure 10.6 clearly takes $O(n^3)$ time and $O(n^2)$ space to solve the APSP problem for *G*. Despite its n^3 running time, the simplicity of the code makes Floyd’s algorithm attractive if the graph is dense and not too large. If *G* is large and sparse, invoking Dijkstra’s algorithm, as implemented in Procedure 10.4, *n* times to solve the APSP problem in $O(ne \log n)$ time would be the better alternative.

10.2.3 Warshall's algorithm and transitive closure

Let $G = (V, E)$ be a directed graph on $n = |V|$ vertices labeled $1, 2, \dots, n$. We define the **transitive closure** of G as the graph $G^+ = (V, E^+)$ such that (i, j) is an edge in E^+ if there is a path of length at least one from vertex i to vertex j in G . The term 'transitive closure' is used interchangeably to refer to G^+ or to its adjacency matrix, say T .

Let A denote the adjacency matrix of G . The problem of generating T from A is closely related to the problem of generating D from C in Floyd's algorithm. The algorithm to generate T from A is called **Warshall's algorithm**, and is, in fact, an older algorithm than Floyd's. The basic idea behind the algorithm is to generate a series of $n \times n$ matrices $T^{(k)}, k = 0, 1, 2, \dots, n$

$$T^{(0)} \implies T^{(1)} \implies T^{(2)} \implies \dots \implies T^{(k-1)} \implies T^{(k)} \implies \dots \implies T^{(n)}$$

with elements defined as follows:

$$\begin{aligned} t_{ij}^{(k)} &= 1 \text{ (true) if there is a path of length } \geq 1 \text{ from vertex } i \text{ to vertex } j \text{ in } \\ &\quad G \text{ which goes through no intermediate vertex with a label} \\ &\quad \text{greater than } k \\ &= 0 \text{ (false) otherwise} \end{aligned}$$

By this definition, $T^{(0)}$ is adjacency matrix A for G , as defined in section 9.2 of Session 9. With $T^{(0)}$ as the starting matrix, each successive matrix is then generated using the iterative formula

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ or } (t_{ik}^{(k-1)} \text{ and } t_{kj}^{(k-1)}) \quad 1 \leq i, j \leq n \quad (10.3)$$

For every pair of vertices i and j , the iterative application of Eq.(10.3) simply tests whether there is a path from vertex i to vertex j if vertices $1, 2, \dots, n$ are successively considered for inclusion as intermediate vertices in the path. Specifically, at the k th iteration, we test if there is a path from i to j which contains no intermediate vertices with label greater than $k-1$, or if there is a path from i to k and a path from k to j which contain no intermediate vertices with label greater than $k-1$. If so, then there must be a path from i to j which contains no intermediate vertices with label greater than k . In the course of the iterations, t_{ij} may be repeatedly set to **true**. If there is no edge from i to j and a path is still not found after vertices $1, 2, \dots, n$ are allowed as intermediate vertices, then t_{ij} remains **false**.

Let A be the adjacency matrix for a directed graph G ; the following algorithm formalizes the process just described.

Warshall's algorithm

1. [Initialize] $T^{(0)} \leftarrow A$
2. [Iterate] Repeat for $k = 1, 2, 3, \dots, n$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ or } (t_{ik}^{(k-1)} \text{ and } t_{kj}^{(k-1)})$$

Then, $T^{(n)}$ is the transitive closure of A , or equivalently, of G .

EASY procedure to implement Warshall's algorithm

The EASY procedure WARSHALL(\mathbb{G}) implements Warshall's algorithm as described above. The input to the procedure is a directed graph $G = (V, E)$. The graph G is represented by the data structure $\mathbb{G} = [A(1:n, 1:n), T(1:n, 1:n), n]$. The first component of \mathbb{G} is the adjacency matrix for G . The second component is the transitive closure of G found by Warshall; this comprise the output.

```

1  procedure WARSHALL( $\mathbb{G}$ )
2   $\mathbb{G}.T \leftarrow \mathbb{G}.A$ 
3  for  $k \leftarrow 1$  to  $\mathbb{G}.n$  do
4    for  $i \leftarrow 1$  to  $\mathbb{G}.n$  do
5      for  $j \leftarrow 1$  to  $\mathbb{G}.n$  do
6         $\mathbb{G}.T(i, j) \leftarrow \mathbb{G}.T(i, j)$  or ( $\mathbb{G}.T(i, k)$  and  $\mathbb{G}.T(k, j)$ )
7      endfor
8    endfor
9  endfor
10 end WARSHALL

```

Procedure 10.8 Implementing Warshall's algorithm

Example 10.5. Warshall's algorithm to find the transitive closure of a directed graph

Figure 10.22 shows the sequence of matrices $T^{(k)}$ generated by Procedure 10.8, culminating in the transitive closure $T = T^{(5)}$ of the input graph G . Note that the computations are done in-place; thus only one matrix T is actually used.

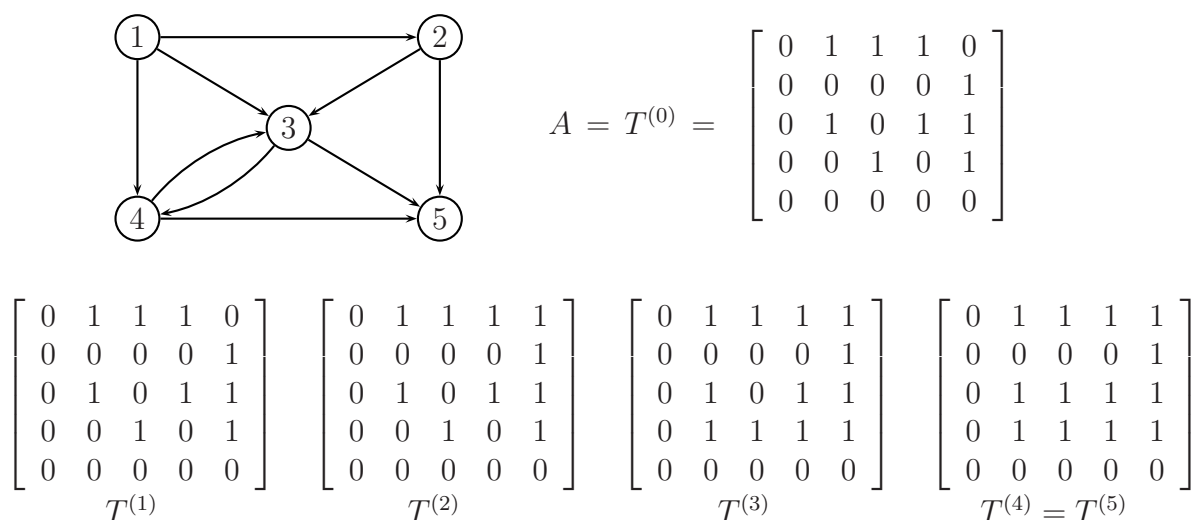


Figure 10.22 Finding the transitive closure of a directed graph

Analysis of Warshall's algorithm

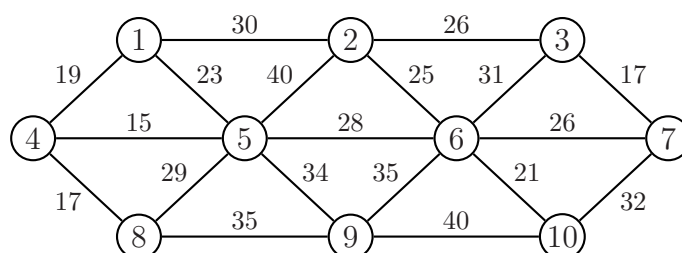
For a directed graph on n vertices, Warshall's algorithm clearly executes in $O(n^3)$ time and $O(n^2)$ space.

Summary

- Despite the apparent dissimilarity of the problems they are designed to solve, Prim's algorithm (for finding a minimum cost spanning tree for a weighted undirected graph) and Dijkstra's algorithm (for finding shortest paths in a weighted directed graph) are closely related. The template for both algorithms is breadth first search and the algorithmic design paradigm is the greedy technique.
- An interesting feature of the implementation of Kruskal's algorithm is how the concept of equivalence classes is utilized to detect possible cycles in the spanning tree being constructed.
- Although Dijkstra's algorithm can be used to solve the all-pairs-shortest-paths problem for directed graphs, Floyd's algorithm which is based on another algorithmic design paradigm called dynamic programming is more commonly used since it requires so much less implementation overhead.
- Floyd's algorithm is sometimes referred to in the CS literature as Floyd-Warshall's algorithm because of its close affinity to Warshall's algorithm for finding the transitive closure of a directed graph. The latter actually predates the former.

Exercises

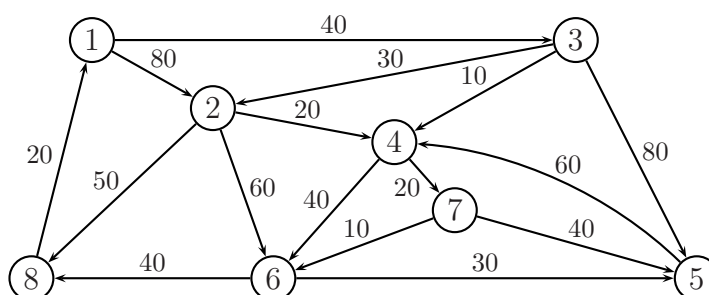
1. Find a minimum cost spanning tree for the graph shown below using Prim's algorithm. Show the solution as in Example 10.1 (Figure 10.3).



2. Assume that Prim's algorithm is applied as implemented in Procedure 10.1 to find a minimum cost spanning tree for the graph in Item 1 and that vertex 1 is the start vertex. Show the priority queue, as defined by the *heap*, *index* and *keys* arrays, after the:
 - (a) first call to EXTRACT_MIN
 - (b) call to DECREASE_KEY for vertex 2
 - (c) call to DECREASE_KEY for vertex 4
 - (d) call to DECREASE_KEY for vertex 5
3. Using a language of your choice, transcribe procedure PRIM (Procedure 10.1) into a running program. Test your program using the graph in Example 10.1 and in Item 1.

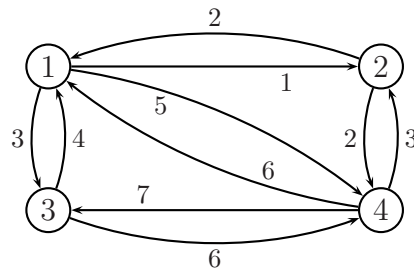
334 SESSION 10. Applications of graphs

4. Find a minimum cost spanning tree for the graph in Item 1 using Kruskal's algorithm. Show the solution as in Example 10.2 (Figure 10.8).
5. Assume that Kruskal's algorithm is applied as implemented in Procedure 10.3 to find a minimum cost spanning tree for the graph in Item 1. Show the priority queue, as defined by the *heap* and *u - v - key* arrays:
 - (a) as initially constituted
 - (b) after the first edge has been selected and placed in T
 - (c) after the second edge has been selected and placed in T
 - (d) after the third edge has been selected and placed in T
6. The advantage of *not* presorting all the edges of a graph by edge cost before applying Kruskal's algorithm is that we may not have to examine all the edges in order to find a minimum cost spanning tree for the graph (which means sorting them all is wasted effort). Give an example of a graph in which all edges are in fact examined before Kruskal's algorithm terminates.
7. Using a language of your choice, transcribe procedure KRUSKAL (Procedure 10.3) into a running program. Test your program using the graph in Example 10.2 and in Item 1.
8. Solve the SSSP problem with vertex 1 as the source vertex for the graph shown below using Dijkstra's algorithm. Show the solution as in Example 10.3 (Figure 10.11).

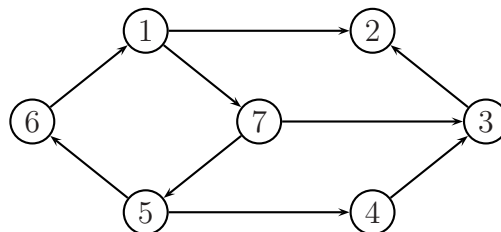


9. Assume that Dijkstra's algorithm is applied as implemented in Procedure 10.4 to solve the SSSP problem for the graph in Item 8. Show the priority queue, as defined by the *heap*, *index* and *keys* arrays, after the:
 - (a) first call to EXTRACT_MIN
 - (b) call to DECREASE_KEY after edge (1,2) is relaxed
 - (c) call to DECREASE_KEY after edge (1,3) is relaxed
 - (d) second call to EXTRACT_MIN
10. Using a language of your choice, transcribe procedure DIJKSTRA (Procedure 10.4) into a running program. Test your program using the graph in Example 10.3.

11. Using your program in Item 10, solve the APSP problem for the graph in Item 8 by taking each of the vertices $1, 2, \dots, 8$ in turn as the start vertex. Using the *pred* and *dist* arrays returned by DIJKSTRA, construct the shortest-paths tree rooted at each vertex.
12. Solve the APSP problem for the graph shown below using Floyd's algorithm.



13. Using a language of your choice, transcribe procedure FLOYD (Procedure 10.5) into a running program. Test your program using the graph in Example 10.4.
14. Using your program in Item 13, solve the APSP problem for the graph in Item 8. Using the *D* and *pred* arrays returned by FLOYD, construct the shortest-paths trees rooted at each of the vertices $1, 2, \dots, 8$. Compare these with the shortest-paths trees obtained in Item 11.
15. Find the transitive closure of the graph shown below.



16. Using a language of your choice, transcribe procedure WARSHALL (Procedure 10.8) into a running program. Test your program using the graph in Example 10.5 and in Item 15.

Bibliographic Notes

The five algorithms discussed in this session, viz., Prim's, Kruskal's, Dijkstra's, Floyd's and Warshall's, are all classical algorithms and can be found in most textbooks on Data Structures or on Algorithms. CORMEN[2001] gives a thorough treatment of all five algorithms.

A proof of Cayley's theorem on the number of spanning trees on n distinct vertices is given in EVEN[1979], pp. 27–29.

A comparative study of different implementations of Prim's and Kruskal's algorithms is given in MORET[1991], pp. 285–288. The results, based on actual computer runs on very sparse graphs ($e \ll n^2$) to pretty dense graphs ($e = n^{\frac{3}{2}}$), show that Prim's

algorithm, implemented using a binary heap (as in Procedure 10.1), is the algorithm of choice.

The description of Dijkstra's algorithm, with the idea of 'class' and 'value' assigned to vertices, is from LEWIS[1976], pp. 209–210.

The proof of the MST theorem, on which Prim's and Kruskal's algorithms are based, is from AHO[1983], p. 234 (where it is referred to as the MST property); a similar proof can be found in EVEN[1979], p. 25 (where it is referred to as Theorem 2.3).

The proof of the correctness of Dijkstra's algorithm is from AHO[1983], pp. 206–207. A more rigorous correctness proof is given in CORMEN[2001], pp. 597–598, as Theorem 24.6.

SESSION 11

Linear lists

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define a list and explain common list operations.
2. Discuss implementation issues pertaining to the sequential and linked representation of lists.
3. Describe different ways of constituting linked linear lists and discuss the relative merits of each one.
4. Assess the suitability of using linear lists, and which variety of list to use, for a given application.
5. Describe how polynomial arithmetic is implemented on a computer using linked lists.
6. Describe how multiple-precision integer arithmetic is implemented on a computer using linked lists.
7. Write EASY procedures to complement those already given for polynomial arithmetic and multiple-precision integer arithmetic.
8. Explain the mechanics of dynamic storage management and the role of lists in implementing DSM.
9. Describe, and explain the relative merits of, the various sequential-fit and buddy-system techniques for reservation and liberation in DSM.
10. Describe how linked lists are utilized in the selection phase of UPCAT processing.

READINGS KNUTH1[1997], pp. 273–281, 435–452; KNUTH2[1998], pp. 265–278; STANDISH[1980], pp. 248–274; HOROWITZ[1976], pp. 140–155; TREMBLAY[1976], pp. 283–291; TENENBAUM[1986], pp. 693–717.

DISCUSSION

In this and the next session we will consider the list ADT. Lists are most useful in applications in which the nature of the information to be processed necessitates a structure

that can change in size or shape or both. When it is difficult to predict, before runtime, the amount of storage needed for the data, lists provide an efficient and elegant way of managing memory. Similarly, when it is needed to nest structures within structures, implying changes not only in length but also in depth, lists are the natural building blocks.

A **list** is a finite, ordered set of zero or more elements, e.g., $L = (x_1, x_2, x_3, \dots, x_n)$. A list is said to be empty or null if it has no element, i.e., if $n = 0$. The elements of a list may be *atoms* or *lists*, where an atom is assumed to be distinguishable from a list.

A **linear list** is a list in which all the elements are atoms. As its name implies, the essential property of a linear list is the linear ordering of its elements. Thus, in the linear list $L = (x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}, x_n)$, we say that x_1 is the first element, x_n is the last element, and the element x_i is preceded by x_{i-1} and followed by x_{i+1} , for $1 < i < n$. The ordering imposed on the elements of a list is governed by considerations specific to a given application.

A list in which some of the elements are lists is called a **generalized list** or a **list structure**. For instance, if the letters a, b, c, d, \dots denote atoms, then the list $L = ((a, b, (c, ())), d, (), (e, ()), (f, (g, (a))), d)$ is a generalized list consisting of four elements x_1, x_2, x_3 and x_4 , where x_1 is the list $(a, b, (c, ()))$, x_2 is the atom d , x_3 is the null list $()$, and x_4 is the list $(e, ()), (f, (g, (a))), d)$. The list $(a, b, (c, ()))$ consists of the atoms a and b and the list $(c, ())$. The list $(c, ())$ consists of the atom c and the null list $()$. We can decompose element x_4 in a similar manner. It is clear that a generalized list has both *length* and *depth*.

In this session we will consider various species of linear lists and study a number of interesting applications of such lists. We will tackle generalized lists, which are a different sort of specie altogether, in the next session.

11.1 Linear Lists

The following is a list of some common operations on linear lists.

1. Initialize a list.
2. Determine whether a list is empty.
3. Find the length, say n , of a list, i.e., the number of elements in the list.
4. Gain access to the i th element of a list, $1 \leq i \leq n$.
5. Replace the i th element of a list.
6. Delete the i th element of a list.
7. Insert a new element into a list.
8. Combine two or more lists into a single list.
9. Split a list into two or more lists.
10. Make a copy of a list.

11. Erase a list.
12. Search a list.
13. Sort a list.

As with any ADT, a list may be represented in the memory of a computer using either sequential allocation or linked allocation. Which representation to use usually depends on the type and mix of list operations required in a particular application. It is not uncommon to have applications in which both representations are used to take full advantage of the merits of each.

11.2 Sequential representation of linear lists

Figure 10.1 shows the list $L = (x_1, x_2, x_3, \dots, x_n)$ stored in an array of size m . If the list grows such that n exceeds m , we get an *overflow* condition. If n remains way below m , possibly because we have over-allocated to avert overflow, space is wasted. With sequential representation, striking a balance between avoiding overflow and conserving space may not be easy in applications where lists grow unpredictably.

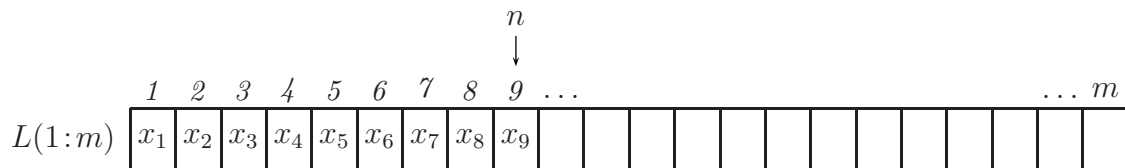


Figure 11.1 Array implementation of a linear list

One advantage of the sequential representation of a linear list is that gaining access to the i th element to select or replace the element can be done in $O(1)$ time through standard array indexing. However, many of the operations listed above cannot be performed with similar efficiency. For example, inserting a new element before the i th element entails moving the n th down to the i th elements one cell *up* to give room for the new element. Deleting the i th element, on the other hand, entails moving the $(i+1)$ th through the n th elements one cell *down* to fill the gap left by the deleted element. If gaps are not filled up, random access through standard array indexing will no longer work and we lose the principal advantage of sequential representation. Thus insertion and deletion take $O(n)$ time.

Unless a list is fairly *static*, with random access to the i th element as the predominant operation, linked representation is usually the better alternative.

11.3 Linked representation of linear lists

There are several variations on the theme of linear lists represented as a collection of linked nodes. For instance, the list may be singly-linked or doubly-linked, ‘straight’ or ‘circular’, with or without a list head. As expected, the way a list is implemented as a particular data structure depends primarily on the operations performed on the list in a given application. We will now consider some common implementations of a linked linear list.

11.3.1 Straight singly-linked linear list

Figure 11.2 shows a singly-linked linear list with node structure $(DATA, LINK)$, and with the $LINK$ field of the last node set to Λ . The condition $l = \Lambda$ means the list is empty. We have already studied insertion and deletion routines for this particular implementation of a list in connection with linked stacks in Session 4 and linked queues and dequeues in Session 5. You may want at this point to review the procedures for insertion and deletion for these data structures. Bear in mind, however, that insertion into a list may be at any arbitrary position in the list, and that any element of a list may be deleted regardless of its position in the list.

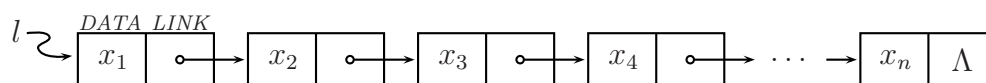


Figure 11.2 Straight singly-linked linear list

The following EASY procedures implement the basic insert and delete operations for the list of Figure 11.2, represented by the data structure $\mathbb{L} = [(DATA, LINK), l]$. The first component of \mathbb{L} specifies the node structure of each node in the list; the second component l is the pointer to the list.

```

1  procedure LIST_INSERT( $\mathbb{L}, w, x$ )
  ▷ Inserts a new element  $w$  before element  $x$  in the list; if there is no element  $x$  in the
  ▷ list,  $w$  is inserted at the tail end of the list.
2  call GETNODE( $\nu$ )
3   $DATA(\nu) \leftarrow w$ 
4   $\alpha \leftarrow l$ 
5  loop
6    if  $\alpha = \Lambda$  or  $DATA(\alpha) = x$  then [if  $\alpha = l$  then  $l \leftarrow \nu$ 
7                                          else  $LINK(\beta) \leftarrow \nu$ 
8                                           $LINK(\nu) \leftarrow \alpha$ ; return]
9                                          else [ $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow LINK(\alpha)$ ]
10 forever
11 end LIST_INSERT

```

Procedure 11.1 Insertion into a straight singly-linked linear list

This is the first of a good number of EASY procedures on lists that you will encounter in this session. You may want to invest a little more effort tracking its logic by tracing its actions on sample lists, including the null list. Verify that the procedure correctly handles the different cases that may arise, namely:

1. $\alpha = l = \Lambda$ — insertion into an initially empty list
2. $\alpha = l \neq \Lambda$, $DATA(\alpha) = x$ — insertion at the head of the list
3. $\alpha \neq l \neq \Lambda$, $DATA(\alpha) = x$ — insertion within the list
4. $\alpha = \Lambda \neq l$ — insertion at the tail end of the list

Take note that the **or** operator in line 6 is assumed to be *short-circuited*: if the first condition is true, the second condition is *not* evaluated.

The time complexity of the procedure is clearly $O(n)$, where n is the length of the list.

```

1  procedure LIST_DELETE( $\mathbb{L}, x$ )
▷ Deletes element  $x$  from the list
2     $\alpha \leftarrow l$ 
3    while  $\alpha \neq \Lambda$  do
4      if  $DATA(\alpha) = x$  then [if  $\alpha = l$  then  $l \leftarrow LINK(l)$ 
5                                else  $LINK(\beta) \leftarrow LINK(\alpha)$ 
6                                call RETNODE( $\alpha$ ); return]
7                                else [ $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow LINK(\alpha)$ ]
8    endwhile
9    end LIST_DELETE

```

Procedure 11.2 Deletion from a straight singly-linked linear list

One of three cases may arise:

1. The list is null.
2. There is no element x in the list.
3. There is an element x in the list.

If any one of the first two cases obtains, the procedure simply returns without issuing an underflow exception. If it is important to flag an attempt to delete from a null list or to delete a nonexistent element as error conditions, the procedure must be modified accordingly.

If the third case obtains, the node containing the element is detached from the list and returned to the memory pool. Note that if the list initially consists of element x only, then it becomes null after the deletion, signified by the condition $l = \Lambda$. Verify that the procedure correctly handles this case.

The time complexity of the procedure is clearly $O(n)$, where n is the length of the list.

In the above procedures, it is necessary to handle the case of the null list differently from the non-null list (line 6 of Procedure 11.1 and line 4 of Procedure 11.2), requiring an additional test. It is often desirable, for economy of thought and economy of code, to be able to handle the null list no differently from the non-null list. It turns out that there is a very simple mechanism to accomplish this: use a **list head** (aka **head node**). We will see shortly that a list head provides other important services as well.

11.3.2 Straight singly-linked linear list with a list head

Figure 11.3(a) shows a singly-linked linear list with a *list head*. This time, the empty list is represented by a list head with the *LINK* field set to Λ , as shown in Figure 11.3(b).

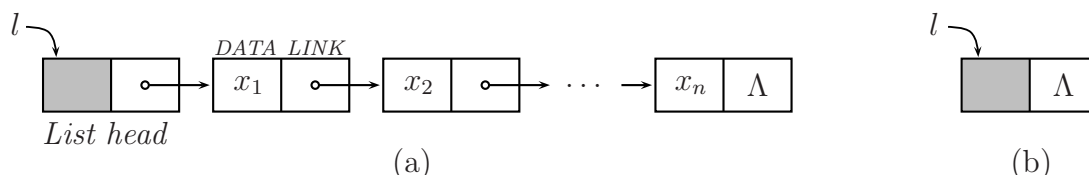


Figure 11.3 (a) Singly-linked linear list with a list head (b) Null list

The following EASY procedures implement the basic insert and delete operations for the list of Figure 11.3, represented by the same data structure $\mathbb{L} = [(DATA, LINK), l]$, with l pointing to the list head.

```

1  procedure LIST_INSERT( $\mathbb{L}, w, x$ )
▷ Inserts a new element  $w$  before element  $x$  in the list; if there is no element  $x$  in the
▷ list,  $w$  is inserted at the tail end of the list.
2  call GETNODE( $\nu$ )
3   $DATA(\nu) \leftarrow w$ 
4   $\alpha \leftarrow LINK(l); \beta \leftarrow l$ 
5  loop
6    if  $\alpha = \Lambda$  or  $DATA(\alpha) = x$  then [ $LINK(\beta) \leftarrow \nu; LINK(\nu) \leftarrow \alpha; \mathbf{return}$ ]
7    else [ $\beta \leftarrow \alpha; \alpha \leftarrow LINK(\alpha)$ ]
8  forever
9  end LIST_INSERT

```

Procedure 11.3 Insertion into a straight singly-linked linear list with a list head

```

1  procedure LIST_DELETE( $\mathbb{L}, x$ )
▷ Deletes element  $x$  from the list
2   $\alpha \leftarrow LINK(l); \beta \leftarrow l$ 
3  while  $\alpha \neq \Lambda$  do
4    if  $DATA(\alpha) = x$  then [ $LINK(\beta) \leftarrow LINK(\alpha); \mathbf{call}$  RETNODE( $\alpha$ ); return]
5    else [ $\beta \leftarrow \alpha; \alpha \leftarrow LINK(\alpha)$ ]
6  endwhile
7  end LIST_DELETE

```

Procedure 11.4 Deletion from a straight singly-linked linear list with a list head

These two procedures perform exactly the same task as their previous counterparts, with one important difference in the implementation: the same code applies to both null and non-null lists, obviating the need for a second **if** statement to test for the null case. The result are neater and more readable procedures. This, of course, is due to the fact that the list representation now uses a list head.

Needless to say, these procedures have the same time complexity as the previous two.

11.3.3 Circular singly-linked linear list

Rather than setting the *LINK* field of the last node to Λ we can make it point to the first node instead, to yield a *circular* singly-linked linear list. This implementation allows us to have direct access to both the leftmost and rightmost nodes of the list while maintaining only one external pointer to the list. With this representation, we are able to perform some basic operations on the list in $O(1)$ time, as the EASY procedures given below show.

Figure 11.4 depicts a circular list. Note that the pointer l to the list now points to the ‘rightmost’ node; this allows us to locate the ‘leftmost’ node in $O(1)$ time. The condition $l = \Lambda$ indicates that the list is empty.

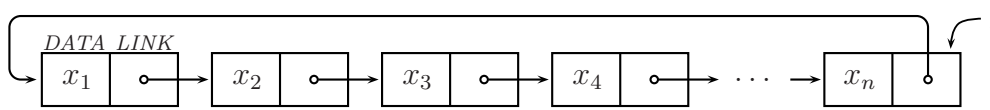


Figure 11.4 Circular singly-linked linear list

Consider now the following EASY procedures for the list of Figure 11.4, represented once more by the data structure $\mathbb{L} = [(DATA, LINK), l]$. Their use will be indicated shortly.

```

1  procedure INSERT_LEFT( $\mathbb{L}, x$ )
▷ Inserts element  $x$  at left end of a circular list
2  call GETNODE( $\alpha$ )
3   $DATA(\alpha) \leftarrow x$ 
4  if  $l = \Lambda$  then [ $LINK(\alpha) \leftarrow \alpha$ ;  $l \leftarrow \alpha$ ]
5      else [ $LINK(\alpha) \leftarrow LINK(l)$ ;  $LINK(l) \leftarrow \alpha$ ]
6  end INSERT_LEFT

```

Procedure 11.5 Inserting a new element at the left end of a circular list

```

1  procedure INSERT_RIGHT( $\mathbb{L}, x$ )
▷ Inserts element  $x$  at right end of a circular list
2  call INSERT_LEFT( $\mathbb{L}, x$ )
3   $l \leftarrow LINK(l)$ 
4  end INSERT_RIGHT

```

Procedure 11.6 Inserting a new element at the right end of a circular list

```

1  procedure DELETE_LEFT( $\mathbb{L}, x$ )
▷ Deletes leftmost element of a circular list and returns this in  $x$ 
2  if  $l = \Lambda$  then call CLIST_EMPTY
3      else [ $\alpha \leftarrow LINK(l)$ ;  $x \leftarrow DATA(\alpha)$ ;  $LINK(l) \leftarrow LINK(\alpha)$ ]
4          if  $\alpha = l$  then  $l \leftarrow \Lambda$ 
5          call RETNODE( $\alpha$ )
6  end DELETE_LEFT

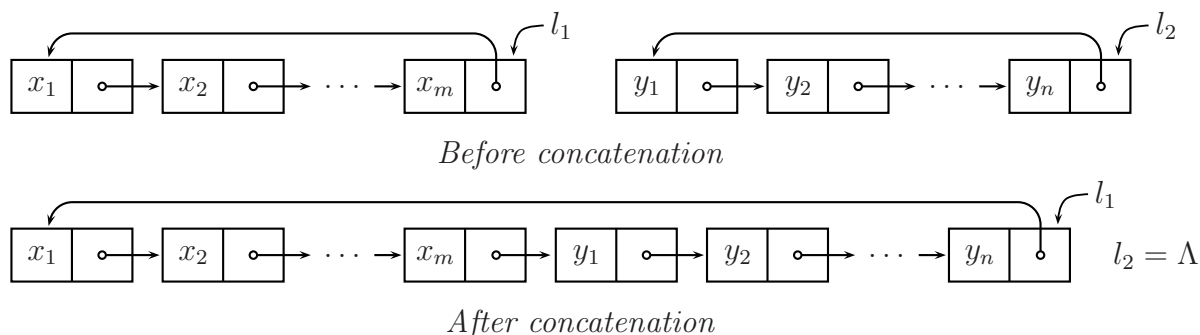
```

Procedure 11.7 Deleting the leftmost element of a circular list

We take note of the following observations pertaining to the three procedures given above.

1. All three procedures execute in $O(1)$ time.
2. With procedures `INSERT_LEFT` and `DELETE_LEFT`, and these only, we obtain a stack; with procedures `INSERT_RIGHT` and `DELETE_LEFT`, and these only, we obtain a queue.
3. With all three procedures, we obtain an output-restricted deque. With a fourth procedure, `DELETE_RIGHT`, we obtain a full blown deque. However, this last procedure takes $O(n)$ time for a list on n nodes. This is because we need the address of the node which immediately precedes the rightmost node so that its *LINK* field can be updated, and we can only obtain this address by marching down the entire length of the list.

Another operation which can be performed in $O(1)$ time for circular lists is *concatenation*. Given two lists $l_1 = (x_1, x_2, \dots, x_m)$ and $l_2 = (y_1, y_2, \dots, y_n)$ where $m \geq 0$ and $n \geq 0$, the resulting lists after concatenation are $l_1 = (x_1, x_2, \dots, x_m, y_1, y_2, \dots, y_n)$ and $l_2 = \Lambda$. The figure below shows before- and after-diagrams for the case in which both l_1 and l_2 are non-null. The two other cases of interest are: (a) l_1 is null and l_2 is non-null, in which case we simply swap pointers, and (b) l_1 is null or non-null and l_2 is null, in which case we leave things as they are.



```

1  procedure CONCATENATE(L1, L2)
▷ Concatenates two circular lists
2  if L2.l ≠ Λ then [if L1.l ≠ Λ then [α ← L1.LINK(L1.l)
3                                     L1.LINK(L1.l) ← L2.LINK(L2.l)
4                                     L2.LINK(L2.l) ← α]
5                                     L1.l ← L2.l
6                                     L2.l ← Λ]
7  end CONCATENATE

```

Procedure 11.8 Concatenating two circular lists

11.3.4 Circular singly-linked linear list with a list head

In certain applications in which a circular list is traversed in cyclic fashion, it becomes desirable to have some kind of a ‘sentinel’ to indicate that we have come full circle in traversing the list. This special node serves as a starting point and an ending point as we march through the list; it is never deleted as long as the list is in use. A list head, which we introduced earlier to simplify boundary conditions, is also used for this purpose. Aside from serving as a sentinel, a list head may also be used to store information pertaining to the list as a whole; for instance, the number of elements in the list, the number of references to the list, and so on.

Figure 11.5(a) shows a singly-linked circular list with a list head. Note that the pointer to the list points to the list head. The empty list is represented by a list head pointing to itself as shown in (b).

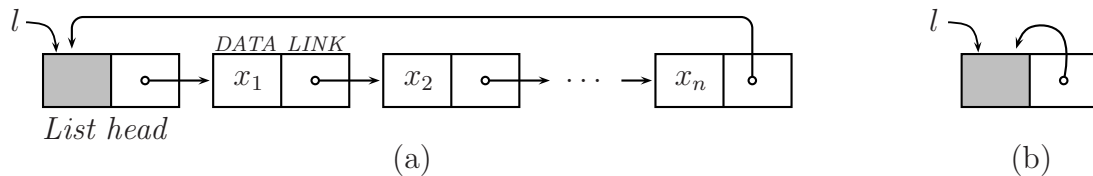


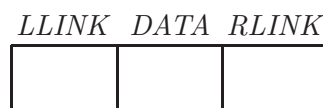
Figure 11.5 (a) Singly-linked circular list with a list head (b) Null circular list

In the next section we will utilize this type of list in implementing polynomial arithmetic on a computer.

11.3.5 Doubly-linked linear lists

We often encounter applications of linear lists in which we need to traverse the list both forwards and backwards. In section 11.5 (Linear lists and multiple-precision integer arithmetic) and in section 11.7 (Linear lists and UPCAT processing) we will study algorithms which require precisely such operations. One way to accomplish these with a singly-linked list is to invert the list in-place when it is to be traversed backwards using, for instance, Procedure 3.3 in Session 3. Of course, the list will have to be re-inverted if it is to be traversed again in the forward direction.

A more direct approach, albeit one which requires more space, is to allocate two link fields per node, say *LLINK* and *RLINK*, as shown below. The *LLINK* field holds a pointer to the node's immediate predecessor, if any, and the *RLINK* field holds a pointer to the node's immediate successor, if any, yielding a doubly-linked linear list.



There are several ways of constituting a doubly-linked list. Analogous to the singly-linked list of Figure 11.2, we may set the *LLINK* field of the leftmost node to Λ and the *RLINK* field of the rightmost node to Λ . Or, analogous to the circular list in Figure 11.4,

we may set the *LLINK* field of the leftmost node to point to the rightmost node, and the *RLINK* field of the rightmost node to point to the leftmost node. Or, we may add a list head to obtain the analogue of the circular list in Figure 11.5. The figure below shows a doubly-linked circular list with a list head. The null list is represented by a list head with the *LLINK* and *RLINK* fields pointing to the list head.

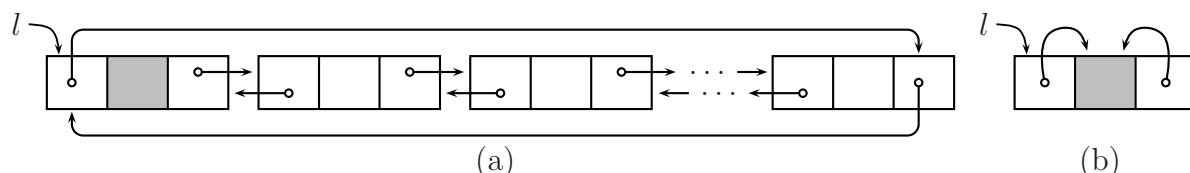
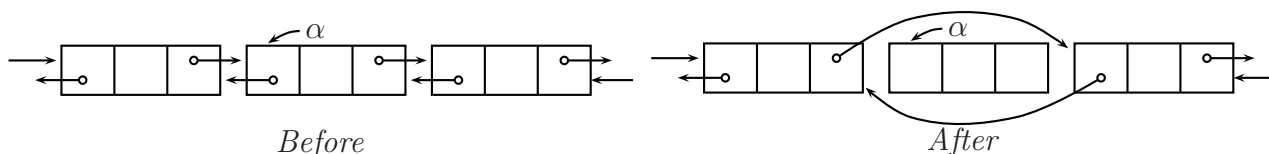


Figure 11.6 (a) Doubly-linked circular list with list head (b) Null doubly-linked list

Apart from the ease of traversing a doubly-linked list forwards and backwards, there is another compelling reason for using a doubly-linked list, namely, we can delete *any* node in the list in $O(1)$ time given only the address of the node. In section 11.6 (Linear lists and dynamic memory management) we will study three algorithms in which a doubly-linked list is used as the implementing data structure for precisely this reason.

Let α be the address of some node in a doubly-linked list; the following segment of EASY code deletes node α from the list in constant time:

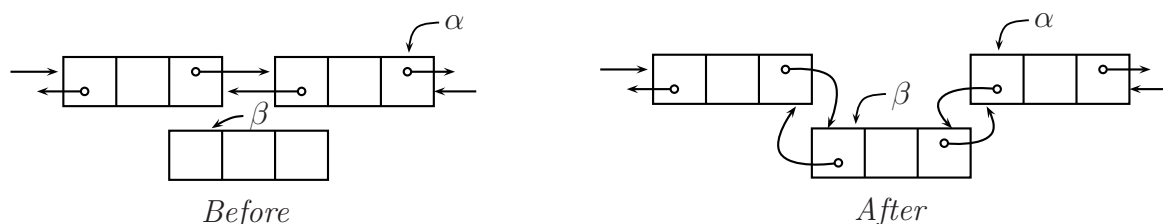
$$RLINK(LLINK(\alpha)) \leftarrow RLINK(\alpha); \quad LLINK(RLINK(\alpha)) \leftarrow LLINK(\alpha)$$



By using a simple programming trick, we can actually perform the same task (deletion in constant time given the node's address only) on a singly-linked list (think about it). Unfortunately, the trick does not work for some node in the list (which one?).

Another operation which we can readily perform on a doubly-linked list is inserting a new node *before* some node in the list. For instance, the following segment of EASY code inserts node β before node α in a doubly-linked list in $O(1)$ time (how would you do this on a singly-linked list?):

$$RLINK(LLINK(\alpha)) \leftarrow \beta; \quad RLINK(\beta) \leftarrow \alpha; \quad LLINK(\beta) \leftarrow LLINK(\alpha); \quad LLINK(\alpha) \leftarrow \beta$$



Interchanging 'left' and 'right' in the above segment of code will insert node β *after* node α .

11.4 Linear lists and polynomial arithmetic

In the remainder of this session, we will consider four applications of linear lists, namely:

1. polynomial arithmetic
2. multiple-precision integer arithmetic
3. dynamic storage management
4. UPCAT processing

The first three are vintage CS; the fourth is vintage UP.

One of the earliest applications of linked linear lists is **polynomial arithmetic**. We are familiar with polynomials and operations on polynomials such as addition, subtraction, multiplication, division, differentiation, integration, and the like. We know that the result of any of these operations is also a polynomial; just how many terms the resulting polynomial will have, however, may not always be predictable. For instance, consider the polynomials $P(x, y) = x^5 - 2x^4y + 3x^3y^2 - 4x^2y^3 + 5xy^4 - 6y^5$ and $Q(x, y) = x^2 + 2xy + y^2$. The product $P \times Q$ can have as many as 18 terms; in fact, it has only three: $P \times Q = x^7 - 7xy^6 - 6y^7$. The product of the polynomial $x^{99} + x^{98} + x^{97} + \dots + x + 1$ (with 100 terms) with $x + 1$ is a polynomial with 101 terms; its product with $x - 1$ has only two terms.

An efficient implementation of polynomial arithmetic on a computer must address two important considerations:

1. a way of representing the terms of a polynomial so that the entities which comprise each such term (coefficient and exponents) can be accessed and processed with ease
2. a way of managing unpredictable space requirements with a structure which can grow or shrink in size as required

Linked lists satisfy these requirements. Specifically, we will use a circular singly-linked list with a list head, as depicted in Figure 11.5, to represent a polynomial in the variables x , y and z . Each term of the polynomial is a node of the linked list. Each node will have the node structure:

<i>EXPO</i>				<i>COEF</i>	<i>LINK</i>
<i>t</i>	e_x	e_y	e_z		

The *EXPO* field is divided into a ‘tag’ subfield and three ‘exponent’ subfields to contain the exponents of the variables x , y and z . All exponents are assumed to be non-negative integers. The list head is distinguished by an entry of ‘-001’ in the *EXPO* field; all other nodes have a tag of ‘+’ in the *EXPO* field. The *COEF* field may contain any real number, signed or unsigned.

Shown below is the linked list representation of the polynomial $P(x, y, z) = x^7 - 7xy^6 - 6y^7$. Note that the nodes are arranged in decreasing value of the triple (e_x, e_y, e_z) . A polynomial with this property is said to be in *canonical* form, and certain operations (e.g., addition) are more efficiently carried out with polynomials in this form.

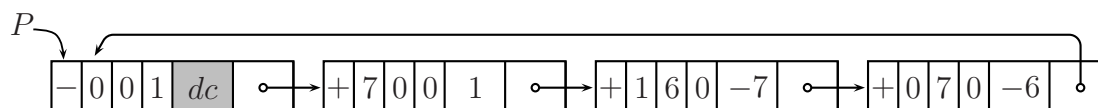


Figure 11.7 Linked-list representation of $P(x, y, z) = x^7 - 7xy^6 - 6y^7$

The zero polynomial is the equivalent of the empty list and is represented by a list head pointing to itself.

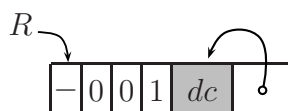


Figure 11.8 Linked-list representation of $R(x, y, z) = 0$

Some EASY procedures for polynomial arithmetic

With polynomials represented as in Figure 11.7, we can readily implement procedures to perform such operations on polynomials as addition, subtraction, multiplication, division, and the like. Likewise we can write routines to perform such housekeeping tasks as reading in a polynomial and generating its internal representation, displaying a polynomial on some output medium, creating a zero polynomial, deleting a polynomial (i.e., returning all its nodes, including list head, to the memory pool), and so on. In the rest of the section we will implement some of these operations; the rest are left as exercises.

We make the following assumptions with respect to the following EASY procedures which implement some of the operations listed above.

- (a) A polynomial, say P , is represented using the data structure

$$\mathbb{P} = [(EXPO, COEF, LINK), l, name]$$

The first component specifies the node structure of each node in the list. The second component, l , contains the address of the list head, or equivalently, it is the pointer to the polynomial. The third component, $name$, is the name given to the polynomial for external identification.

- (b) All expressions involving the *EXPO* field, e.g., $EXPO(P) < EXPO(Q)$, $EXPO(P) = EXPO(Q)$, $EXPO(P) + EXPO(Q)$, etc., should be interpreted as calls to functions which implement the indicated operations *exponent-wise*.

The following four EASY procedures are illustrative of the kind of operations usually performed on linked lists, for instance, traversing one or more lists concurrently, inserting a node into a list, deleting a node from a list, and so on. Study the procedures carefully.

1. *Generating the internal representation of an input polynomial*

The procedure $\text{POLYREAD}(\mathbb{P}, \text{name})$ reads in quadruples of the form $(e_x, e_y, e_z, \text{coef})$, each representing a polynomial term, and generates the internal representation of the polynomial in canonical form. The generated polynomial is given the name *name*.

```

1  procedure POLYREAD( $\mathbb{P}$ , name)
2  call ZEROPOLY( $\mathbb{P}$ , name)
3  while not EOI do
4    input  $e_x, e_y, e_z, \text{coef}$ 
5    call GETNODE( $\nu$ )
6     $\mathbb{P}.\text{EXPO}(\nu) \leftarrow (+, e_x, e_y, e_z)$ 
7     $\mathbb{P}.\text{COEF}(\nu) \leftarrow \text{coef}$ 
8     $\beta \leftarrow \mathbb{P}.l$ ;  $\alpha \leftarrow \mathbb{P}.\text{LINK}(\beta)$ 
9    while  $\mathbb{P}.\text{EXPO}(\alpha) > \mathbb{P}.\text{EXPO}(\nu)$  do
10      $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \mathbb{P}.\text{LINK}(\alpha)$ 
11  endwhile
12   $\mathbb{P}.\text{LINK}(\beta) \leftarrow \nu$ 
13   $\mathbb{P}.\text{LINK}(\nu) \leftarrow \alpha$ 
14 endwhile
15 end POLYREAD

```

```

1  procedure ZEROPOLY( $\mathbb{P}$ , name)
2  call GETNODE( $\tau$ )
3   $\mathbb{P}.l \leftarrow \tau$ 
4   $\mathbb{P}.\text{name} \leftarrow \text{name}$ 
5   $\mathbb{P}.\text{EXPO}(\tau) \leftarrow (-, 0, 0, 1)$ 
6   $\mathbb{P}.\text{LINK}(\tau) \leftarrow \tau$ 
7  end ZEROPOLY

```

Procedure 11.9 Generating the internal representation of an input polynomial

The call to ZEROPOLY in line 2 creates a zero polynomial \mathbb{P} , pointed to by $\mathbb{P}.l$ and whose name is $\mathbb{P}.\text{name}$; this becomes the list head. In lines 3–14 the rest of the polynomial is generated, term by term, through a three-step process: (a) read in the exponents and coefficient which define the next term (line 4) (b) get a node from the memory pool and store the input values in the appropriate fields of the node (lines 5–7), and (c) insert the new node into the list such that the list is sorted by decreasing value of the triple (e_x, e_y, e_z) , i.e., the resulting polynomial is in canonical form (lines 8–13).

The procedure has time complexity $\Omega(n)$ and $O(n^2)$ where n is the number of terms in the polynomial. The best-case linear running time obtains when the input is reverse-sorted; the worst-case quadratic running time results when the input is already sorted! This is because the list is traversed in the forward direction, starting from the list head, whenever a new term is inserted.

2. Polynomial addition

The procedure $\text{POLYADD}(\mathbb{P}, \mathbb{Q}, \text{name})$ performs the operation $Q \leftarrow P + Q$, where P and Q are polynomials in canonical form. The sum is retained in \mathbb{Q} ; if the original polynomial Q is to be used again, a copy of it must first be made. Invoking $\text{POLYADD}(\mathbb{Q}, \mathbb{C}, \text{name})$, where \mathbb{C} is initially a zero polynomial, returns a copy of \mathbb{Q} in \mathbb{C} .

```

1  procedure POLYADD( $\mathbb{P}, \mathbb{Q}, \text{name}$ )
2     $\alpha \leftarrow \mathbb{P}.\text{LINK}(\mathbb{P}.l)$             $\triangleright$  pointer to current term (node) in  $P$ 
3     $\beta \leftarrow \mathbb{Q}.\text{LINK}(\mathbb{Q}.l)$         $\triangleright$  pointer to current term (node) in  $Q$ 
4     $\sigma \leftarrow \mathbb{Q}.l$                   $\triangleright$  a pointer that is always one node behind  $\beta$ 
5    loop
6      case
7        :  $\mathbb{P}.\text{EXPO}(\alpha) < \mathbb{Q}.\text{EXPO}(\beta)$  : [  $\sigma \leftarrow \beta$ ;  $\beta \leftarrow \mathbb{Q}.\text{LINK}(\beta)$ ; ]
8        :  $\mathbb{P}.\text{EXPO}(\alpha) = \mathbb{Q}.\text{EXPO}(\beta)$  : [ if  $\mathbb{P}.\text{EXPO}(\alpha) < 0$  then [  $\mathbb{Q}.\text{name} \leftarrow \text{name}$ ; return ]
9                                            $\mathbb{Q}.\text{COEF}(\beta) \leftarrow \mathbb{Q}.\text{COEF}(\beta) + \mathbb{P}.\text{COEF}(\alpha)$ 
10                                          if  $\mathbb{Q}.\text{COEF}(\beta) = 0$  then [  $\tau \leftarrow \beta$ 
11                                                          $\mathbb{Q}.\text{LINK}(\sigma) \leftarrow \mathbb{Q}.\text{LINK}(\beta)$ 
12                                                          $\beta \leftarrow \mathbb{Q}.\text{LINK}(\beta)$ 
13                                                         call RETNODE( $\tau$ ) ]
14                                          else [  $\sigma \leftarrow \beta$ 
15                                                          $\beta \leftarrow \mathbb{Q}.\text{LINK}(\beta)$  ]
16                                           $\alpha \leftarrow \mathbb{P}.\text{LINK}(\alpha)$  ]
17        :  $\mathbb{P}.\text{EXPO}(\alpha) > \mathbb{Q}.\text{EXPO}(\beta)$  : [ call GETNODE( $\tau$ )
18                                            $\text{COEF}(\tau) \leftarrow \mathbb{P}.\text{COEF}(\alpha)$ ;  $\text{EXPO}(\tau) \leftarrow \mathbb{P}.\text{EXPO}(\alpha)$ 
19                                            $\mathbb{Q}.\text{LINK}(\sigma) \leftarrow \tau$ ;  $\text{LINK}(\tau) \leftarrow \beta$ 
20                                            $\sigma \leftarrow \tau$ ;  $\alpha \leftarrow \mathbb{P}.\text{LINK}(\alpha)$  ]
21      endcase
22    forever
23  end POLYADD

```

Procedure 11.10 Implementing polynomial addition using linked lists

We take note of the following observations pertaining to procedure POLYADD.

- (a) The implementation of the addition operation as $Q \leftarrow P + Q$ has certain advantages, scil.,
 - i. It obviates the need for a separate *copy* routine; as pointed out earlier, POLYADD serves the purpose.
 - ii. In general, it requires fewer calls to GETNODE, saving both time and space.
 - iii. It provides a convenient way of computing a polynomial as a running sum of several other polynomials, as in procedure POLYMULT given below.
- (b) We use three running pointers: α to point to the current term (node) in polynomial P , β to point to the current term (node) in polynomial Q , and σ which is always one node behind β (lines 2–4).

- (c) One of three cases may obtain. If $EXPO(\alpha) < EXPO(\beta)$, the pointers to polynomial Q are simply advanced one node down the list; this is because the sum is retained in Q (line 7). If $EXPO(\alpha) > EXPO(\beta)$, a copy of the current term in P is made and is inserted before the current term in Q ; with the trailing pointer σ , the insertion is done in $O(1)$ time. Then we move on to the next term in P (lines 17–20).
- (d) The case $EXPO(\alpha) = EXPO(\beta)$ indicates one of two conditions: if $EXPO(\alpha) < 0$, then both pointers α and β have come full circle and are now pointing to the list heads, and the procedure terminates. Otherwise, α and β are pointing to two terms which can be added. Should the coefficients sum to zero, then the node is deleted from Q and is returned to the memory pool. With the trailing pointer σ , the deletion is done in $O(1)$ time. We then move on to the next terms in P and Q (lines 8–16).
- (e) The procedure makes only one pass through each list; this is because P and Q are in canonical form. If they are not, the procedure will not yield the correct result. If P has m terms and Q has n terms, the time complexity of the procedure is clearly $O(m + n)$.
- (f) There is no special test for the zero polynomial. The procedure works whether P is zero and/or Q is zero. This is a consequence of the fact that we used a list with a list head to represent a polynomial, enabling us to handle the zero polynomial (null list) no differently from a non-zero polynomial (non-null list).

3. Polynomial subtraction

The procedure $POLYSUB(\mathbb{P}, \mathbb{Q}, name)$ performs the operation $\mathbb{Q} \leftarrow P - Q$, where P and Q are polynomials in canonical form. The difference is retained in \mathbb{Q} ; if the original polynomial Q is to be used again, a copy of it must first be made. Invoking $POLYADD(\mathbb{Q}, \mathbb{C}, name)$, where \mathbb{C} is initially a zero polynomial returns a copy of \mathbb{Q} in \mathbb{C} .

```

1  procedure POLYSUB( $\mathbb{P}, \mathbb{Q}, name$ )
2  if  $\mathbb{P}.l = \mathbb{Q}.l$  then [call RETPOLY( $\mathbb{Q}$ ); call ZEROPOLY( $\mathbb{Q}, name$ )]
3      else [ $\beta \leftarrow \mathbb{Q}.LINK(\mathbb{Q}.l)$ 
4          while  $\mathbb{Q}.EXPO(\beta) \neq -001$  do
5               $\mathbb{Q}.COEF(\beta) \leftarrow -\mathbb{Q}.COEF(\beta)$ 
6               $\beta \leftarrow \mathbb{Q}.LINK(\beta)$ 
7          endwhile
8          call POLYADD( $\mathbb{P}, \mathbb{Q}, name$ )]
9  end POLYSUB

```

Procedure 11.11 Implementing polynomial subtraction using linked lists

The operation $\mathbb{Q} \leftarrow P - Q$ is actually implemented as $\mathbb{Q} \leftarrow P + (-Q)$. However, this procedure does not work if $P = Q$ since then we will obtain $-2Q$ instead of 0. This special case is handled in line 2, which deletes \mathbb{Q} (returns the entire list, including list head, to the memory pool) and then creates a zero polynomial \mathbb{Q} .

4. Polynomial multiplication

The procedure `POLYMULT(P, Q, R, name)` performs the operation $R \leftarrow R + P * Q$. Upon entry into the procedure, R is a zero polynomial; upon exit, it contains the product $P * Q$ in canonical form.

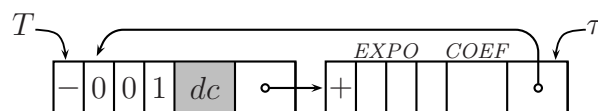
```

1  procedure POLYMULT(P, Q, R, name)
  ▷ Create temporary polynomial T to contain product term.
2  call ZEROPOLY(T, ' ')
3  call GETNODE( $\tau$ )
4  T.LINK(T.l)  $\leftarrow \tau$ 
5  LINK( $\tau$ )  $\leftarrow$  T.l
  ▷ Perform multiplication
6   $\alpha \leftarrow$  P.LINK(P.l)
7  while P.EXPO( $\alpha$ )  $\neq$  -001 do
8     $\beta \leftarrow$  Q.LINK(Q.l)
9    while Q.EXPO( $\beta$ )  $\neq$  -001 do
10     T.COEF( $\tau$ )  $\leftarrow$  P.COEF( $\alpha$ ) * Q.COEF( $\beta$ )
11     T.EXPO( $\tau$ )  $\leftarrow$  P.EXPO( $\alpha$ ) + Q.EXPO( $\beta$ )
12     call POLYADD(T, R, name)
13      $\beta \leftarrow$  Q.LINK( $\beta$ )
14   endwhile
15    $\alpha \leftarrow$  P.LINK( $\alpha$ )
16 endwhile
17 call RETPOLY(T)
18 end POLYMULT

```

Procedure 11.12 Implementing polynomial multiplication using linked lists

In lines 2–5 we create a temporary working polynomial T as shown below, to serve as a placeholder for the product terms generated during the multiplication. Specifically



each term in P (lines 6–7, 15) is multiplied with every term in Q (lines 8–9, 13); the product is stored in T (lines 10–11) which is then accumulated into the product polynomial R (line 12) via a call to `POLYADD`, thus: $R \leftarrow R + T$. Finally, the call to `RETPOLY` in line 17 erases the temporary polynomial T by returning all its nodes, including list head, to the memory pool.

The most costly part of the operation is forming a product term and adding this to R (lines 10–12); these steps are carried out $m \times n$ times. The time complexity of the procedure is therefore $O(mn)$, where m and n are the number of terms in P and Q , respectively.

11.5 Linear lists and multiple-precision integer arithmetic

In a computing environment which uses four bytes of memory to represent integers, the range of representable values is $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$ (i.e., -2^{31}) to $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$ (i.e., $2^{31}-1$) or $-2,147,483,648$ to $2,147,483,647$. If, for instance, we add two integers whose sum lies outside this range using the built-in operator '+', an overflow condition obtains and we get erroneous results. Likewise if we attempt to assign a value outside this range to an integer variable the compiler will issue an integer overflow exception. Clearly, in applications which involve large, or 'long', integers it becomes necessary to devise ways of representing such integers in computer memory and to devise algorithms to perform the desired operations, such as the four basic arithmetic operations.

11.5.1 Integer addition and subtraction

Let A and B be integers. We can write A , B and their sum $S = A + B$ as

$$\begin{aligned} A &= \sum_{i=0}^{m-1} a_i r^i = a_0 r^0 + a_1 r^1 + a_2 r^2 + \dots + a_{m-1} r^{m-1} \\ B &= \sum_{i=0}^{n-1} b_i r^i = b_0 r^0 + b_1 r^1 + b_2 r^2 + \dots + b_{n-1} r^{n-1} \\ S &= \sum_{i=0}^{k-1} s_i r^i = s_0 r^0 + s_1 r^1 + s_2 r^2 + \dots + s_{k-1} r^{k-1} \end{aligned} \quad (11.1)$$

where $k \leq \max(m, n) + 1$. For instance, if $A = 9753$ and $B = 819$, then $S = 10572$; taking $r = 10$ we have

$$\begin{aligned} A &= \sum_{i=0}^3 a_i r^i = 3 \times 10^0 + 5 \times 10^1 + 7 \times 10^2 + 9 \times 10^3 \\ &\quad \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \\ &\quad \quad \quad a_0 \quad \quad \quad a_1 \quad \quad \quad a_2 \quad \quad \quad a_3 \\ B &= \sum_{i=0}^2 b_i r^i = 9 \times 10^0 + 1 \times 10^1 + 8 \times 10^2 \\ &\quad \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \\ &\quad \quad \quad b_0 \quad \quad \quad b_1 \quad \quad \quad b_2 \\ S &= \sum_{i=0}^4 s_i r^i = 2 \times 10^0 + 7 \times 10^1 + 5 \times 10^2 + 0 \times 10^3 + 1 \times 10^4 \\ &\quad \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \\ &\quad \quad \quad s_0 \quad \quad \quad s_1 \quad \quad \quad s_2 \quad \quad \quad s_3 \quad \quad \quad s_4 \end{aligned}$$

We know how to add 9753 and 819 'by hand' as in

$$\begin{array}{rcccc} & 1 & 0 & 1 & \leftarrow \text{carry} \\ & 9 & 7 & 5 & 3 \\ + & & 8 & 1 & 9 \\ \hline 1 & 0 & 5 & 7 & 2 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

354 SESSION 11. Linear lists

What we may not be aware of is that in the process we have actually used the **mod** and **floor** functions. The computations below formalize what we have mechanically carried out as integer addition. In these computations, the quantity c_i is the ‘carry’ that is generated at the i th stage and accumulated in the next.

$$\begin{aligned}
 s_0 &= (a_0 + b_0) \bmod 10 = (3 + 9) \bmod 10 = 12 \bmod 10 = 2 \\
 c_0 &= \lfloor (a_0 + b_0)/10 \rfloor = \lfloor (3 + 9)/10 \rfloor = \lfloor 12/10 \rfloor = 1 \\
 s_1 &= (a_1 + b_1 + c_0) \bmod 10 = (5 + 1 + 1) \bmod 10 = 7 \bmod 10 = 7 \\
 c_1 &= \lfloor (a_1 + b_1 + c_0)/10 \rfloor = \lfloor (5 + 1 + 1)/10 \rfloor = \lfloor 7/10 \rfloor = 0 \\
 s_2 &= (a_2 + b_2 + c_1) \bmod 10 = (7 + 8 + 0) \bmod 10 = 15 \bmod 10 = 5 \\
 c_2 &= \lfloor (a_2 + b_2 + c_1)/10 \rfloor = \lfloor (7 + 8 + 0)/10 \rfloor = \lfloor 15/10 \rfloor = 1 \\
 s_3 &= (a_3 + b_3 + c_2) \bmod 10 = (9 + 0 + 1) \bmod 10 = 10 \bmod 10 = 0 \\
 c_3 &= \lfloor (a_3 + b_3 + c_2)/10 \rfloor = \lfloor (9 + 0 + 1)/10 \rfloor = \lfloor 10/10 \rfloor = 1 \\
 s_4 &= c_3 = 1
 \end{aligned}$$

The following equation summarizes the above procedure. Let $c_{-1} = 0$; then we have

$$\left. \begin{aligned}
 s_i &= (a_i + b_i + c_{i-1}) \bmod r \\
 c_i &= \lfloor (a_i + b_i + c_{i-1})/r \rfloor
 \end{aligned} \right\} \quad 0 \leq i \leq k-2 \quad (11.2)$$

$$s_{k-1} = c_{k-2}$$

Actually, the given example is just the first of four possible cases that may arise. We will see shortly that Eq.(11.2) suffices to handle all four cases.

Case 1. Both operands are positive.

$$\begin{array}{r}
 A = + \quad 9753 \\
 B = + \quad 819 \\
 \hline
 C = + \quad 10572
 \end{array}$$

Case 2. Both operands are negative.

$$\begin{array}{r}
 A = - \quad 9753 \\
 B = - \quad 819 \\
 \hline
 C = - \quad 10572
 \end{array}$$

We simply handle Case 2 as Case 1, i.e., we consider both operands to be positive, perform the addition, then we affix a minus sign to the sum.

Case 3. Operands have opposite signs; positive operand is numerically larger.

$$\begin{array}{r}
 A = + \quad 9753 \\
 B = - \quad 819 \\
 \hline
 C = + \quad 8934
 \end{array}$$

We apply Eq.(11.2) directly as in Case 1. This time the quantity c_i is actually a ‘borrow’ rather than a ‘carry’. The following computations should help further clarify the procedure.

$$\begin{aligned}
 s_0 &= (a_0 + b_0) \bmod 10 = (3 - 9) \bmod 10 = -6 \bmod 10 = 4 \\
 c_0 &= \lfloor (a_0 + b_0)/10 \rfloor = \lfloor (3 - 9)/10 \rfloor = \lfloor -6/10 \rfloor = -1 \\
 s_1 &= (a_1 + b_1 + c_0) \bmod 10 = (5 - 1 - 1) \bmod 10 = 3 \bmod 10 = 3 \\
 c_1 &= \lfloor (a_1 + b_1 + c_0)/10 \rfloor = \lfloor (5 - 1 - 1)/10 \rfloor = \lfloor 3/10 \rfloor = 0 \\
 s_2 &= (a_2 + b_2 + c_1) \bmod 10 = (7 - 8 + 0) \bmod 10 = -1 \bmod 10 = 9 \\
 c_2 &= \lfloor (a_2 + b_2 + c_1)/10 \rfloor = \lfloor (7 - 8 + 0)/10 \rfloor = \lfloor -1/10 \rfloor = -1 \\
 s_3 &= (a_3 + b_3 + c_2) \bmod 10 = (9 + 0 - 1) \bmod 10 = 8 \bmod 10 = 8 \\
 c_3 &= \lfloor (a_3 + b_3 + c_2)/10 \rfloor = \lfloor (9 + 0 - 1)/10 \rfloor = \lfloor 8/10 \rfloor = 0 \\
 s_4 &= c_3 = 0
 \end{aligned}$$

Case 4. Operands have opposite signs; negative operand is numerically larger.

$$\begin{array}{r}
 A = - 9753 \\
 B = + 819 \\
 \hline
 C = - 8934
 \end{array}$$

We handle Case 4 as Case 3, i.e., we reverse the signs of the operands, perform the addition, then we affix a minus sign to the sum.

We can cast these observations, along with Eq.(11.2), as an algorithm for integer addition. The algorithm subsumes integer subtraction since the quantities involved are signed integers.

Algorithm A: Integer addition

Let $A = \sum_{i=0}^{m-1} a_i r^i$ and $B = \sum_{i=0}^{n-1} b_i r^i$ be two integers; their sum $S = \sum_{i=0}^{k-1} s_i r^i$, $k \leq \max(m, n) + 1$ is obtained as follows:

1. [Find temporary signs.] Find s_A and s_B according to the following table:

Sign	$ A > B $		$ A \leq B $	
	$A \cdot B > 0$	$A \cdot B \leq 0$	$A \cdot B > 0$	$A \cdot B \leq 0$
s_A	+1	+1	+1	-1
s_B	+1	-1	+1	+1

2. [Find absolute value of sum.] Let $k = \max(m, n) + 1$, $c_{-1} = 0$.
 - (a) Do for $i = 0, 1, 2, \dots, k - 2$

$$\begin{aligned}
 t &\leftarrow s_A \cdot |a_i| + s_B \cdot |b_i| + c_{i-1} \\
 s_i &\leftarrow t \bmod r \\
 c_i &\leftarrow \lfloor t/r \rfloor
 \end{aligned}$$
 - (b) $s_{k-1} \leftarrow c_{k-2}$
3. [Find actual sign of sum.] If $A > 0$ **and** $B > 0$ **or** $|A| > |B|$ **and** $A > 0$ **or** $|A| \leq |B|$ **and** $B > 0$ **then** $sign = '+'$ **else** $sign = '-'$.
4. [Find sum.] $S = sign \ s_{k-1} s_{k-2} \dots s_1 s_0$

Addition of long integers

If we want to add 9753 and 819 on a computer we will of course use the intrinsic operator '+' instead of Algorithm A. The utility of the algorithm becomes manifest when we need to add long integers. Suppose we want to add $A = 5,125,698,642,396,745$ and $B = -749,865,712,998,197$. Taking $r = 10,000$ and using Eq.(11.1) we write A and B as

$$\begin{aligned}
 A &= \sum_{i=0}^3 a_i r^i = \underbrace{6745}_{a_0} \times 10000^0 + \underbrace{4239}_{a_1} \times 10000^1 + \underbrace{6986}_{a_2} \times 10000^2 + \underbrace{5125}_{a_3} \times 10000^3 \\
 B &= \sum_{i=0}^3 b_i r^i = -\underbrace{8197}_{b_0} \times 10000^0 - \underbrace{1299}_{b_1} \times 10000^1 - \underbrace{8657}_{b_2} \times 10000^2 - \underbrace{749}_{b_3} \times 10000^3
 \end{aligned}$$

Applying Algorithm A we obtain the sum $S = A + B$ as follows:

1. $|A| > |B|$ and $A \cdot B < 0$, therefore $s_A = +1$, $s_B = -1$
2. $|A| > |B|$ and $A > 0$, therefore $sign = +$
3. Find $|S|$

$$\begin{aligned}
 c_{-1} &= 0 \\
 s_0 &= (6745 - 8197 + 0) \bmod 10000 = (-1452) \bmod 10000 = 8548 \\
 c_0 &= \lfloor (-1452)/10000 \rfloor = -1 \\
 s_1 &= (4239 - 1299 - 1) \bmod 10000 = 2939 \bmod 10000 = 2939 \\
 c_1 &= \lfloor 2939/10000 \rfloor = 0 \\
 s_2 &= (6986 - 8657 + 0) \bmod 10000 = (-1671) \bmod 10000 = 8329 \\
 c_2 &= \lfloor -1671/10000 \rfloor = -1 \\
 s_3 &= (5125 - 749 - 1) \bmod 10000 = 4375 \bmod 10000 = 4375 \\
 c_3 &= \lfloor 4375/10000 \rfloor = 0 \\
 s_4 &= 0
 \end{aligned}$$

4. Hence: $S = +4375832929398548$ (as you may easily verify)

11.5.2 Integer multiplication

Let A and B be integers. We can write A and B as in Eq.(11.1) and their product $P = A * B$ as

$$P = \sum_{i=0}^{k-1} p_i r^i = p_0 r^0 + p_1 r^1 + p_2 r^2 + \dots + p_{k-1} r^{k-1}$$

where $k \leq m + n$. For instance, if $A = 816$ and $B = 9573$, then $P = 7811568$; taking $r = 10$ we have

$$P = \sum_{i=0}^6 p_i r^i = \underset{\substack{\uparrow \\ p_0}}{8} \times 10^0 + \underset{\substack{\uparrow \\ p_1}}{6} \times 10^1 + \underset{\substack{\uparrow \\ p_2}}{5} \times 10^2 + \underset{\substack{\uparrow \\ p_3}}{1} \times 10^3 + \underset{\substack{\uparrow \\ p_4}}{1} \times 10^4 + \underset{\substack{\uparrow \\ p_5}}{8} \times 10^5 + \underset{\substack{\uparrow \\ p_6}}{7} \times 10^6$$

We know how to multiply 816 and 9573 ‘by hand’ as in

$$\begin{array}{rcccccc}
 & & 9 & 5 & 7 & 3 & \\
 & & \times & & 8 & 1 & 6 \\
 \hline
 & & 5 & 7 & 4 & 3 & 8 \leftarrow T^{(1)} \\
 & & 9 & 5 & 7 & 3 & \leftarrow T^{(2)} \\
 & 7 & 6 & 5 & 8 & 4 & \leftarrow T^{(3)} \\
 \hline
 7 & 8 & 1 & 1 & 5 & 6 & 8 \leftarrow P \\
 \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
 p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
 \end{array}$$

On a computer we can implement this conventional procedure by computing P as a ‘running sum’ of the partial products $T^{(i)}$ using the iterative formula $P \leftarrow P + T^{(i)} \times sf$, where sf is an appropriate scale factor. The following computations should clarify the process.

$$\begin{aligned}
 P &\leftarrow 0 \\
 T &\leftarrow 6 \times 9573 &= 57438 \\
 P &\leftarrow 0 + 57438 \times 10^0 &= 57438 \\
 T &\leftarrow 1 \times 9573 &= 9573 \\
 P &\leftarrow 57438 + 9573 \times 10^1 &= 153168 \\
 T &\leftarrow 8 \times 9573 &= 76584 \\
 P &\leftarrow 153168 + 76584 \times 10^2 = 7811568
 \end{aligned}$$

We can describe the above process as computing P ‘ $T^{(i)}$ -wise’, i.e., we compute a complete partial product before we accumulate it into P . A better strategy is to compute P ‘ p_i -wise’, i.e., we compute $p_0, p_1, p_2, \dots, p_{k-1}$, as *separate* running sums by performing the multiplication and addition *concurrently*. The following algorithm formalizes the idea.

Algorithm M: Integer multiplication

Let $A = \sum_{i=0}^{m-1} a_i r^i$ and $B = \sum_{i=0}^{n-1} b_i r^i$ be two integers; the product $P = \sum_{i=0}^{k-1} p_i r^i$, $k \leq m + n$ is obtained as follows:

1. [Initialize P .] Set $p_i \leftarrow 0$, $i = 0, 1, 2, \dots, k - 1$
2. [Find absolute value of product, ignoring signs of A and B .]

Do for $i = 0, 1, 2, \dots, m - 1$
 $c \leftarrow 0$
 Do for $j = 0, 1, 2, \dots, n - 1$
 $t \leftarrow a_i \times b_j + p_{i+j} + c$
 $p_{i+j} \leftarrow t \bmod r$
 $c \leftarrow \lfloor t/r \rfloor$
 $p_{i+n} \leftarrow c$

3. [Find actual sign of product.] If A and B have the same sign, then $sign = '+'$, else $sign = '-'$.
4. [Find product.] $P = sign \ p_{k-1} p_{k-2} \dots p_1 p_0$

Multiplication of long integers

As with algorithm A, the principal usefulness of algorithm M is in performing arithmetic on long integers, since otherwise we will simply use the intrinsic arithmetic operators. Suppose we want to multiply $A = -102,465$ and $B = 512,569,864,239,675$. Taking $r = 10,000$ and using Eq.(11.1) we write A and B as

$$A = \sum_{i=0}^1 a_i r^i = -\underbrace{2465}_{a_0} \times 10000^0 - \underbrace{10}_{a_1} \times 10000^1$$

$$B = \sum_{i=0}^3 b_i r^i = +\underbrace{9675}_{b_0} \times 10000^0 + \underbrace{6423}_{b_1} \times 10000^1 + \underbrace{5698}_{b_2} \times 10000^2 + \underbrace{512}_{b_3} \times 10000^3$$

Applying Algorithm M we obtain the product $P = A \times B$ as follows:

Step 1. $k = m + n = 2 + 4 = 6$; $p_0 = p_1 = \dots = p_5 = 0$

Step 2. $i = 0$

$$c = 0$$

$$\begin{aligned} j = 0: t &= a_0 \times b_0 + p_0 + c = 2465 \times 9675 + 0 + 0 = 23848875 \\ p_0 &= t \bmod r = 23848875 \bmod 10^4 = 8875 \\ c &= \lfloor t/r \rfloor = \lfloor 23848875/10^4 \rfloor = 2384 \end{aligned}$$

$$\begin{aligned} j = 1: t &= a_0 \times b_1 + p_1 + c = 2465 \times 6423 + 0 + 2384 = 15835079 \\ p_1 &= 15835079 \bmod 10^4 = 5079 \\ c &= \lfloor 15835079/10^4 \rfloor = 1583 \end{aligned}$$

$$\begin{aligned} j = 2: t &= a_0 \times b_2 + p_2 + c = 2465 \times 5698 + 0 + 1583 = 14047153 \\ p_2 &= 14047153 \bmod 10^4 = 7153 \\ c &= \lfloor 14047153/10^4 \rfloor = 1404 \end{aligned}$$

$$\begin{aligned} j = 3: t &= a_0 \times b_3 + p_3 + c = 2465 \times 512 + 0 + 1404 = 1263484 \\ p_3 &= 1263484 \bmod 10^4 = 3484 \\ c &= \lfloor 1263484/10^4 \rfloor = 126 \end{aligned}$$

$$p_4 = c = 126$$

$i = 1$

$$c = 0$$

$$\begin{aligned} j = 0: t &= a_1 \times b_0 + p_1 + c = 10 \times 9675 + 5079 + 0 = 101829 \\ p_1 &= 101829 \bmod 10^4 = 1829 \\ c &= \lfloor 101829/10^4 \rfloor = 10 \end{aligned}$$

$$\begin{aligned} j = 1: t &= a_1 \times b_1 + p_2 + c = 10 \times 6423 + 7153 + 10 = 71393 \\ p_2 &= 71393 \bmod 10^4 = 1393 \\ c &= \lfloor 71393/10^4 \rfloor = 7 \end{aligned}$$

$$\begin{aligned} j = 2: t &= a_1 \times b_2 + p_3 + c = 10 \times 5698 + 3484 + 7 = 60471 \\ p_3 &= 60471 \bmod 10^4 = 0471 \\ c &= \lfloor 60471/10^4 \rfloor = 6 \end{aligned}$$

$$\begin{aligned}
 j = 3: t &= a_1 \times b_3 + p_4 + c = 10 \times 512 + 126 + 6 = 5252 \\
 p_4 &= 5252 \bmod 10^4 = 5252 \\
 c &= \lfloor 5252/10^4 \rfloor = 0 \\
 p_5 &= c = 0
 \end{aligned}$$

Step 3. Integers A and B have opposite signs, therefore $sign = '-'$

Step 4. $p_{k-1} = p_5 = 0$, therefore $k = 5$. Hence:

$$P = sign \ p_{k-1}p_{k-2} \dots p_1p_0 = sign \ p_4p_3p_2p_1p_0 = -52520471139318298875$$

11.5.3 Representing long integers in computer memory

If we are to implement algorithms A and M on a computer, we need to devise a way of representing long integers in computer memory. To this end consider the integer

$$A = \sum_{i=0}^{m-1} a_i r^i = a_0 r^0 + a_1 r^1 + a_2 r^2 + \dots + a_{m-1} r^{m-1}$$

where r is a suitably chosen radix and $0 \leq |a_i| \leq r - 1$, $i = 0, 1, \dots, m - 1$; $a_{m-1} \neq 0$. We recognize this representation of A as a polynomial in r . Some operations on A , such as addition and multiplication, require that we process the quantities a_i in the order a_0, a_1, \dots, a_{m-1} , as we have seen in the above examples. On the other hand, the natural sequence in which to process these quantities during input/output operations is $a_{m-1}, a_{m-2}, \dots, a_0$. These observations suggest that we represent A as a doubly-linked linear list in the manner of Figure 11.6, as depicted in the figure below.

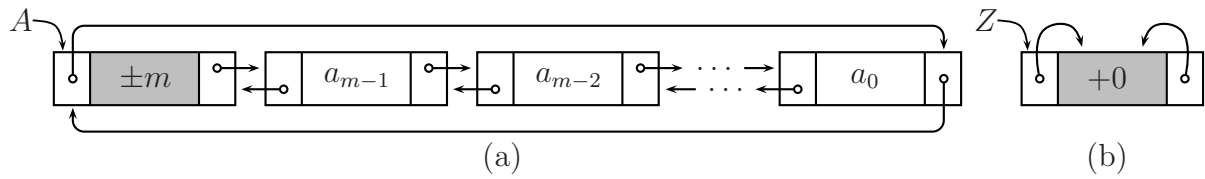


Figure 11.9 (a) The integer $A = a_{m-1}a_{m-2} \dots a_1a_0$ (b) The zero integer Z

We take note of the following conventions pertaining Figure 11.9.

1. The *DATA* field of the list head contains $+m$ to indicate that A is a positive integer, and $-m$ otherwise, where m is the number of terms in the polynomial expansion of A with radix r .
2. Since the sign of the integer is stored in the list head, the coefficients a_i are taken to be unsigned quantities.
3. The zero integer is represented by a list head with $+0$ stored in its *DATA* field.

Figure 11.10 shows two representations of the integer $B = -749865712998197$ with $r = 10^4$ and $r = 10^9$. In a computing environment where the primitive integer data type is represented using four bytes, we can use a radix r as large as 10^9 if the only operations performed are addition and subtraction; with multiplication and division, we may use a radix no larger than 10^4 , assuming that r is a multiple of 10.

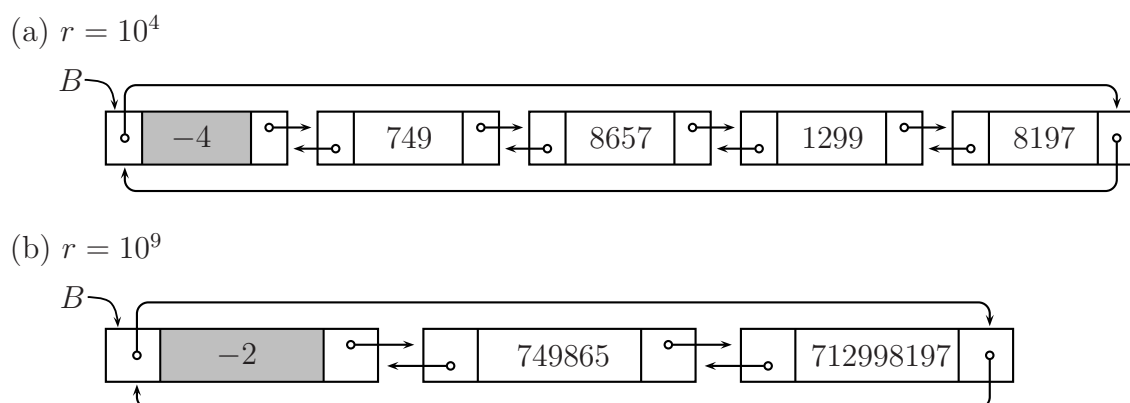


Figure 11.10 Two linked-list representations of the integer $B = -749865712998197$

11.5.4 EASY procedures for multiple-precision integer arithmetic

With long integers represented as in Figure 11.9, we can now implement the algorithms to perform such operations on long integers as addition, multiplication, and the like. As with polynomial arithmetic, we also need to write routines for such housekeeping tasks as reading in an integer and generating its internal representation, displaying an integer on some output medium, creating a zero integer, deleting an integer (i.e., returning all its nodes, including list head, to the memory pool), and so on. In the remainder of this section we will implement some of these operations; we leave the rest as exercises.

In the EASY procedures which follow, we assume that an integer, say L , is represented using the data structure

$$\mathbb{L} = [(LLINK, DATA, RLINK), l]$$

The first component specifies the node structure in the list representation of the integer; the second component, l , contains the address of the list head, or equivalently, it is the pointer to the integer.

The following EASY procedures are illustrative of the kind of operations usually performed on doubly-linked lists, for instance, traversing the list forward or backward, inserting a node at the head or tail end of the list, deleting a node from the list, and so on. Study the procedures carefully.

1. *Generating the internal representation of an input integer*

The procedure READ_LONGINTEGER(\mathbb{L}) reads in groups of decimal digits, each representing a term in the polynomial expansion of the input integer with radix r , and generates the internal representation of the integer as a doubly-linked circular list with a list head. The terms are inputted from the most to the least significant. The most significant term bears the sign of the integer; the rest are unsigned. For instance, if the integer is -749865712998197 and $r = 10^4$, then the input will be: -749 , 8657 , 1299 and 8197 . The resulting list is that shown in Figure 11.10(a).

```

1  procedure READ_LONGINTEGER( $\mathbb{L}$ )
2  call ZERO_LONGINTEGER( $\mathbb{L}$ )
3   $nterms \leftarrow 0$ 
4   $sign \leftarrow +1$ 
5  while not EOI do
6    input  $term$ 
7    if  $nterms = 0$  and  $term < 0$  then [ $sign \leftarrow -1$ ;  $term \leftarrow |term|$ ]
8    call INSERT_AT_TAIL( $\mathbb{L}, term$ )
9     $nterms \leftarrow nterms + 1$ 
10 endwhile
11  $\mathbb{L}.DATA(\mathbb{L}.l) \leftarrow sign * nterms$ 
12 end READ_LONGINTEGER

```

```

1  procedure ZERO_LONGINTEGER( $\mathbb{L}$ )
2  call GETNODE( $\tau$ )
3   $\mathbb{L}.l \leftarrow \tau$ 
4   $\mathbb{L}.DATA(\tau) \leftarrow 0$ 
5   $\mathbb{L}.LLINK(\tau) \leftarrow \tau$ 
6   $\mathbb{L}.RLINK(\tau) \leftarrow \tau$ 
7  end ZERO_LONGINTEGER

```

```

1  procedure INSERT_AT_TAIL( $\mathbb{L}, term$ )
2  call GETNODE( $\tau$ )
3   $\mathbb{L}.DATA(\tau) \leftarrow term$ 
4   $\mathbb{L}.RLINK(\mathbb{L}.LLINK(\mathbb{L}.l)) \leftarrow \tau$ 
5   $\mathbb{L}.RLINK(\tau) \leftarrow \mathbb{L}.l$ 
6   $\mathbb{L}.LLINK(\tau) \leftarrow \mathbb{L}.LLINK(\mathbb{L}.l)$ 
7   $\mathbb{L}.LLINK(\mathbb{L}.l) \leftarrow \tau$ 
8  end INSERT_AT_TAIL

```

Procedure 11.13 Generating the internal representation of a long integer

The call to ZERO_LONGINTEGER in line 2 creates a zero integer \mathbb{L} , pointed to by $\mathbb{L}.l$; this becomes the list head. Subsequently, each term is read in and is inserted into the list via a call to INSERT_AT_TAIL (lines 5–10). Insertion is at the tail end of the list since the terms are read in from most to least significant. Finally, the sign of the integer and the number of terms in its polynomial expansion are stored in the *DATA* field of the list head (line 11).

2. Multiple-precision integer addition

The procedure ADD_LONGINTEGERS($\mathbb{A}, \mathbb{B}, \mathbb{S}$) performs the operation $S \leftarrow A + B$ according to Algorithm A. The integers A and B can have any sign and can be of any length.

```

1  procedure ADD_LONGINTEGERS ( $\mathbb{A}, \mathbb{B}, \mathbb{S}$ )
  ▷ Find temporary working signs
2  if IsGT( $\mathbb{A}, \mathbb{B}$ ) then [ $s_A \leftarrow +1$ 
3                      if  $\mathbb{A}.DATA(\mathbb{A}.l) * \mathbb{B}.DATA(\mathbb{B}.l) > 0$  then  $s_B \leftarrow +1$ 
4                      else  $s_B \leftarrow -1$ ]
5                      else [ $s_B \leftarrow +1$ 
6                      if  $\mathbb{A}.DATA(\mathbb{A}.l) * \mathbb{B}.DATA(\mathbb{B}.l) > 0$  then  $s_A \leftarrow +1$ 
7                      else  $s_A \leftarrow -1$ ]
  ▷ Find actual sign of sum
8  if ( $\mathbb{A}.DATA(\mathbb{A}.l) > 0$  and  $\mathbb{B}.DATA(\mathbb{B}.l) > 0$ )
   or (IsGT( $\mathbb{A}, \mathbb{B}$ ) and  $\mathbb{A}.DATA(\mathbb{A}.l) > 0$ )
   or (IsGT( $\mathbb{B}, \mathbb{A}$ ) and  $\mathbb{B}.DATA(\mathbb{B}.l) > 0$ ) then  $sign \leftarrow +1$ 
9                      else  $sign \leftarrow -1$ 
  ▷ Perform addition
10 call ZERO_LONGINTEGER( $\mathbb{S}$ )    ▷ create list head for sum
11  $k \leftarrow 0$     ▷ number of terms in sum
12  $c \leftarrow 0$     ▷ carry
13  $\alpha \leftarrow \mathbb{A}.LLINK(\mathbb{A}.l)$     ▷ least significant term in A
14  $\beta \leftarrow \mathbb{B}.LLINK(\mathbb{B}.l)$     ▷ least significant term in B
15 loop
16   case
17     :  $\alpha \neq \mathbb{A}.l$  and  $\beta \neq \mathbb{B}.l$  : [ $t \leftarrow s_A * \mathbb{A}.DATA(\alpha) + s_B * \mathbb{B}.DATA(\beta) + c$ 
18                                      $\alpha \leftarrow \mathbb{A}.LLINK(\alpha)$ ;  $\beta \leftarrow \mathbb{B}.LLINK(\beta)$ ]
19     :  $\alpha \neq \mathbb{A}.l$  and  $\beta = \mathbb{B}.l$  : [ $t \leftarrow s_A * \mathbb{A}.DATA(\alpha) + c$ ;  $\alpha \leftarrow \mathbb{A}.LLINK(\alpha)$ ]
20     :  $\alpha = \mathbb{A}.l$  and  $\beta \neq \mathbb{B}.l$  : [ $t \leftarrow s_B * \mathbb{B}.DATA(\beta) + c$ ;  $\beta \leftarrow \mathbb{B}.LLINK(\beta)$ ]
21     :  $\alpha = \mathbb{A}.l$  and  $\beta = \mathbb{B}.l$  : [if  $c \neq 0$  then [call INSERT_AT_HEAD( $\mathbb{S}, c$ )
22                                      $k \leftarrow k + 1$ ]
23                                     else call DELETE_LEADING_ZEROS( $\mathbb{S}, k$ )
24                                      $\mathbb{S}.DATA(\mathbb{S}.l) \leftarrow sign * k$ 
25                                     return]
26   endcase
27    $term \leftarrow t \bmod r$ 
28   call INSERT_AT_HEAD( $\mathbb{S}, term$ )
29    $c \leftarrow \lfloor t/r \rfloor$ 
30    $k \leftarrow k + 1$ 
31 forever
32 end ADD_LONGINTEGERS

```

Procedure 11.14 Implementing long integer addition using linked lists
(continued on next page)

▷ Returns **true** if $|A| > |B|$; else returns **false**.

```

1  procedure IsGT( $\mathbb{A}, \mathbb{B}$ )
2     $m \leftarrow |\mathbb{A}.DATA(\mathbb{A}.l)|$ 
3     $n \leftarrow |\mathbb{B}.DATA(\mathbb{B}.l)|$ 
4    case
5      :  $m > n$ : return (true)
6      :  $m < n$ : return (false)
7      :  $m = n$ : [ $\alpha \leftarrow \mathbb{A}.RLINK(\mathbb{A}.l)$ 
8                   $\beta \leftarrow \mathbb{B}.RLINK(\mathbb{B}.l)$ 
9                  while  $\alpha \neq \mathbb{A}.l$  do
10                     case
11                       :  $\mathbb{A}.DATA(\alpha) > \mathbb{B}.DATA(\beta)$ : return (true)
12                       :  $\mathbb{A}.DATA(\alpha) < \mathbb{B}.DATA(\beta)$ : return (false)
13                       :  $\mathbb{A}.DATA(\alpha) = \mathbb{B}.DATA(\beta)$ : [ $\alpha \leftarrow \mathbb{A}.RLINK(\alpha)$ 
14                                                          $\beta \leftarrow \mathbb{B}.RLINK(\beta)$ ]
15                     endcase
16                 endwhile
17                 return (false)]
18    endcase
19    end IsGT

```



```

1  procedure INSERT_AT_HEAD( $\mathbb{L}, term$ )
2    call GETNODE( $\tau$ )
3     $DATA(\tau) \leftarrow term$ 
4     $RLINK(\tau) \leftarrow \mathbb{L}.RLINK(\mathbb{L}.l)$ 
5     $LLINK(\tau) \leftarrow \mathbb{L}.l$ 
6     $\mathbb{L}.LLINK(\mathbb{L}.RLINK(\mathbb{L}.l)) \leftarrow \tau$ 
7     $\mathbb{L}.RLINK(\mathbb{L}.l) \leftarrow \tau$ 
8    end INSERT_AT_HEAD

```



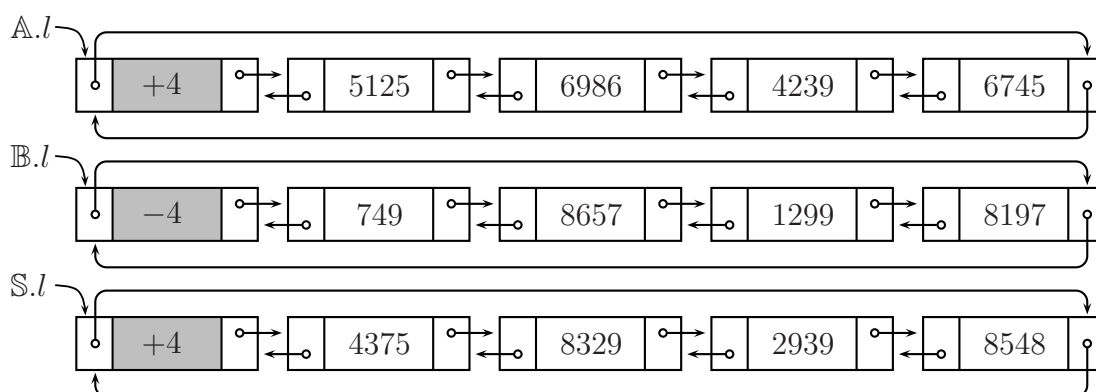
```

1  procedure DELETE_LEADING_ZEROS( $\mathbb{L}, nterms$ )
2     $\alpha \leftarrow \mathbb{L}.RLINK(\mathbb{L}.l)$ 
3    while  $\alpha \neq \mathbb{L}.l$  do
4      if  $\mathbb{L}.DATA(\alpha) = 0$  then [ $\mathbb{L}.RLINK(\mathbb{L}.l) \leftarrow \mathbb{L}.RLINK(\alpha)$ 
5                                 $\mathbb{L}.LLINK(\mathbb{L}.RLINK(\alpha)) \leftarrow \mathbb{L}.l$ 
6                                call RETNODE( $\alpha$ )
7                                 $nterms \leftarrow nterms - 1$ 
8                                 $\alpha \leftarrow \mathbb{L}.RLINK(\mathbb{L}.l)$ ]
9      else return
10    endwhile
11    end DELETE_LEADING_ZEROS

```

We take note of the following observations pertaining to procedure `ADD_LONG-INTEGERS` and the routines it invokes.

- (a) The procedure is a straightforward implementation of Algorithm A, with integers represented as doubly-linked lists as indicated in Figure 11.9.
- (b) The lists are traversed forwards and backwards. Procedure `IsGT`, invoked in lines 2 and 8, traverses the lists forwards, from most to least significant terms, via the *RLINK* field. The main loop which implements Eq.(11.2) traverses the lists backwards from least to most significant terms via the *LLINK* field (lines 13–31). As we have pointed out earlier, this is one reason why we use doubly-linked lists to represent long integers.
- (c) The **mod** function invoked in line 27 is the **mod** function as defined by Eq.(2.1) in Session 2. It is *not* equivalent to the modulus division operator `%` in C nor the elemental intrinsic function **mod** in FORTRAN.
- (d) The radix r used with the **mod** and **floor** functions in lines 27 and 29 is assumed to be a global variable. As we have previously pointed out, its value depends on how large the intrinsic integer type can be and on which arithmetic operations are supported (addition only or multiplication also).
- (e) Since the coefficients in the representation of an integer as a polynomial in r are stored as unsigned quantities (see Figure 11.9), the computation of t in step 2(a) of Algorithm A in lines 17, 19 and 20 does not require the use of the absolute value function.
- (f) It is important to delete leading zeros in the resulting sum (line 23) for two reasons: to conserve space and to make sure that `IsGT` always works correctly.
- (g) There is no special test for the special cases where A and/or B are zero; these are handled no differently from the ordinary, resulting in neater code. This is because we used a list with a list head to represent integers (see Figure 11.9).
- (h) The time complexity of the procedure is $O(m + n)$, where m and n are the lengths of the lists representing integers A and B .
- (i) As an example, the figure below depicts the input lists representing the integers $+5,125,698,642,396,745$ and $-749,865,712,998,197$ with $r = 10^4$ and the output list representing the sum $+4,375,832,929,398,548$. Note that Procedure 11.14 can add or subtract *any* two integers, no matter how long, subject only to constraints on available memory.



3. Multiple-precision integer multiplication

The procedure MULTIPLY_LONGINTEGERS($\mathbb{A}, \mathbb{B}, \mathbb{P}$) performs the operation $P \leftarrow A * B$ according to Algorithm M. The integers A and B can have any sign and can be of any length.

```

1  procedure MULTIPLY_LONGINTEGERS( $\mathbb{A}, \mathbb{B}, \mathbb{P}$ )
2   $k \leftarrow |\mathbb{A}.DATA(\mathbb{A}.l)| + |\mathbb{B}.DATA(\mathbb{B}.l)|$        $\triangleright k = m + n$ 
3  call INIT_PRODUCT( $\mathbb{P}, k$ )
4   $\tau \leftarrow \mathbb{P}.l$ 
5   $\alpha \leftarrow \mathbb{A}.LLINK(\mathbb{A}.l)$ 
6  while  $\alpha \neq \mathbb{A}.l$  do
7     $a \leftarrow \mathbb{A}.DATA(\alpha)$ 
8     $\beta \leftarrow \mathbb{B}.LLINK(\mathbb{B}.l)$ 
9     $\gamma \leftarrow \mathbb{P}.LLINK(\tau)$ 
10    $\tau \leftarrow \gamma$ 
11    $c \leftarrow 0$ 
12   while  $\beta \neq \mathbb{B}.l$  do
13      $t \leftarrow a * \mathbb{B}.DATA(\beta) + \mathbb{P}.DATA(\gamma) + c$ 
14      $\mathbb{P}.DATA(\gamma) \leftarrow t \bmod r$ 
15      $c \leftarrow \lfloor t/r \rfloor$ 
16      $\beta \leftarrow \mathbb{B}.LLINK(\beta)$ 
17      $\gamma \leftarrow \mathbb{P}.LLINK(\gamma)$ 
18   endwhile
19    $\mathbb{P}.DATA(\gamma) \leftarrow c$ 
20    $\alpha \leftarrow \mathbb{A}.LLINK(\alpha)$ 
21 endwhile
    $\triangleright$  Delete excess zero term if  $k = m + n - 1$ 
22 call DELETE_LEADING_ZEROS( $\mathbb{P}, k$ )
23  $\mathbb{P}.DATA(\mathbb{P}.l) \leftarrow k$ 
    $\triangleright$  Establish sign of product
24 if  $\mathbb{A}.DATA(\mathbb{A}.l) * \mathbb{B}.DATA(\mathbb{B}.l) < 0$  then  $\mathbb{P}.DATA(\mathbb{P}.l) \leftarrow -\mathbb{P}.DATA(\mathbb{P}.l)$ 
25 end MULTIPLY_LONGINTEGERS

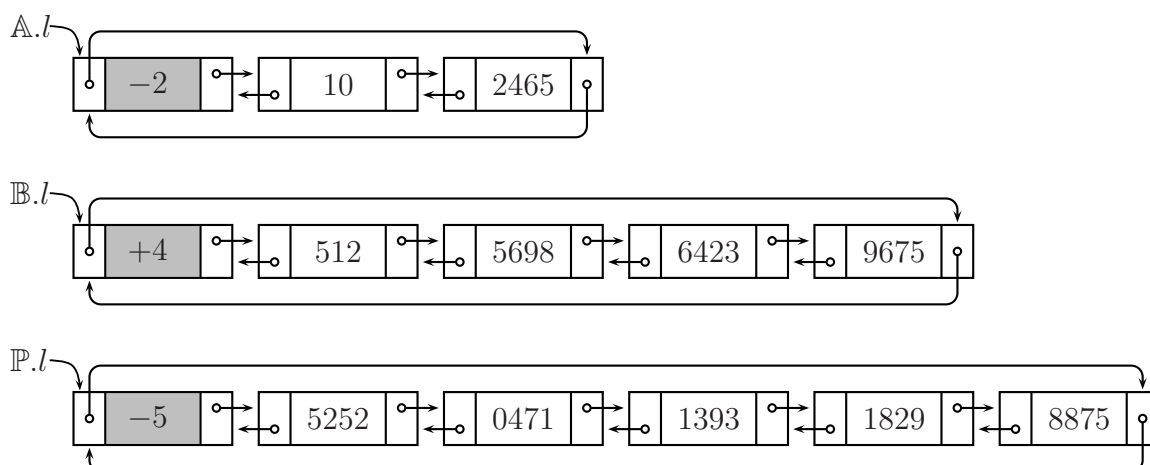
1  procedure INIT_PRODUCT( $\mathbb{P}, k$ )
2  call ZERO_LONGINTEGER( $\mathbb{P}$ )
3   $zero \leftarrow 0$ 
4  for  $i \leftarrow 0$  to  $k - 1$  do
5    call INSERT_AT_HEAD( $\mathbb{T}, zero$ )
6  enddo
7  end INIT_PRODUCT

```

Procedure 11.15 Implementing long integer multiplication using linked lists

We take note of the following observations pertaining to procedure MULTIPLY_LONGINTEGERS and the routines it invokes.

- (a) The procedure is a straightforward implementation of Algorithm M, with integers represented as doubly-linked lists as indicated in Figure 11.9.
- (b) The call to INIT_PRODUCT in line 3 creates a list representing the product P with $k = m + n$ nodes to store the $m + n$ possible terms of P . In accordance with step 1 of algorithm M, the *DATA* field of each node is set to zero (lines 3–6 of INIT_PRODUCT).
- (c) It may turn out that the product has $m + n - 1$ terms only; in such an event, the excess leading zero term is deleted from the list \mathbb{P} (lines 22–23) via a call to DELETE_LEADING_ZEROS (the same procedure invoked by ADD_LONGINTEGERS).
- (d) Arbitrarily, we take integer A as the multiplier and B as the multiplicand.
- (e) In a computing environment which uses four bytes to store the intrinsic integer data type, the largest product of two terms that is within the representable range is 9999×9999 , assuming that we choose the radix r as a multiple of 10 (verify). This implies that we should use $r = 10^4$.
- (f) There is no special test for the special cases where A and/or B are zero; these are handled no differently from the ordinary, resulting in neater code. This is because we used a list with a list head to represent integers (see Figure 11.9).
- (g) The sign of the product P is established in line 24 and stored in the *DATA* field of the list head of list \mathbb{P} .
- (h) The time complexity of the procedure is $O(mn)$ where m and n are the lengths of the lists representing integers A and B .
- (i) As an example, the figure below depicts the input lists representing the integers $-10,2465$ and $512,569,864,239,675$ with $r = 10^4$ and the output list representing the product $-52,520,471,139,318,298,875$. Note that Procedure 11.15 can multiply *any* two integers, no matter how long, subject only to constraints on available memory.



11.6 Linear lists and dynamic storage management

Up until now, whenever we used linked allocation for our structures, whether these be binary trees, forests or lists, we used fix-sized nodes which we obtained from the memory pool through a call to GETNODE and which, when no longer needed, we returned to the memory pool through a call to RETNODE. This is fine as long as the particular application requires *uniform-sized* nodes, as in the examples discussed thus far, namely, polynomial arithmetic and multiple-precision integer arithmetic. However, there are various other applications in which we need the capability to allocate and deallocate contiguous areas of memory *of variable size*, called **blocks**, *at run time*, upon request. For instance, an operating system needs to allocate space to different programs executing concurrently in a multiprogramming environment, and to deallocate the same when the program is done. An application software needs dynamically allocated space for such objects as windows, spreadsheets, menus, pictures, and the like.

Dynamic storage management, or DSM, refers to the management of a contiguous area of memory, called the **memory pool** (aka ‘heap’, which should not be confused with the heap in heapsort) through various techniques for allocating and deallocating blocks in the pool. In the discussion which follows, we assume that the memory pool consists of individually addressable units called **words** and that block sizes are measured in number of words. (Note: in an actual computing environment a word normally consists of $c > 1$ addressable units.)

There are two closely related operations in DSM:

1. **reservation** — allocate a block of memory of the requested size (or more) to a requesting task
2. **liberation** — return to the memory pool a block that is no longer needed

The fact that blocks are of variable size, ranging from a few words in certain applications to thousands of words in others, makes these two operations somewhat more complicated than GETNODE and RETNODE. Different approaches to DSM have been proposed and implemented, their relative performance have been analyzed under various conditions (e.g., block size distribution, block lifetimes distribution, time intervals between reservations and/or liberations, etc.), mainly through simulation. Here, we will consider only two categories of DSM techniques, viz., the **sequential-fit methods** and the **buddy-system methods**.

11.6.1 The memory pool

We can imagine the memory pool as initially consisting of one available block of contiguous memory locations. After a sequence of reservations and liberations, the memory pool will be *fragmented*, consisting of **free blocks** interspersed with **reserved blocks**. A free block is one that is available for allocation; a reserved block is one that is currently being used by some task and is not therefore available for allocation. Figure 11.11 shows a schematic representation of the memory pool after DSM has been in effect for some time, with reserved blocks shown shaded.



Figure 11.11 The memory pool, with DSM in progress

A fundamental question in DSM is how to organize the free blocks so that reservations and liberations can be efficiently performed. The traditional answer is: use linked lists.

In the sequential-fit methods, all the free blocks are constituted into a single linked list, called the *list of available blocks* or, simply, the *avail list* or the *free list*. In the buddy-system methods, blocks are allocated in ‘quantum’ sizes only, e.g., in sizes of $1, 2, 4, 8, 16, 32, \dots, 2^k$ words. Consequently, several *avail* lists, one for each allowable size, are maintained.

11.6.2 Sequential-fit methods: reservation

One way to constitute the *avail* list for the sequential-fit methods is as shown in Figure 11.12. The first word of each free block is drafted as a **control word**, and is divided into two fields, *SIZE* and *LINK*. The *SIZE* field contains the size of the block in words; the *LINK* field contains a pointer to the next free block in the memory pool. The list may be sorted according to block addresses, block sizes, or it may not be sorted at all.

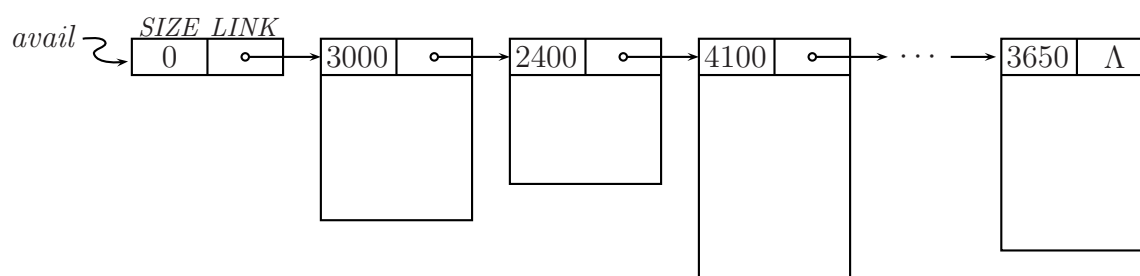


Figure 11.12 The list of available blocks, or *avail* list

To satisfy a request for n words, we scan the *avail* list and choose a block which meets a particular ‘fitting’ criterion, such as:

1. first-fit method— we choose the first block we encounter which has $m \geq n$ words
2. best-fit method — we choose the smallest block which has $m \geq n$ words
3. optimal-fit method — as we scan the *avail* list, we build up a sample of free blocks which satisfy the requested size; then we choose the first block past the end of the sample which fits the requested size better than the best fitting block in the sample. If no such block is found, we use the best fitting block in the sample.

4. worst-fit method — we choose the largest block which has $m \geq n$ words

The motivation behind the first-fit method is to minimize search time on the *avail* list; the moment a block large enough to satisfy the requested size is found, the search terminates and a portion of the block, or all of it, is reserved. On the other hand, the motivation behind the best-fit method is to save the larger blocks for later requests; thus the entire *avail* list is searched for the best fitting block before any reservation is done, unless of course an exact-fitting block is found sooner. The best-fit method has two obvious disadvantages: (a) longer search times on the *avail* list, and (b) accumulation of small blocks on the *avail* list, the remnants of tight fits, which results in yet longer searches.

The optimal-fit method attempts to strike a compromise between first-fit and best-fit; it does not settle for the first block that satisfies the request but it does not go for the best fitting block either. (There are obvious parallels to this strategy in real life.) The crucial parameter is the sample size, i.e., how many blocks must be sampled first before a reservation is made. STANDISH[1980], pp. 255–256, gives as a good approximation the quantity $\lceil l/e \rceil$, where l is the number of blocks on the *avail* list and $e = 2.5576052$.

Finally, the idea behind the worst-fit method is to avoid the proliferation of small blocks on the *avail* list. By always doing the reservations from the largest available blocks, it is anticipated that the remnants will be large enough blocks to satisfy many more subsequent requests.

Which one of these four strategies for reservation is recommended? It is relatively easy to invent scenarios in which one method out-performs another in that the latter is unable to satisfy a request whereas the former can. Figure 11.13 depicts one such scenario in which first-fit out-performs best-fit. Of course it is equally easy to come up with an example which shows the opposite, or even with an example which shows worst-fit out-performing both first-fit and best-fit (try both). On a larger, more realistic scale however, the methods turn out to be rather difficult to analyze mathematically. Thus comparative studies of the methods rely primarily on *simulations* in order to characterize their behavior and arrive at quantitative performance measures.

Request	First-fit	Best-fit
—	1700, 1100	1700, 1100
900	800, 1100	1700, 200
1000	800, 100	700, 200
750	50, 100	—stuck—

Figure 11.13 An example in which first-fit out-performs best-fit

Simulations by Knuth indicated that first-fit ‘always appeared to be superior’ to best-fit. Other investigators have found conditions under which best-fit did better than first-fit. For a detailed comparison of the performance of these various sequential-fit methods for reservation, refer to the Readings. *Ceteris paribus*, first-fit is the recommended method.

EASY implementation of first-fit

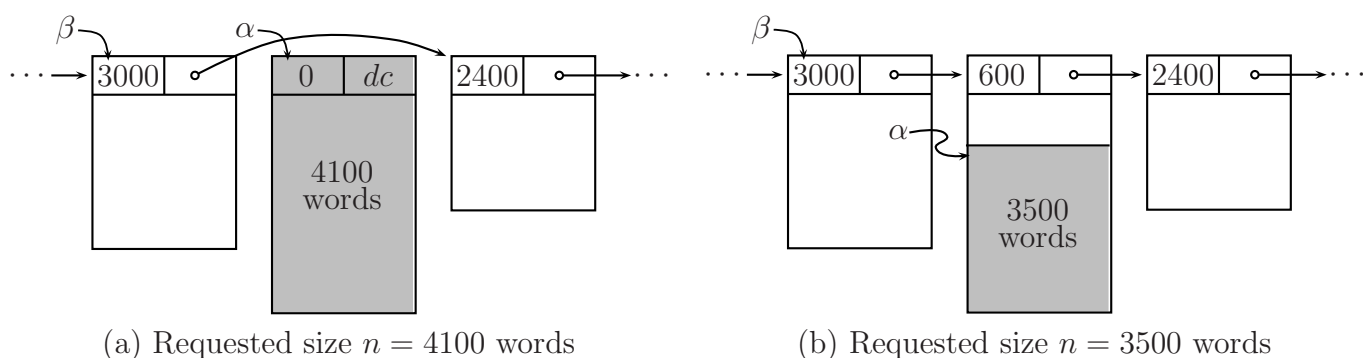
\triangleright Returns the address of a block of size n acquired from the first-fitting block in the
 \triangleright avail list; returns Λ if no such block is found.

```

1  procedure FIRST_FIT_1( $n$ )
2     $\alpha \leftarrow \text{LINK}(\text{avail})$        $\triangleright$  pointer to current block
3     $\beta \leftarrow \text{avail}$                $\triangleright$  trailing pointer
4    while  $\alpha \neq \Lambda$ 
5      if  $\text{SIZE}(\alpha) \geq n$  then [ $\text{SIZE}(\alpha) \leftarrow \text{SIZE}(\alpha) - n$ 
6                                if  $\text{SIZE}(\alpha) = 0$  then  $\text{LINK}(\beta) \leftarrow \text{LINK}(\alpha)$ 
7                                else  $\alpha \leftarrow \alpha + \text{SIZE}(\alpha)$ 
8                                return( $\alpha$ )]
9      else [ $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \text{LINK}(\alpha)$ ]
10   endwhile
11   return( $\Lambda$ )
12 end FIRST_FIT_1
  
```

Procedure 11.16 First-fit method for reservation in DSM (initial version)

Procedure FIRST_FIT_1 is a straightforward implementation of the first-fit method for reservation. It scans the free list, starting from the first block, until a block of size greater than or equal to the requested size n is found (line 5). The entire block is reserved if it exactly fits the requested size by deleting it from the *avail* list (line 6); otherwise, the ‘bottom’ part, i.e., the portion in the *higher* (address-wise) region of memory is reserved (line 7), retaining whatever is left of it on the list. In either case, the address α of the first word of the reserved block is returned to the requesting task (line 8), thus giving the task access to this area of the memory pool. If there is no block on the free list which satisfies the request, the procedure returns the null pointer (line 11).

**Figure 11.14** Reserving a block (sequential-fit methods)

There are two aspects of the above procedure which merit closer scrutiny, namely:

1. It retains on the *avail* list whatever is left of a block, no matter how small, after a portion of the block is reserved. Eventually, this will result in the accumulation of small blocks on the *avail* list. In all likelihood such blocks will not satisfy any request; they simply clutter the *avail* list and cause longer searches on the list. This undesirable phenomenon in DSM is called **external fragmentation**.

2. Search always begins at the head of the *avail* list, leaving smaller and smaller remnants of reserved blocks in the leading portion of the list. This, again, results in longer searches on the list since these small blocks are simply skipped until a large enough block is found.

A simple way to counteract external fragmentation is to reserve the entire block when what would otherwise be left of it is smaller than a specified minimum size, say *minsize*. This strategy has two important consequences:

- (a) There is now unused space in reserved blocks. Although the space is unused it is not available for allocation since it resides in a reserved block. This phenomenon in DSM is called **internal fragmentation**. Thus avoiding one kind of memory fragmentation results in fragmentation of another kind.
- (b) Since the requested size, n , may no longer tally with the *actual* size of the reserved block, it now becomes necessary to store the actual size somewhere. For instance, the size of a reserved block is needed when the block is liberated. The logical place in which to keep this information is inside the reserved block itself, right in the *SIZE* field. This means that both free and reserved blocks will have a *SIZE* field.

A simple way to solve the second problem cited above is to start the search for a block where the previous search ended. Specifically, if the last reservation was made from block α , we could start the next search from block $LINK(\alpha)$. This will result in smaller blocks being evenly distributed over the entire *avail* list. We can retain the address α in a ‘roving’ pointer, say ρ .

Procedure FIRST_FIT_2 incorporates these suggested modifications.

```

1  procedure FIRST_FIT_2( $n, \rho, minsize$ )
2     $\alpha \leftarrow LINK(\rho)$ 
3     $\beta \leftarrow \rho$ 
4     $flag \leftarrow 0$ 
5    while  $\beta \neq \rho$  or  $flag = 0$  do
6      if  $\alpha = \Lambda$  then [ $\alpha \leftarrow LINK(avail); \beta \leftarrow avail; flag \leftarrow 1$ ]
7      if  $SIZE(\alpha) \geq n$  then [ $excess \leftarrow SIZE(\alpha) - n$ 
8                                if  $excess < minsize$  then [ $LINK(\beta) \leftarrow LINK(\alpha)$ 
9                                                          $\rho \leftarrow \beta$ ]
10                               else [ $SIZE(\alpha) \leftarrow excess$ 
11                                        $\rho \leftarrow \alpha$ 
12                                        $\alpha \leftarrow \alpha + excess$ 
13                                        $SIZE(\alpha) \leftarrow n$ ]
14                               return( $\alpha$ )]
15      else [ $\beta \leftarrow \alpha; \alpha \leftarrow LINK(\alpha)$ ]
16    endwhile
17    return( $\Lambda$ )
18  end FIRST_FIT_2

```

Procedure 11.17 First-fit method for reservation in DSM (improved version)

Since with a roving pointer ρ the search now starts where the last search ended (lines 2–3), it is possible that we will reach the end of the free list without finding a block with the requisite size, although there may be such a block in the leading part of the list. For this reason we re-start the search at the head of the list (line 6), but we should make sure that we do not go around in circles in the event that no block in the entire list satisfies the request. This is the function of the boolean variable *flag* in the test in line 5.

Lines 8–9 implement the strategy for reducing external fragmentation whereby the entire block is reserved if what would be left of it after reservation is less than the parameter *minsize*. Note that the *SIZE* field is retained in a reserved block and that it contains the true size of the block (lines 7 and 13).

With these modifications incorporated into FIRST_FIT_2, how do these two implementations of the first-fit method compare? Simulations by Knuth indicated that the latter implementation performed much better than the former in that it required significantly much shorter searches on the free list. Consequently, you may now discard FIRST_FIT_1 and rename FIRST_FIT_2 to simply FIRST_FIT.

The time complexity of the first-fit method, as implemented in Procedure 11.17, is clearly $O(l)$, where l is the length of the *avail* list.

EASY implementation of best-fit

```

1  procedure BEST_FIT( $n, minsize$ )
2     $\alpha \leftarrow LINK(avail)$ 
3     $\beta \leftarrow avail$ 
4     $bestsize \leftarrow \infty$ 
5    while  $\alpha \neq \Lambda$  do
6      if  $SIZE(\alpha) \geq n$  then if  $bestsize > SIZE(\alpha)$  then [ $bestsize \leftarrow SIZE(\alpha); \tau \leftarrow \beta$ ]
7       $\beta \leftarrow \alpha$ 
8       $\alpha \leftarrow LINK(\alpha)$ 
9    endwhile
10   if  $bestsize = \infty$  then return( $\Lambda$ )
11   else [ $\alpha \leftarrow LINK(\tau)$   $\triangleright$  best-fitting block
12      $excess \leftarrow bestsize - n$ 
13     if  $excess < minsize$  then  $LINK(\tau) \leftarrow LINK(\alpha)$ 
14     else [ $SIZE(\alpha) \leftarrow excess$ 
15        $\alpha \leftarrow \alpha + excess$ 
16        $SIZE(\alpha) \leftarrow n$ ]
17   return( $\alpha$ )
18 end BEST_FIT

```

Procedure 11.18 Best-fit method for reservation in DSM

It is even more crucial, if not imperative, to apply the parameter *minsize* in BEST_FIT since there is a greater tendency for very small blocks, the remnants of tight fits, to proliferate. Of course the roving pointer ρ in FIRST_FIT_2 is no longer relevant in BEST_FIT.

The time complexity of the best-fit method, as implemented in Procedure 11.18, is clearly $O(l)$, where l is the length of the *avail* list.

11.6.3 Sequential-fit methods: liberation

When a reserved block is no longer needed, it is liberated. At the heart of the liberation operation is the **collapsing problem**: two adjacent free blocks must be collapsed, or merged, into one. Without collapsing during liberation to counteract fragmentation during reservation, DSM will soon grind to a halt. Figure 11.15 shows four possible scenarios which may arise when a reserved block, B, is freed and returned to the memory pool.

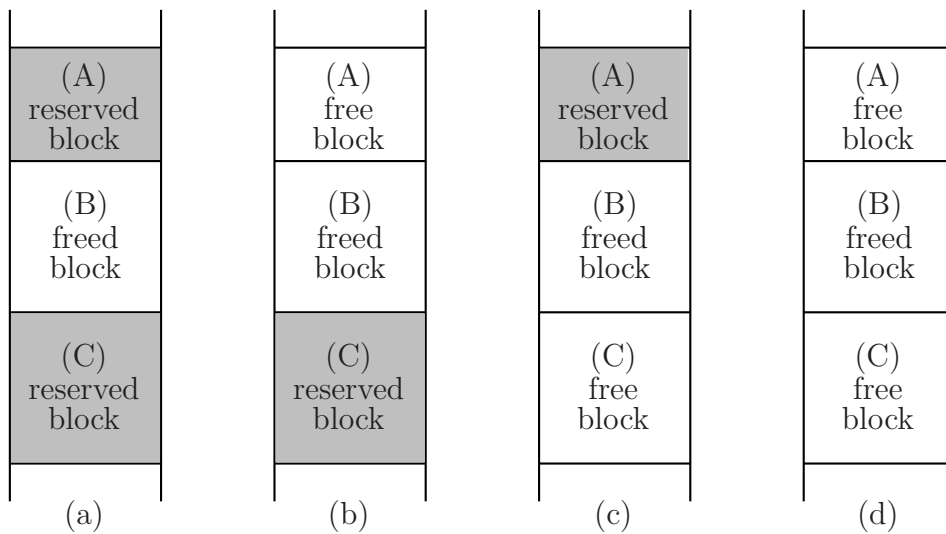


Figure 11.15 A freed block is collapsed with adjacent free blocks

In all four cases, we need to determine whether block B can be collapsed with block A; following Knuth, we refer to this as ‘checking the *lower* bound’ since A is in the lower (address-wise) region of memory. Likewise, we need to determine whether block B can be collapsed with block C; we refer to this as ‘checking the *upper* bound’ since C is in the higher (address-wise) region of memory. Two methods which perform collapsing at liberation are the **sorted-list technique** and the **boundary-tag technique**.

The sorted-list technique

In the sorted-list technique, the *avail* list is assumed to be constituted as shown in Figure 11.16 with the blocks sorted in order of increasing memory addresses. The *avail* list is traversed and the position of the liberated block, say block τ , with respect to the blocks on the list is determined. One of three cases may obtain:

- (a) the freed block comes before (i.e., is in a lower region of memory than) the first block on the free list
- (b) the freed block is straddled by two blocks on the free list
- (c) the freed block comes after (i.e., is in a higher region of memory than) the last block on the free list

Knowing the size of the blocks, which is stored in the *SIZE* field of both free and reserved blocks, it is easily determined whether the freed block is adjacent to any of the blocks on the free list. To collapse two adjacent free blocks, the *SIZE* field of the lower block is simply updated to contain the sum of the sizes of the combined blocks.

To be more specific, consider Figure 11.16 which depicts the case in which the freed block τ is straddled by two blocks on the *avail* list, i.e., $\beta < \tau < \alpha$. If $\tau + \text{SIZE}(\tau) = \alpha$, then clearly block τ and block α are adjacent and must therefore be collapsed into one; otherwise they are not adjacent. Likewise, if $\beta + \text{SIZE}(\beta) = \tau$, then clearly block β and block τ are adjacent and must therefore be collapsed into one; otherwise they are not adjacent.

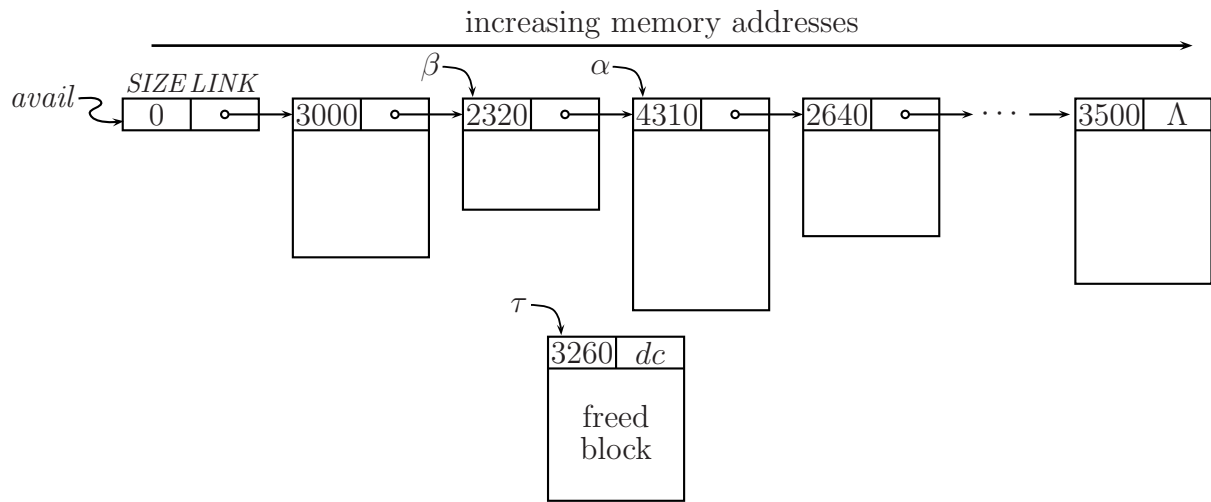


Figure 11.16 Liberation in the sorted-list technique: case (b)

▷ Returns block τ to the *avail* list. Collapsing is performed if possible.

```

1  procedure SORTED_LIST_TECHNIQUE( $\tau$ )
2     $n \leftarrow \text{SIZE}(\tau)$ 
3  ▷ Find position of block  $\tau$  with respect to blocks on the avail list.
4     $\alpha \leftarrow \text{LINK}(\text{avail})$ 
5     $\beta \leftarrow \text{avail}$ 
6    while  $\alpha \neq \Lambda$  and  $\alpha \leq \tau$  do
7       $\beta \leftarrow \alpha$ 
8       $\alpha \leftarrow \text{LINK}(\alpha)$ 
9    endwhile
10  ▷ Collapse with higher (address-wise) block, if free.
11    if  $\alpha \neq \Lambda$  and  $\tau + n = \alpha$  then [ $n \leftarrow n + \text{SIZE}(\alpha)$ ;  $\text{LINK}(\tau) \leftarrow \text{LINK}(\alpha)$ ]
12    else  $\text{LINK}(\tau) \leftarrow \alpha$ 
13  ▷ Collapse with lower (address-wise) block, if free.
14    if  $\beta + \text{SIZE}(\beta) = \tau$  then [ $\text{SIZE}(\beta) \leftarrow \text{SIZE}(\beta) + n$ ;  $\text{LINK}(\beta) \leftarrow \text{LINK}(\tau)$ ]
15    else [ $\text{LINK}(\beta) \leftarrow \tau$ ;  $\text{SIZE}(\tau) \leftarrow n$ ]
16  end SORTED_LIST_TECHNIQUE

```

Procedure 11.19 Sorted-list technique for liberation in DSM

Procedure SORTED_LIST_TECHNIQUE formalizes the process outlined above for liberation in DSM. The simplest way to understand how the procedure works is to trace its actions for each of the four scenarios depicted in Figure 11.15.

The sorted-list technique, as implemented in Procedure 11.19, has time complexity $O(l)$, where l is the length of the *avail* list. This is because we need to march down the list to locate the blocks with which the freed block may be merged (lines 3–8).

An $O(1)$ procedure for liberation in DSM is possible, albeit at the expense of an additional control word and double linking, as in the boundary-tag technique.

The boundary-tag technique

In this method, each block is bounded by two control words, as shown below. With these conventions, we are able to locate the blocks adjacent to the freed block and to

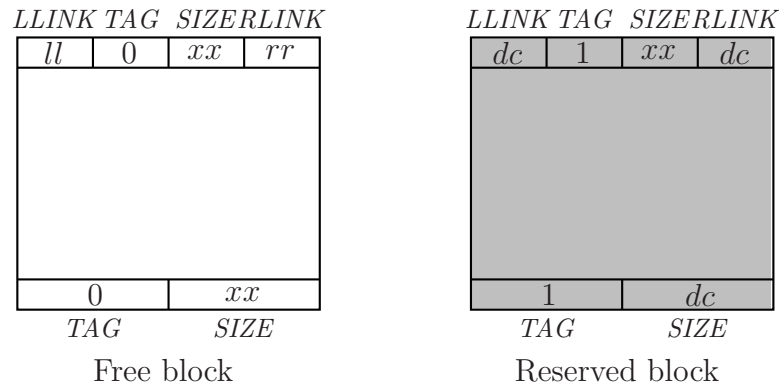


Figure 11.17 Block structure for the boundary-tag technique

determine whether they are reserved or free in $O(1)$ time, as depicted in Figure 11.18.

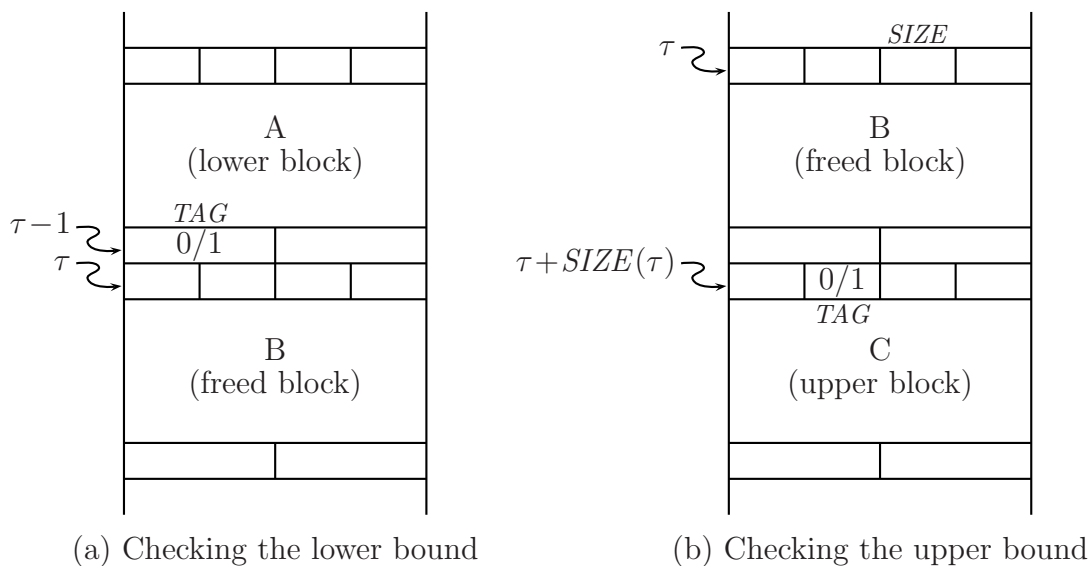


Figure 11.18 Liberation in the boundary-tag technique

In Figure 11.18(a), the condition $TAG(\tau - 1) = 0$ means block A is free, in which case it is collapsed with B. In Figure 11.18(b), the condition $TAG(\tau + SIZE(\tau)) = 0$ means block C is free, in which case it is collapsed with B. Note that blocks A and C are found in $O(1)$ time, without having to search the *avail* list.

Before B is collapsed with A and/or C, these blocks must first be deleted from the *avail* list. So that the deletion can be done in $O(1)$ time, the *avail* list is constituted as a doubly-linked list. As a final step block B, or B merged with A and/or C, is inserted into the *avail* list, also in $O(1)$ time.

Initially, the *avail* list consists of just one block, viz., the entire memory pool. This is shown in Figure 11.19. So that DSM will be confined to the memory pool only, the memory pool is bounded below and above by reserved blocks (which comprise the rest of memory not available for DSM), as indicated by the shaded control words.

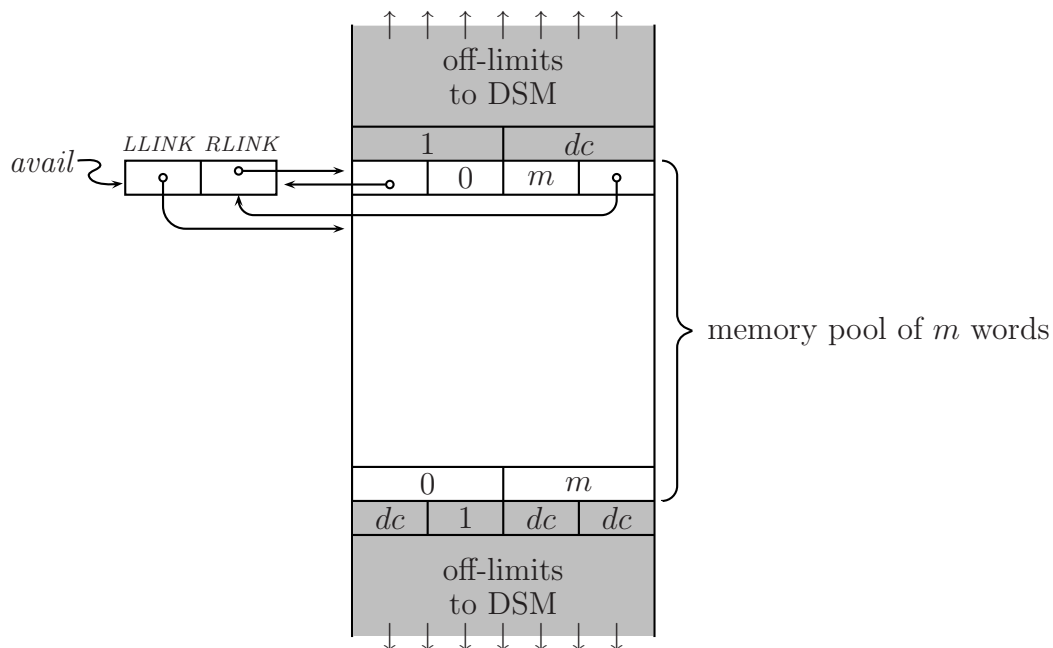


Figure 11.19 Initializing the *avail* list (boundary-tag technique)

After DSM has been in effect for some time, the memory pool will be fragmented and the *avail* list will appear as shown in Figure 11.20.

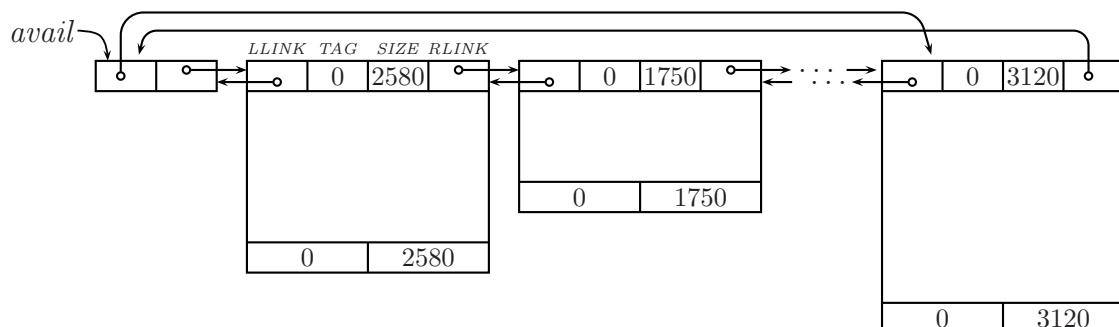


Figure 11.20 The *avail* list with DSM in progress (boundary-tag technique)

▷ Returns block τ to the avail list. Collapsing is performed if possible.
 1 **procedure** BOUNDARY_TAG _TECHNIQUE(τ)
 ▷ Collapse with lower (address-wise) block, if free.
 2 **if** $TAG(\tau - 1) = 0$ **then** [$\alpha \leftarrow \tau - SIZE(\tau - 1)$
 3 $RLINK(LLINK(\alpha)) \leftarrow RLINK(\alpha)$
 4 $LLINK(RLINK(\alpha)) \leftarrow LLINK(\alpha)$
 5 $SIZE(\alpha) \leftarrow SIZE(\alpha) + SIZE(\tau)$
 6 $\tau \leftarrow \alpha$]
 ▷ Collapse with higher (address-wise) block, if free.
 7 $\alpha \leftarrow \tau + SIZE(\tau)$
 8 **if** $TAG(\alpha) = 0$ **then** [$RLINK(LLINK(\alpha)) \leftarrow RLINK(\alpha)$
 9 $LLINK(RLINK(\alpha)) \leftarrow LLINK(\alpha)$
 10 $SIZE(\tau) \leftarrow SIZE(\tau) + SIZE(\alpha)$
 11 $\alpha \leftarrow \alpha + SIZE(\alpha)$]
 ▷ Attach block to avail list.
 12 $RLINK(\tau) \leftarrow RLINK(avail)$
 13 $LLINK(\tau) \leftarrow avail$
 14 $LLINK(RLINK(avail)) \leftarrow \tau$
 15 $RLINK(avail) \leftarrow \tau$
 ▷ Update control words.
 16 $TAG(\tau) \leftarrow 0$
 17 $SIZE(\alpha - 1) \leftarrow SIZE(\tau)$
 18 $TAG(\alpha - 1) \leftarrow 0$
 19 **end** BOUNDARY_TAG _TECHNIQUE

Procedure 11.20 Boundary-tag technique for liberation in DSM

Procedure BOUNDARY_TAG _TECHNIQUE(τ) returns a block, say B, to the *avail* list given the address τ of the block and its size. The quantity $\tau - 1$ in line 2 is the address of the bottom control word of block A [see Figure 11.18(a)]. If the *TAG* field of the control word contains 0 then A is free. To collapse A with B, we go through the following steps: (a) we find the address α of block A (line 2); (b) we detach block A from the *avail* list (lines 3–4); (c) we update the *SIZE* field of block A to contain the combined sizes of A and B to yield the single block AB (line 5), and (d) we set τ to point to AB (line 6) so that when we get to line 7, τ is a pointer to either B or AB.

The quantity α in line 7 is the address of the top control word of block C [see Figure 11.18(b)]. If the *TAG* field of the control word contains 0 then C is free. We detach C from the *avail* list (lines 8–9) and then collapse C with B or AB (line 10) to obtain BC or ABC. In line 11, we find the address of the block next to C (to be used in lines 17–18).

In lines 12–15, we insert block τ (i.e., B or AB or BC or ABC) at the head of the *avail* list. Finally, we update the top and bottom control words of block τ (lines 16–18).

The boundary-tag technique for liberation in DSM, as implemented in Procedure 11.20 has time complexity $O(1)$. This is because with two control words per block we are able to locate the neighbors of the freed block without having to scan the *avail* list, and with double linking we are able to detach a block from the *avail* list as needed in constant time.

11.6.4 Buddy-system methods: reservation and liberation

An interesting approach to dynamic memory management is exemplified by the class of DSM techniques called the *buddy-system methods*. Two distinguishing features of these methods, which make them rather different from the sequential-fit methods, are:

1. Blocks are allocated in ‘quantum’ sizes.

For instance, in the binary buddy-system method, blocks are allocated in sizes of $1, 2, 4, 8, 16, 32, \dots, 2^k$ words; in the Fibonacci buddy-system method, block allocation is in sizes of $1, 2, 3, 5, 8, 13, 21, \dots$ words. Thus, in the binary buddy-system method, a block of size 2048 words is allocated for any request of size 1025 through 2048. (This is not as irrational as it may appear to be.)

2. Multiple *avail* lists are maintained.

So that reservations and liberations can be serviced very fast, separate *avail* lists, one for each of the allowable sizes, are maintained. Thus, in the binary buddy-system method, there is an *avail* list for blocks of size 1 (at least in principle), another *avail* list for blocks of size 2, and so on.

In the rest of this section, we will consider the binary buddy-system method only.

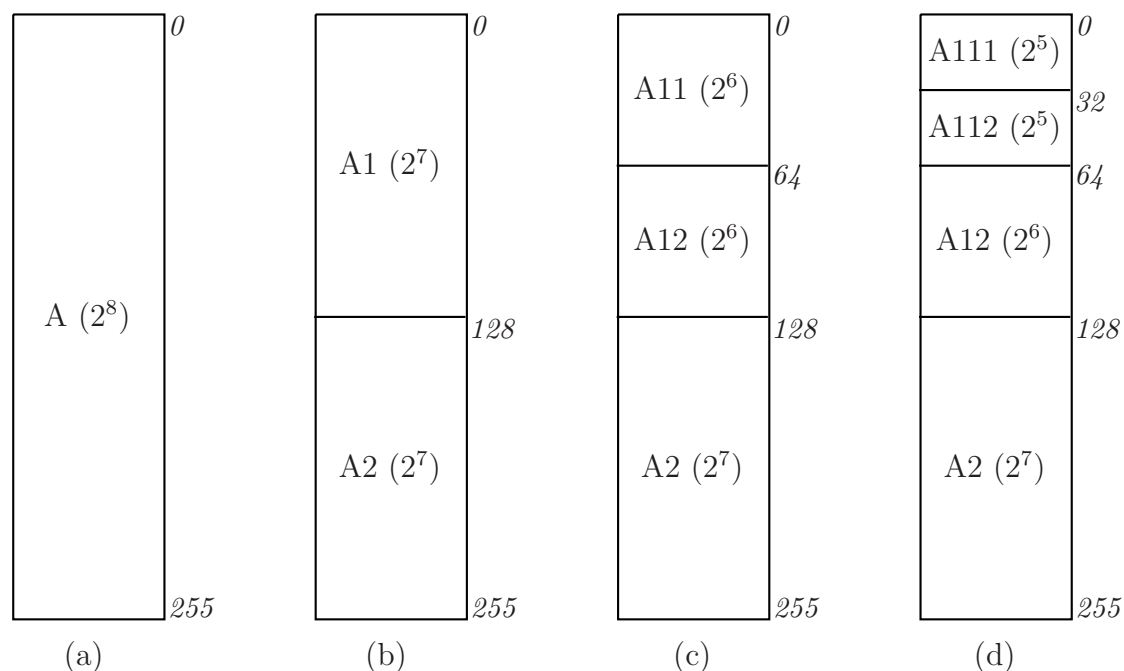


Figure 11.21 Buddies in the binary buddy-system method

Suppose that we have a memory pool of size $2^8 = 256$ words at relative addresses 0 through 255 and that all of it is available as shown in Figure 11.21(a). Now assume that a request for $n = 25$ words is made. Then, the block is split into two blocks of size $2^7 = 128$ words (Figure 11.21b); these blocks are called **buddies**. Either block is still too large, so the lower (address-wise) block is again split into two blocks of size $2^6 = 64$ words (Figure 11.21c), creating two smaller buddies. Either one is still too large, so the lower buddy is

split into two blocks of size $2^5 = 32$ words (Figure 11.21d). We have at last obtained a block whose size is the smallest multiple of 2 which is greater than or equal to $n = 25$, so the lower buddy (A111) is reserved.

During liberation, we reverse the process. If the buddy of the block being freed is free, then the two are collapsed, yielding a block twice as big. If the buddy of this new block is also free, then again the two are collapsed, yielding once more a new block twice as big. And so on. This sequence, whereby successively larger buddies are collapsed, terminates once one of the buddies is not free (i.e., it is wholly or partially reserved), or we have already reclaimed the entire memory pool!

Locating the buddy

A crucial step in the liberation operation in DSM is locating the blocks adjacent to the block being liberated so that, if possible, the blocks may be collapsed. In the sorted-list technique, this was accomplished in $O(l)$ time by searching the *avail* list; in the boundary-tag technique, this was accomplished in $O(1)$ time by using the first and last words of every block as control words. In the buddy-system methods, locating the adjacent block (the buddy) is done by *computation*.

Specifically, if we let $\beta(k, \alpha)$ represent the address of the (binary) buddy of the block of size 2^k at address α , then

$$\begin{aligned}\beta(k, \alpha) &= \alpha + 2^k && \text{if } \alpha \bmod 2^{k+1} = 0 \\ &= \alpha - 2^k && \text{otherwise}\end{aligned}\tag{11.3}$$

For instance, given block A111 at address $\alpha = 0$ with size 2^5 (see Figure 11.24), we locate its buddy, A112, as follows:

$$0 \bmod 2^6 = 0 \quad \text{therefore} \quad \beta = \alpha + 2^k = 0 + 2^5 = 32$$

Similarly, given block A112 at address $\alpha = 32$ with size 2^5 , we locate its buddy, A111, as follows:

$$32 \bmod 2^6 = 32 \neq 0 \quad \text{therefore} \quad \beta = \alpha - 2^k = 32 - 2^5 = 0$$

You may want to try these formulas on the buddies created by successively splitting block A2 in Figure 11.21.

Constituting the avail lists

If the buddy-system methods are to be operationally efficient, separate *avail* lists, one list for each allowable size, must be maintained. For instance, if a request for n words is made, then we get a block from the avail list for blocks of size 2^k , where $k = \lceil \log_2 n \rceil$. If this list happens to be empty, then we get a block from the 2^p -list, where p is the smallest integer greater than k for which the list is not empty. We then split the block $p - k$ times, inserting the unused halves into their appropriate lists.

During liberation, we locate the buddy, say β , of the block, say α , to be liberated by using Eq.(11.3). If block β happens to be free, we delete it from its *avail* list so that it can be collapsed with block α . So that we can perform the deletion in constant time, we

will constitute the *avail* lists as doubly-linked lists. The first word in each block will be used as a control word with the following fields:

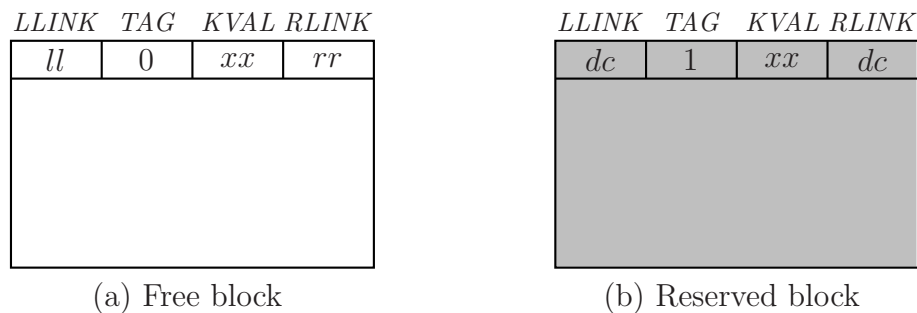


Figure 11.22 Block structure for the binary buddy-system method

Initializing the memory pool

Figure 11.23 shows the memory pool, assumed to be of size 2^m , as initialized for the binary buddy-system method. In principle, as many as $m + 1$ lists are set up, with the first m lists initially empty. The pointers $avail(0), avail(1), \dots, avail(m)$ to the lists are maintained sequentially in an array of size $m + 1$.

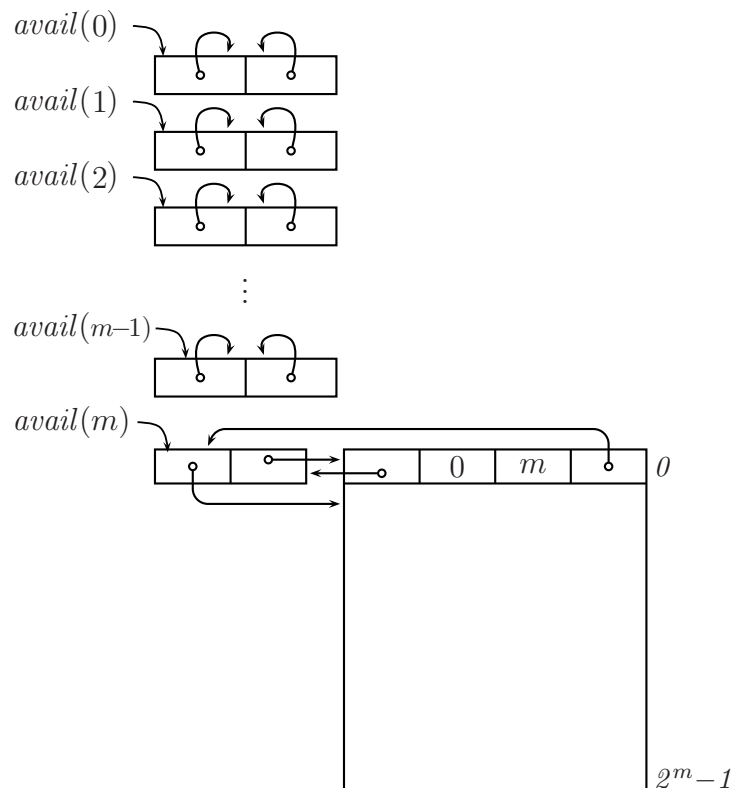
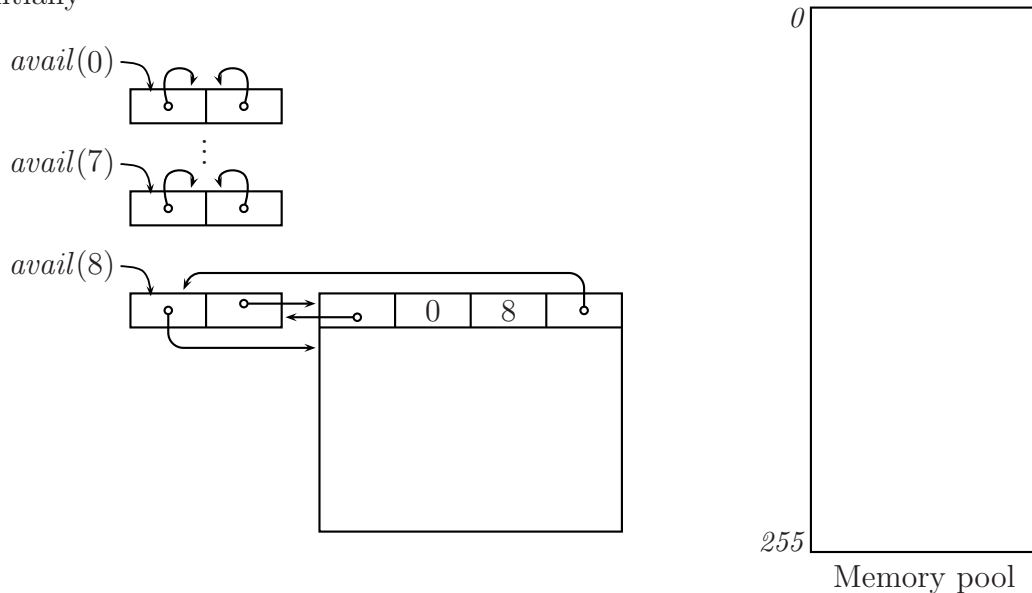


Figure 11.23 Initializing the *avail* lists for the binary buddy-system method

Example 11.1. Binary-buddy system reservation and liberation

Assume that we have a memory pool of size 2^8 at relative addresses 0 through 255, and that initially all of it is free. This example shows the lists of available blocks and corresponding map of the memory pool after a series of reservations and liberations.

(a) Initially



(b) Reserve B1, size = 25 words

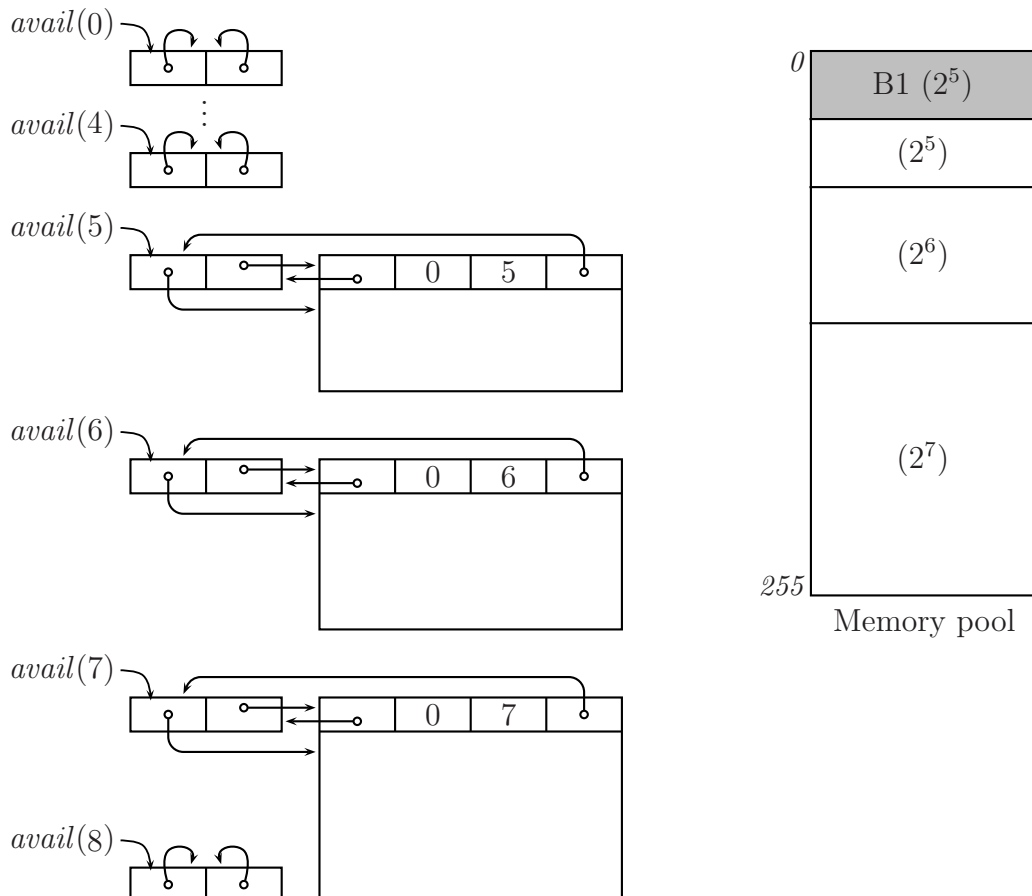
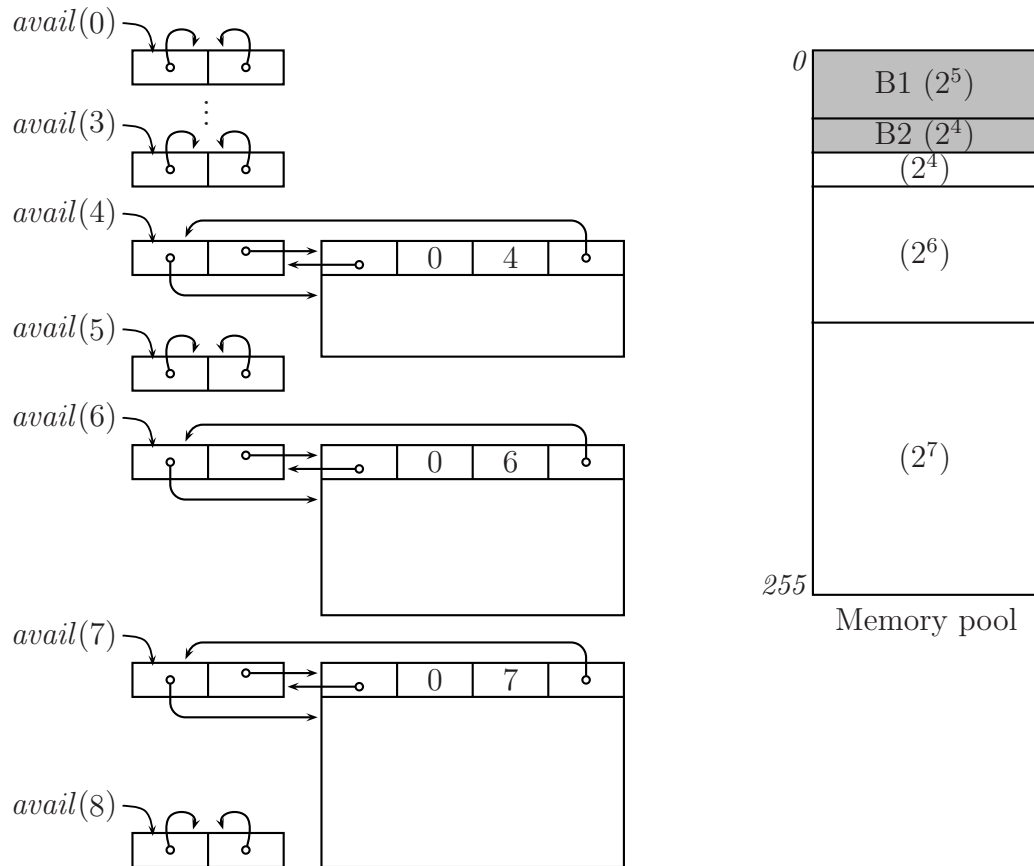


Figure 11.24 Binary buddy-system for DSM in action (continued on next page)

(c) Reserve B2, size = 12 words



(d) Reserve B3, size = 50 words

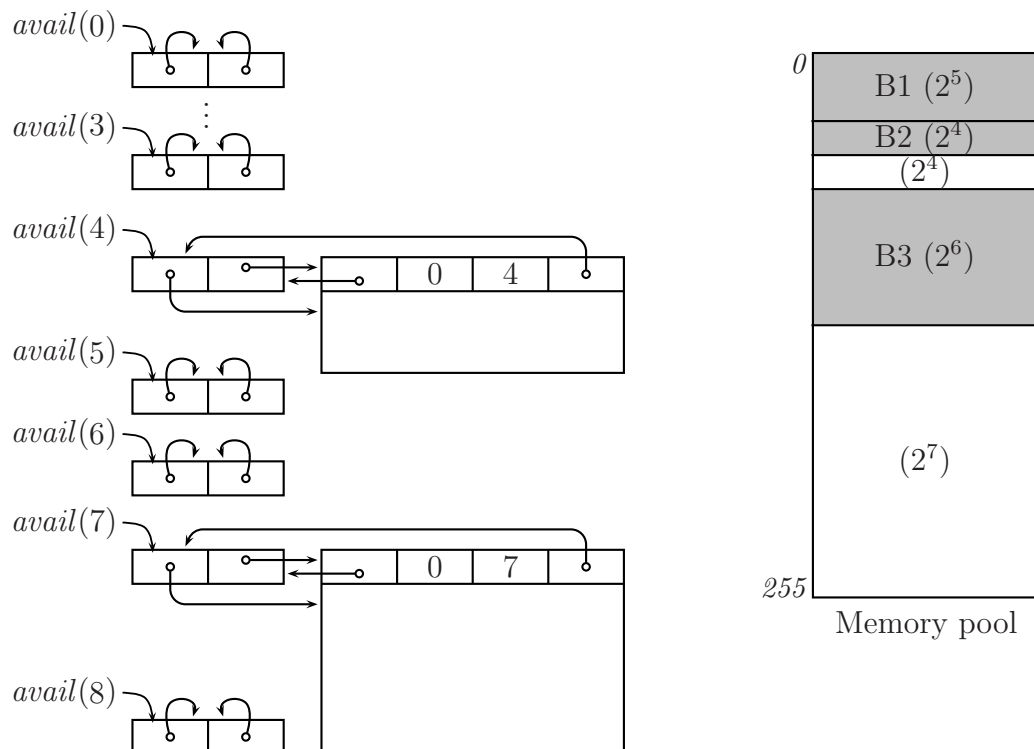
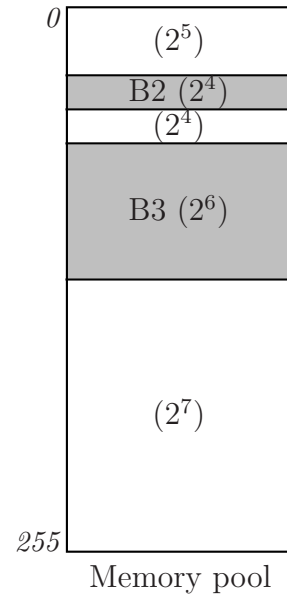
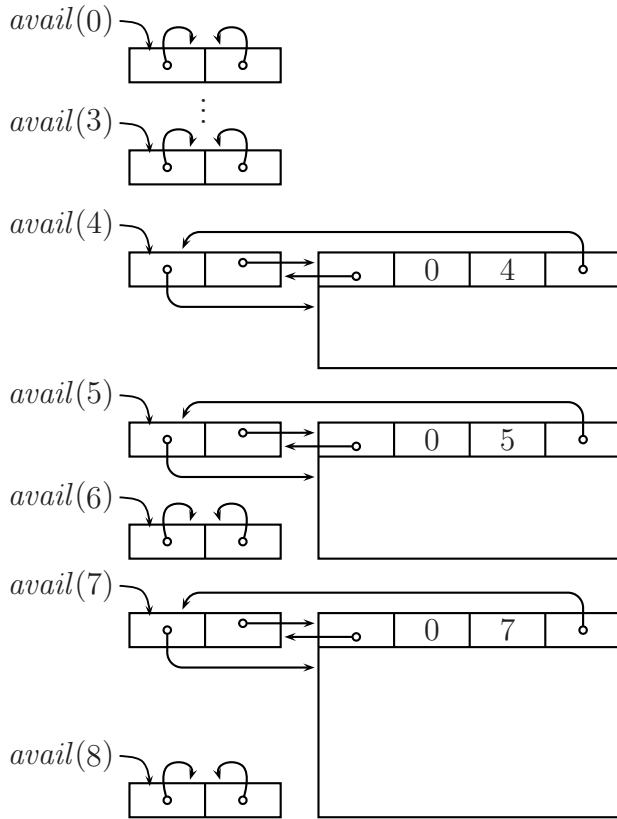


Figure 11.24 Binary buddy-system for DSM in action (continued on next page)

(e) Release B1



(f) Reserve B4, size = 40 words

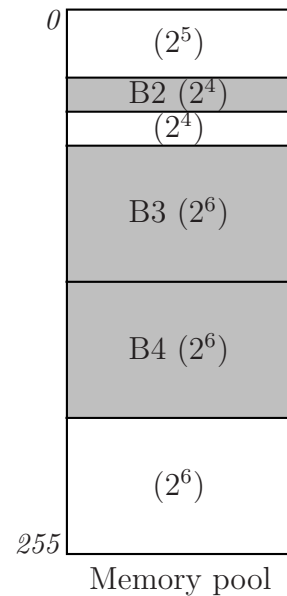
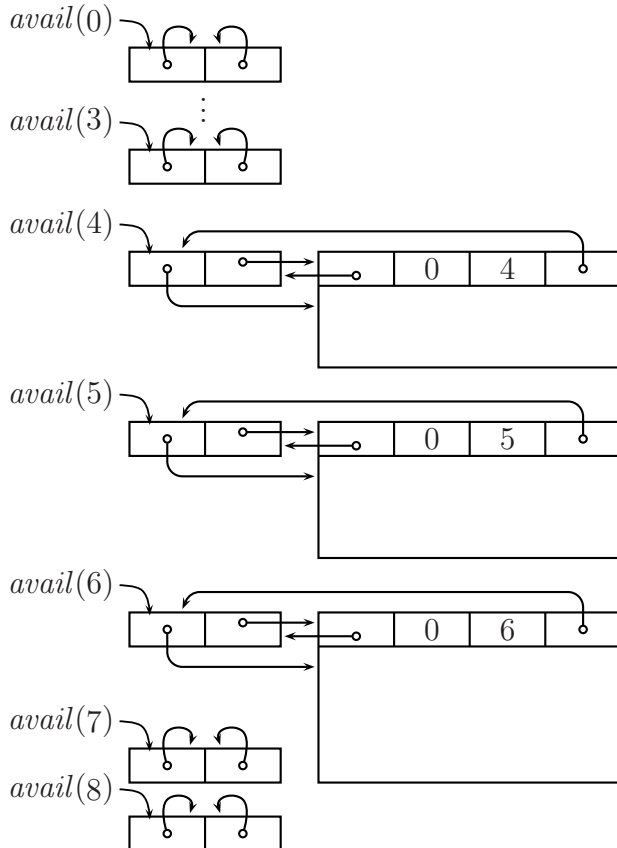
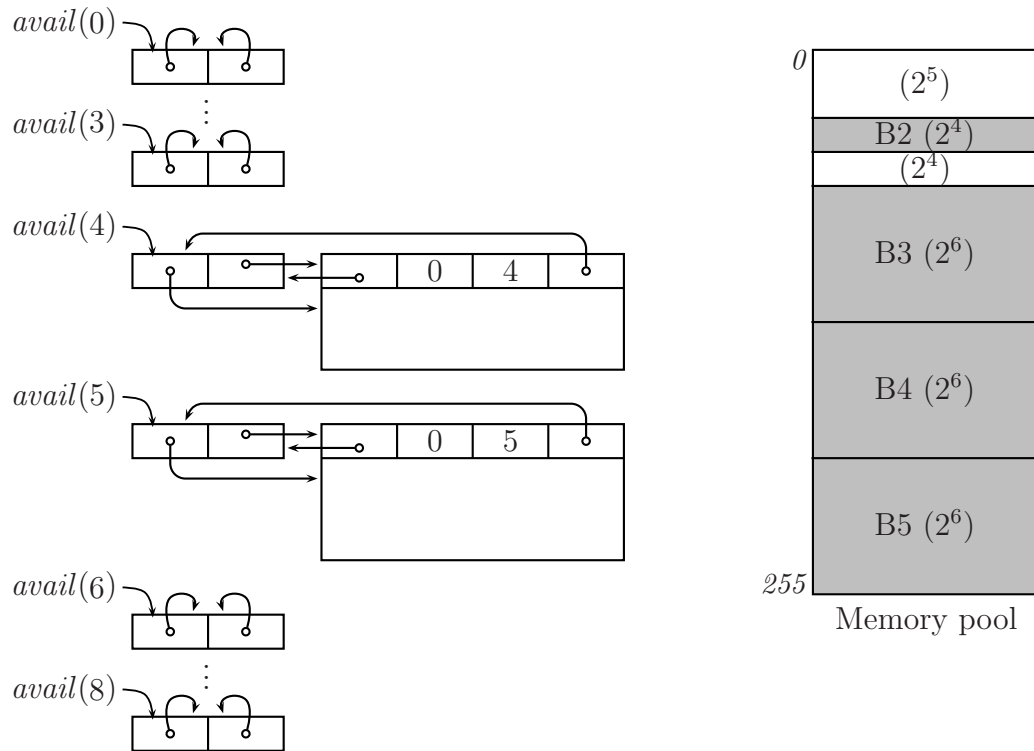


Figure 11.24 Binary buddy-system for DSM in action (continued on next page)

(g) Reserve B5, size = 55 words



(h) Release B3

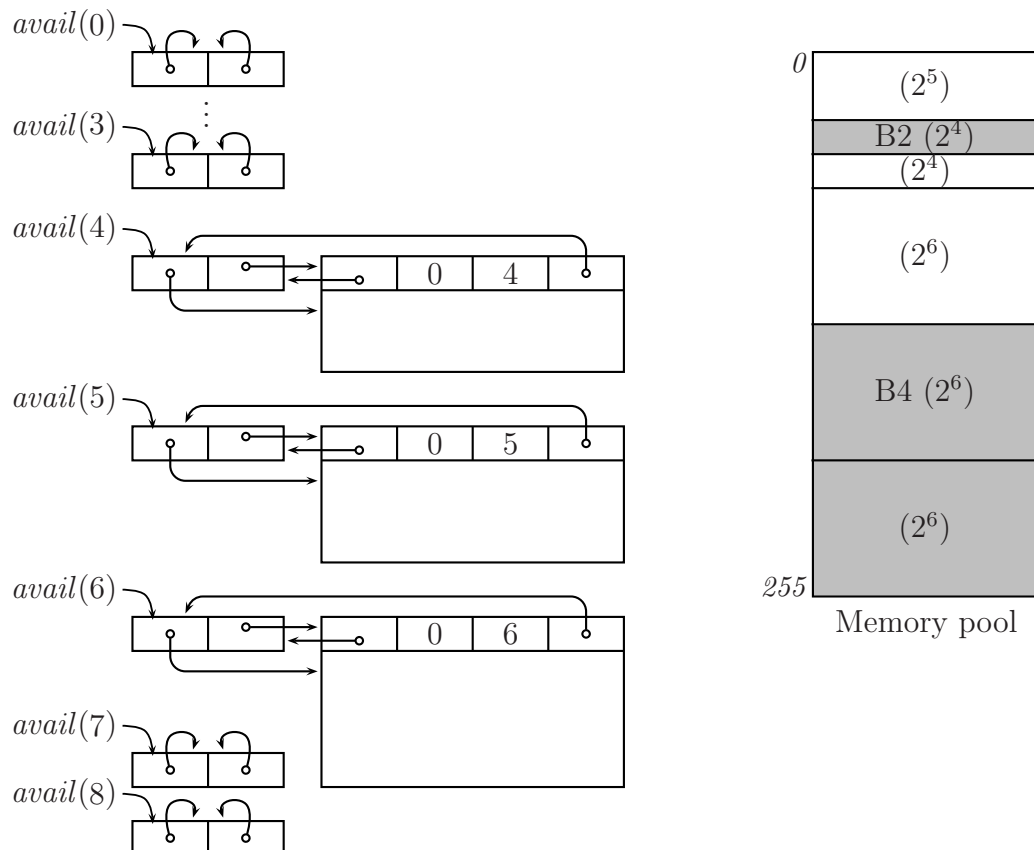


Figure 11.24 Binary buddy-system for DSM in action (continued on next page)

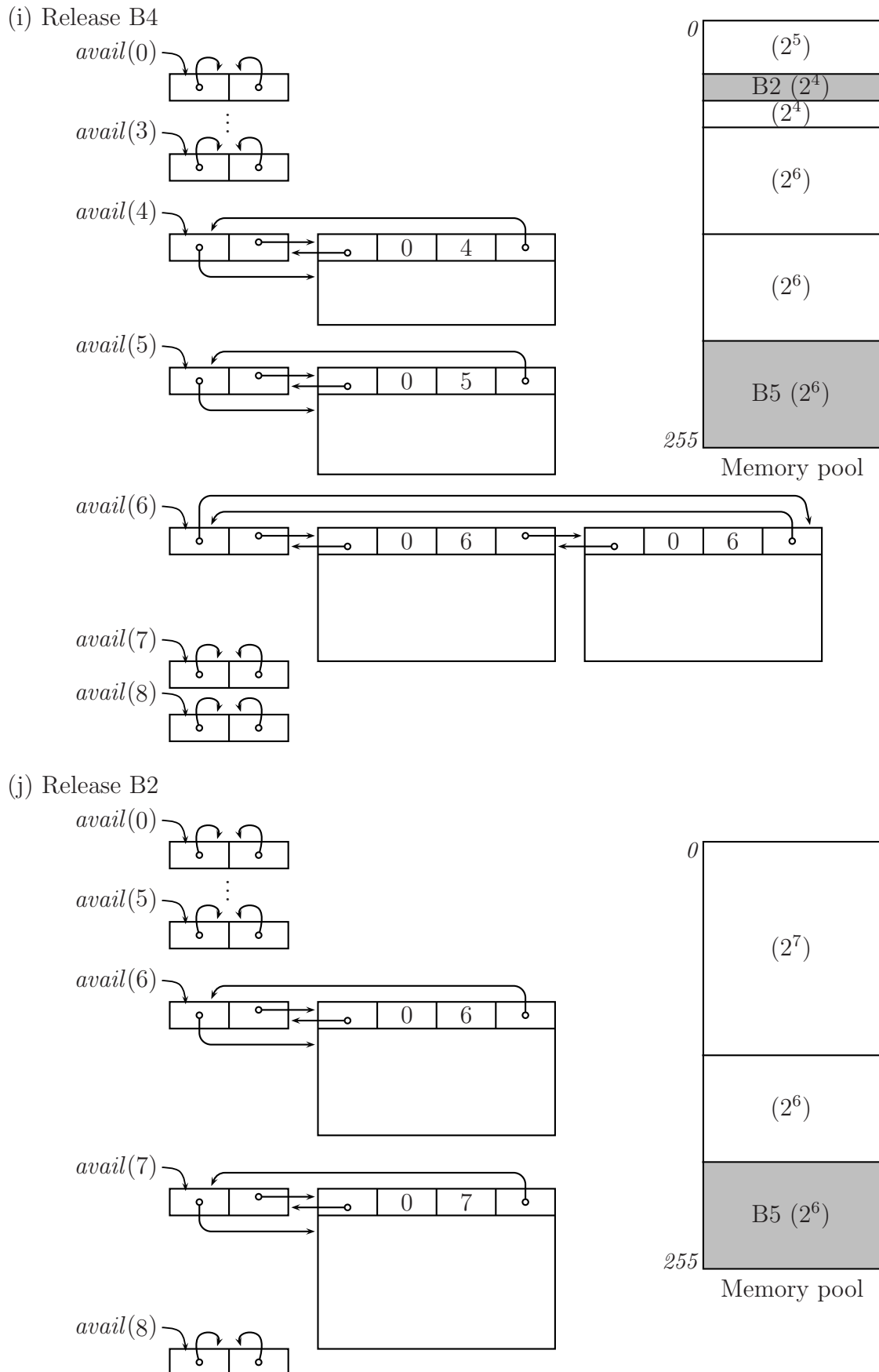


Figure 11.24 Binary buddy-system for DSM in action

The preceding example, though admittedly a small one, captures the essential features of the binary buddy-system method, to wit:

1. In step (b) the request for $n = 25$ words is rounded off to the nearest power of 2 which is $2^5 = 32$. The smallest available block which satisfies the request has size 2^8 , so the block is split 3 times fragmenting the memory pool into blocks of size 2^7 , 2^6 and 2^5 .
2. The request is for 25 words but 32 are reserved, resulting in internal fragmentation. This observation applies to all the other requests in steps (c), (d), (f) and (g).
3. In step (e) the buddy of the block that is released is reserved in part, so no collapsing can be done.
4. In step (i) two adjacent free blocks of size 2^6 are formed but they are *not* collapsed because they are not buddies.
5. In step (j) the freed block, B2, of size 2^4 is collapsed with its buddy yielding a block of size 2^5 , which is in turn collapsed with its buddy yielding a block of size 2^6 which is once more collapsed with its buddy yielding a block of size 2^7 , at which point no further collapsing can be done.

EASY procedures for the binary buddy-system method

```

▷ Reserves block of size  $2^k$ . Returns relative address of block found; else returns  $\Lambda$ .
1  procedure BUDDY_SYSTEM_RESERVATION( $k$ )
▷ Find block of size  $2^j$  where  $j$  is the smallest integer in the range  $k \leq j \leq m$  for
▷ which the  $avail(j)$  list is not empty
2    for  $j \leftarrow k$  to  $m$  do
3      if  $RLINK(avail(j)) \neq avail(j)$  then goto A
4    endfor
5    return ( $\Lambda$ )
▷ Remove block from its  $avail$  list
6  A:  $\alpha \leftarrow RLINK(avail(j))$ 
7     $RLINK(avail(j)) \leftarrow RLINK(\alpha)$ 
8     $LLINK(RLINK(\alpha)) \leftarrow avail(j)$ 
9     $TAG(\alpha) \leftarrow 1$ 
▷ Split block, if necessary
10 B: if  $j = k$  then return( $\alpha$ )
11      else [ $j \leftarrow j - 1$ 
12               $\beta \leftarrow \alpha + 2^j$       ▷ address of higher (addresswise) buddy
13               $TAG(\beta) \leftarrow 0$ 
14               $KVAL(\beta) \leftarrow j$ 
15               $LLINK(\beta) \leftarrow RLINK(\beta) \leftarrow avail(j)$ 
16               $RLINK(avail(j)) \leftarrow LLINK(avail(j)) \leftarrow \beta$ 
17              goto B]
18  end BUDDY_SYSTEM_RESERVATION

```

Procedure 11.21 Binary buddy-system reservation

Lines 2–4 searches for a non-null free list, starting with the $avail(k)$ list. If the $avail(k)$ list is not empty, then the first block on the list is removed (lines 6–9) and a pointer to it is returned to the requesting task (line 10). Otherwise, if the $avail(k)$ list is empty, then we try the next free list. The search may end in one of two ways: we find that rest of the free lists are empty in which case we return Λ to indicate that the reservation for a block of size 2^k cannot be satisfied (line 5), or we find a non-empty free list, say the $avail(j)$ list. In the latter case, we: (a) remove the first block on the $avail(j)$ list (lines 6–9), (b) split the block in half (line 11), (c) locate the higher (address-wise) buddy (line 12), and (d) insert this buddy into its own free list (lines 13–16). We repeat this process (line 17) until we obtain a block of size 2^k , at which point we return a pointer to it (line 10) and the reservation is done.

▷ Returns block of size 2^k with address α to the memory pool.

```

1  procedure BUDDY_SYSTEM_LIBERATION( $\alpha, k$ )
▷ Check if buddy is free.
2  A: if  $\alpha \bmod 2^{k+1} = 0$  then  $\beta \leftarrow \alpha + 2^k$ 
3      else  $\beta \leftarrow \alpha - 2^k$ 
4  if  $k = m$  or  $TAG(\beta) = 1$  or ( $TAG(\beta) = 0$  and  $KVAL(\beta) \neq k$ ) then goto B
▷ Combine with buddy
5   $RLINK(LLINK(\beta)) \leftarrow RLINK(\beta)$ 
6   $LLINK(RLINK(\beta)) \leftarrow LLINK(\beta)$ 
7   $k \leftarrow k + 1$ 
8  if  $\beta < \alpha$  then  $\alpha \leftarrow \beta$ 
9  goto A
▷ Insert block into  $avail(k)$  list.
10 B:  $RLINK(\alpha) \leftarrow RLINK(avail(k))$ 
11   $LLINK(\alpha) \leftarrow avail(k)$ 
12   $LLINK(RLINK(avail(k))) \leftarrow \alpha$ 
13   $RLINK(avail(k)) \leftarrow \alpha$ 
▷ Update control word.
14   $TAG(\alpha) \leftarrow 0$ 
15   $KVAL(\alpha) \leftarrow k$ 
16  end BUDDY_SYSTEM_LIBERATION

```

Procedure 11.22 Binary buddy-system liberation

Block α cannot be collapsed with block β , its buddy, if block α is the entire memory pool or if the buddy is reserved in part or in full (lines 2–4), in which case block α (of size 2^k) is inserted into the $avail(k)$ list (lines 10–15). Otherwise, if block β is free then it is deleted from the $avail(k)$ list and collapsed with block α . Note that the deletion is done in $O(1)$ time (lines 5–6); it is to this end that we maintain the free lists as doubly-linked lists. Merging the two buddies is accomplished by simply incrementing k by 1 (line 7). Finally, in line 8 we set the pointer α to point to the first word of the merged blocks, and then we check if further collapsing can be done (line 9).

As expected, the binary buddy-system method exhibits a very high rate of internal fragmentation; more than 40% of the reserved space may actually be unused. Other

buddy-system methods which reduce internal fragmentation have been proposed and these include the Fibonacci buddy-system by Hirschberg, the weighted buddy-system by Shen and Peterson, and the generalized Fibonacci buddy-system by Hinds.

While the binary buddy-system method is quite inefficient in terms of memory utilization, it turns out to be very efficient in terms of the speed with which it performs reservations and liberations. Simulations by Knuth indicated that it was seldom necessary to split blocks during reservation or to merge blocks during liberation, which explains why the method turns out to be so fast.

For a comparative study of the various algorithms for DSM presented in this section, see KNUTH1[1997], pp. 444–452 and STANDISH[1980], pp. 267–274.

11.7 Linear lists and UPCAT processing

Every year, the University of the Philippines System admits some 12,000 new freshmen into its nine campuses from some 70,000 applicants. The principal measure used by the University to determine who are admitted, in addition to the weighted average of the applicants' grades in their first three years in high school, is their performance in the U.P. College Admissions Test, or UPCAT.

Each applicant is given two choices of campus and two choices of degree program, or course, per campus. An applicant qualifies into a campus, and into a course within a campus, on the basis of so-called *predicted grades*, which are a composite of the applicant's high school grades and UPCAT scores. Four such predictors are computed: UPG (university predicted grade), MPG (mathematics predicted grade), BSPG (biological sciences predicted grade) and PSPG (physical sciences predicted grade).

UPCAT processing may be viewed as consisting of three successive phases, namely: (a) *data preparation*, (b) *selection*, and (c) *report generation*. Data preparation and report generation are primarily EDP activities, the former generally taking months to finish, and the latter a week or so. Data preparation involves: (a) validating/encoding applicants' personal data (b) validating/encoding applicants' high school data (c) validating/scanning/scoring UPCAT answer sheets, and (d) computing predictors: UPG, MPG, BSPG, PSPG. Report generation, on the other hand, involves mainly: (a) preparing lists of qualifiers, as determined during the selection phase, for posting on bulletin boards and on the Net, and (b) printing admission notices to be sent to all UPCAT applicants.

The middle phase, *selection*, is our present concern. Selection consists essentially of two tasks, namely:

1. For the entire UP System, *determining campus qualifiers* from among all UPCAT applicants, and
2. For each campus in the UP System, assigning campus qualifiers to the different degree programs within that campus, or equivalently, *determining degree program qualifiers*.

As implemented on the computer, both tasks utilize the linked list as the underlying data structure. For purposes of the present discussion, it suffices to consider the second task only.

11.7.1 Determining degree program qualifiers within a campus

The algorithm to assign campus qualifiers to the different degree programs within a campus needs to take into account the following points:

1. Some degree programs have more applicants than available slots (program quota); others have more available slots than applicants.
2. Degree program qualifiers are chosen by straight *pataasan* on the basis of their grades in the predictor for the course, e.g., UPG for AB Political Science, MPG for BS Computer Science, PSPG for BS Architecture, BSPG for BS Biology, and so on.
3. Each campus qualifier has two choices of degree program in that campus. An applicant who qualifies in his first choice of degree program is assigned to that program and is not considered for his second choice. An applicant who does not make it to his first choice of program is processed for his second choice. An applicant who does not make it to both choices is assigned to the campus *sink*.

The following example illustrates what transpires during the assignment process. Assume that the applicants are processed in non-descending order of their UPG and that, at this stage of the processing, degree programs P and Q, with quotas p and q , respectively, have already p and q qualifiers assigned to them, comprising the P and Q qualifiers' lists. The P-list is sorted according to the predictor for course P, say MPG; the Q-list is sorted according to the predictor for course Q, say PSPG. Now comes applicant X whose first choice of degree program is P. Even though his UPG is lower than that of the last qualifier, say S , in the P-list, X 's MPG may be higher than S 's MPG. Assuming that this is the case, X is inserted into his proper position in the P-list such that the list remains sorted. The P-list is now $p + 1$ long, or one in excess of the quota for course P. Accordingly, S is deleted from the P-list (unless he is 'tied' with the last qualifier in the P-list, in which case S remains in the list, albeit precariously). S is now considered for his second choice of degree program, say Q. If S 's PSPG (the predictor for course Q) is higher than that of the last qualifier in the Q-list, say T , then S is inserted into the Q-list; otherwise, S is placed in the campus sink, which is simply a list of all those who do not make it to both choices of degree program. In the former case, the Q-list becomes $(q + 1)$ long, so T is deleted from the list (unless, he is tied with the last qualifier in the Q-list). Now, it is T 's turn to be processed for his second choice of degree program. This insertion-deletion-reinsertion sequence triggered by the assignment of applicant X to the P-list may occur a number of times and then finally subside; at this point, the next applicant, say Y , is processed. And so on, until all qualifiers within the campus are assigned to their chosen courses, or to the campus sink.

It is clear from this example that it is necessary to solve the assignment problem *simultaneously* for all degree programs within a campus, and that assignment to a program becomes final only after all the qualifiers in that campus have been processed.

Tie processing

An important consideration during the assignment process is the handling of *ties* at the tail of a program qualifiers' list. Consider a program, say P, whose quota is p and

whose predictor is, say, MPG. Assume that the P-list, which is sorted by MPG, is already p long and that the last $m = 4$ qualifiers (say W, X, Y and Z) in the list have the same MPG, i.e., they are tied. Now, if a new qualifier with a higher MPG than Z is inserted into the list, the P-list becomes $p + 1$ long, or one in excess of the quota. However, since Y is still in the list, and Z has the same MPG as Y , Z is not dropped from the list (it is largely by accident that Z is the last among the four; it could have been W). If still another qualifier with a higher MPG than Z is inserted into the list, then both Y and Z are ‘shifted out’, with two qualifiers now in excess of the quota, but they are not deleted from the list because they have the same MPG as X , who is now the p th qualifier in the list. Only if four insertions are made into the P-list, so that all four of them are shifted out will W, X, Y and Z be dropped from the P-list. In general, as long as the excess over the quota is less than the number of ties, the excess qualifiers will be retained in the list (*bagama’t nanganganib na matanggal sa anumang sandali*).

Qualifiers to a degree program who are subsequently displaced by higher ranking qualifiers to the same program are considered for their second choice of program and may, of course, qualify there. Those who do not make it in both choices of degree programs are assigned to the campus sink; they can later choose a degree program with available slots.

11.7.2 Implementation issues

An efficient computer-based implementation of the assignment procedure described above must address the following issues:

1. We will need to maintain lists of program qualifiers, one list for each degree program within a campus, and a list for all those who do not qualify in both choices of courses. Each such list must be sorted according to the grades of the qualifiers in the list in the predictor for the course. We should be able to traverse the list in the forward direction during insertion, and in the backward direction during tie processing and possibly, deletion. We need to be able to do these operations relatively fast, considering that we need to process thousands of qualifiers for the nine campuses of the U.P. System.
2. We will need to maintain a table of all the degree programs for a given campus. For each course in the table, we need to keep the following information: the course code, the course title, the quota and predictor for the course, the number of qualifiers for the course, and a pointer to the list of qualifiers for the course. We should be able to find any course in the table, so that we gain access to the rest of the information pertaining to the course, as fast as possible.

Representing degree program qualifiers

To meet the requirements enumerated in 1, we will use a doubly-linked list with a list head to represent a degree program qualifiers' list, or PQ-list. Each node in the list represents a qualifier, and will have the node structure:

[illegible]

Figure 11.25 shows a typical degree program qualifiers' list. The program in this instance is BSCS (course code 4132) for which the predictor is MPG.

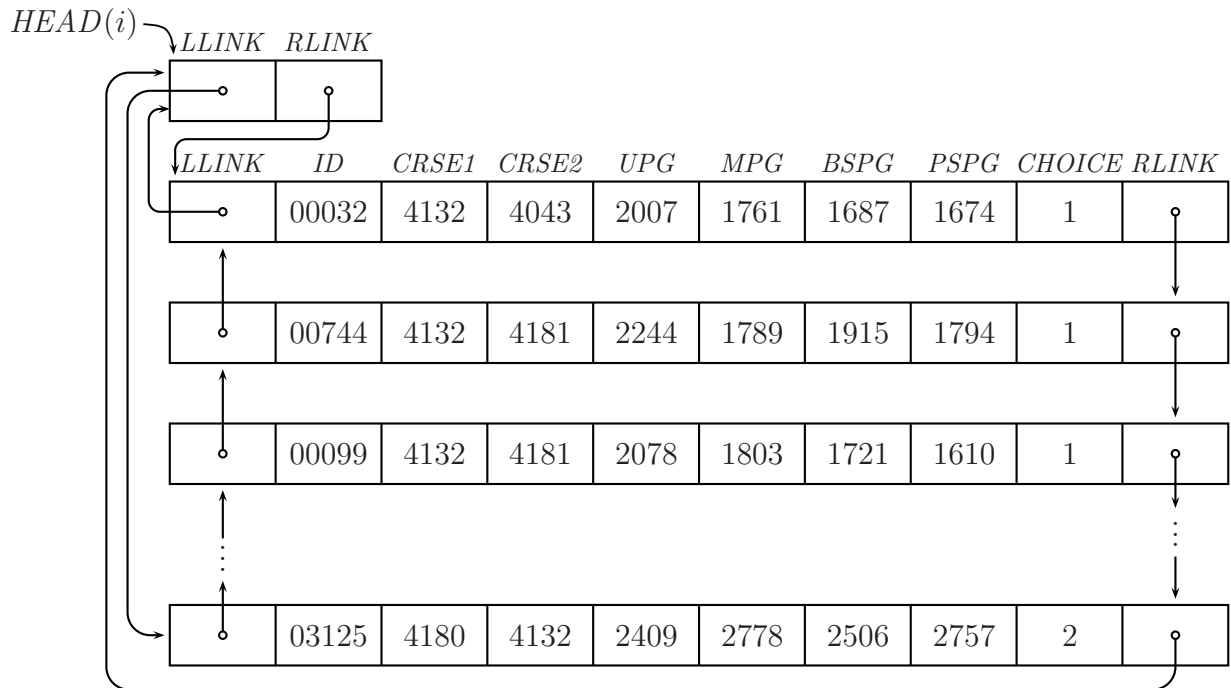


Figure 11.25 A typical degree program qualifiers' list

We take note of the following observations pertaining to the PQ-list shown in Figure 11.25 and to PQ-lists in general.

- There is one PQ-list for each degree program, or course, in a campus. Each node in a list represents a program qualifier, identified by a 5-digit ID . The list is sorted according to the grades of the qualifiers in the predictor for the course.
- The particular PQ-list shown in Figure 11.25 is for BS Computer Science in UP Diliman (course code 4132) whose predictor is MPG. Thus the list is sorted according to the entries in the MPG field from the numerically smallest (highest MPG) to the numerically largest (lowest MPG).
- The first choice of degree program of the first three qualifiers in the list is BSCS ($CRSE1 = 4132$), hence the $CHOICE$ field is set to 1. The first choice of the last qualifier in the list is BS Computer Engineering (4180) but didn't make it there; thus he/she is assigned to his/her second choice ($CRSE2 = 4132$) and the $CHOICE$ field is set to 2.
- A qualifier in an *over-subscribed* degree program, i.e., a program with more applicants than available slots such as BSCS, may be subsequently dis-qualified (deleted from the list), under conditions previously explained.
- Applicants who do not qualify in both choices of degree program are assigned to the campus sink which is simply a list similar to a PQ-list except that it does not pertain to any particular degree program.

- (f) A particular list is located via a pointer to its list head; this pointer is maintained in an array of pointers, *HEAD*, as discussed below.

Nodes which are deleted from PQ-lists are constituted into a linked queue, with linking done via the *RLINK* field. Nodes from the queue are subsequently re-inserted into other PQ-lists or into the campus sink.

Figure 11.26 shows a queue of nodes which have been deleted from PQ-lists such as the one shown above. Note that *LLINK* is now irrelevant and that the *RLINK* field of the rear node is set to Λ . The queue in this particular instance consists of two applicants tied at 2784 in the *MPG* field which were deleted from the BSCS list and a third applicant deleted from the BS Biology list (course code 4038). Upon re-assignment, the *CHOICE* field is set to 2 if finally assigned to their second choice of program, or to 0 if assigned to the campus sink.

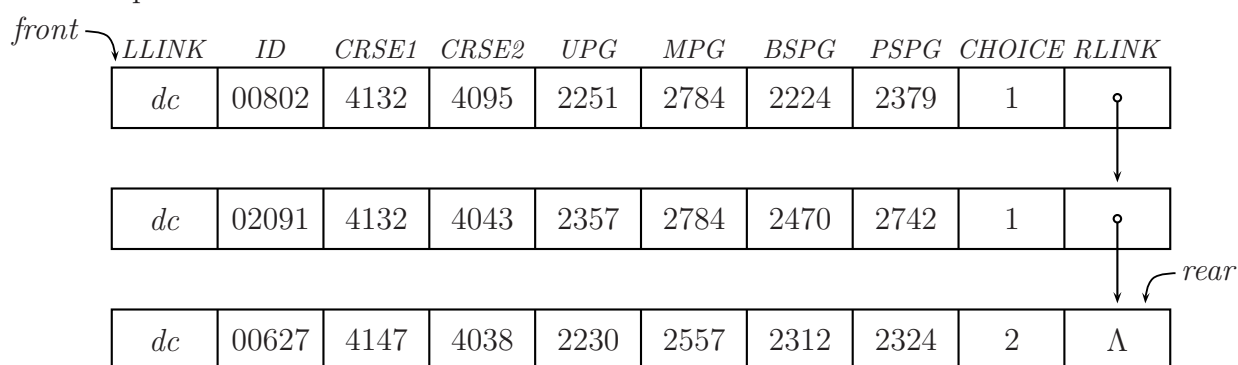


Figure 11.26 A queue of dis-qualified applicants awaiting re-assignment

Representing degree programs in a campus

To meet the requirements enumerated in 2, we use an array of *records*, where each record consists of the following data elements:

<i>course code</i>	a four digit code which identifies a campus and a course within the campus; this serves as the <i>key</i> which uniquely identifies the record
<i>course title</i>	name of the degree program
<i>course quota</i>	the number of available slots for the degree program
<i>course predictor</i>	1 = UPG, 2 = MPG, 3 = BSPG, 4 = PSPG
<i>count</i>	actual number of qualifiers for the course
<i>head</i>	pointer to the list head of the list of qualifiers for the course (see Figure 11.25)

In Session 13, we refer to this structure as a *sequential table*. We assume that the records in the table are sorted according to course code. This allows us to locate a course, given its code, using *binary search* and thereby gain access to the rest of the data pertaining to the course. Figure 11.27 shows the tables for UP Cebu and UP Iloilo for UPCAT 2006; the entries in the *QUOTA* column may change from year to year.

CODE	COURSE TITLE	QUOTA	PRED	COUNT	HEAD
2002	BA Political Science	55	1	<i>xxx</i>	<i>yyy</i>
2023	BA Psychology	64	3	<i>xxx</i>	<i>yyy</i>
2038	BS Biology	64	3	<i>xxx</i>	<i>yyy</i>
2099	BS Management	127	1	<i>xxx</i>	<i>yyy</i>
2101	BS Mathematics	55	2	<i>xxx</i>	<i>yyy</i>
2132	BS Computer Science	109	2	<i>xxx</i>	<i>yyy</i>
2153	B Fine Arts (Painting)	55	1	<i>xxx</i>	<i>yyy</i>
2163	BA Mass Communication	55	1	<i>xxx</i>	<i>yyy</i>
2777	Cebu Sink	—	—	<i>xxx</i>	<i>yyy</i>
⋮	⋮	⋮	⋮	⋮	⋮
5012	BA History	64	1	<i>xxx</i>	<i>yyy</i>
5022	BA Political Science	64	1	<i>xxx</i>	<i>yyy</i>
5023	BA Psychology	64	3	<i>xxx</i>	<i>yyy</i>
5025	BA Sociology	64	1	<i>xxx</i>	<i>yyy</i>
5034	BS Applied Mathematics	64	2	<i>xxx</i>	<i>yyy</i>
5038	BS Biology	109	3	<i>xxx</i>	<i>yyy</i>
5042	BS BA (Marketing)	127	2	<i>xxx</i>	<i>yyy</i>
5046	BS Chemistry	64	4	<i>xxx</i>	<i>yyy</i>
5087	BS Fisheries	182	1	<i>xxx</i>	<i>yyy</i>
5088	BS Food Technology	64	1	<i>xxx</i>	<i>yyy</i>
5099	BS Management	127	1	<i>xxx</i>	<i>yyy</i>
5111	BS Public Health	64	1	<i>xxx</i>	<i>yyy</i>
5114	BS Statistics	64	2	<i>xxx</i>	<i>yyy</i>
5132	BS Computer Science	64	2	<i>xxx</i>	<i>yyy</i>
5141	BA Broadcast Communication	64	1	<i>xxx</i>	<i>yyy</i>
5145	BS Economics	64	2	<i>xxx</i>	<i>yyy</i>
5176	BS Accountancy	127	2	<i>xxx</i>	<i>yyy</i>
5182	BA Literature	64	1	<i>xxx</i>	<i>yyy</i>
5202	BA Community Development	64	1	<i>xxx</i>	<i>yyy</i>
5777	Iloilo Sink	—	—	<i>xxx</i>	<i>yyy</i>

Figure 11.27 The course tables for UP Cebu and UP Iloilo (UPCAT 2006)

For each course in the table, say the i th course, $COUNT(i)$ is initialized to zero; at end of processing, it contains the actual number of qualifiers for the course. Similarly, $HEAD(i)$ is initialized to Λ ; subsequently it will point to the list head of the qualifiers' list for the course. Each of the other campuses of the UP System has a similar table.

11.7.3 Sample EASY procedures

The following EASY procedures, and the names of those invoked but not listed below, should suffice to give us an idea of the kind of operations on lists performed in the processing of the UPCAT, both at the system and campus levels.

```

1  procedure ASSIGN_PROGRAM_QUALIFIERS(infile)
2  call InitQueue(Q)
3  while not EOF(infile) do
4    call GETNODE( $\tau$ )
5    input ID( $\tau$ ), CRSE1( $\tau$ ), CRSE2( $\tau$ ), UPG( $\tau$ ), MPG( $\tau$ ), BSPG( $\tau$ ), PSPG( $\tau$ )
6    CHOICE( $\tau$ )  $\leftarrow$  1
7    course  $\leftarrow$  CRSE1( $\tau$ )
8  1:  $i \leftarrow$  BINARY_SEARCH(course)
9     $\alpha \leftarrow$  HEAD( $i$ )  $\triangleright$  list head of PQ-list for course  $i$ 
10    $\beta \leftarrow$  LLINK( $\alpha$ )  $\triangleright$  last node in PQ-list for course  $i$ 
11   if COUNT( $i$ )  $\geq$  QUOTA( $i$ ) and PG( $\tau$ ,  $i$ ) > PG( $\beta$ ,  $i$ ) then
12     [ case
13       : CHOICE( $\tau$ ) = 1: [ CHOICE( $\tau$ ) = 2; course  $\leftarrow$  CRSE2( $\tau$ ); goto 1 ]
14       : CHOICE( $\tau$ ) = 2: [ CHOICE( $\tau$ ) = 0; call INSERT_PQLIST( $\tau$ , sink); cycle ]
15     endcase ]
16   call INSERT_PQLIST( $\tau$ ,  $i$ )
17   call CHECK_LENGTH_PQLIST( $i$ , Q)
18   if not IsEmptyQueue(Q) then call REINSERT_PQLIST(Q)
19 endwhile
20 call OUTPUT_PQLISTS
21 end ASSIGN_PROGRAM_QUALIFIERS

1  procedure INSERT_PQLIST( $\tau$ ,  $i$ )
2   $\alpha \leftarrow$  HEAD( $i$ )
3   $\beta \leftarrow$  LLINK( $\alpha$ )
4  if COUNT( $i$ ) = 0 or PG( $\tau$ ,  $i$ )  $\geq$  PG( $\beta$ ,  $i$ ) then [ RLINK( $\beta$ )  $\leftarrow$   $\tau$ 
5                                         LLINK( $\tau$ )  $\leftarrow$   $\beta$ 
6                                         LLINK( $\alpha$ )  $\leftarrow$   $\tau$ 
7                                         RLINK( $\tau$ )  $\leftarrow$   $\alpha$  ]
8  else [  $\beta \leftarrow$  RLINK( $\alpha$ )
9        while PG( $\tau$ ,  $i$ )  $\geq$  PG( $\beta$ ,  $i$ ) do
10          $\beta \leftarrow$  RLINK( $\beta$ )
11       endwhile
12       RLINK(LLINK( $\beta$ ))  $\leftarrow$   $\tau$ 
13       LLINK( $\tau$ )  $\leftarrow$  RLINK( $\beta$ )
14       RLINK( $\tau$ )  $\leftarrow$   $\beta$ 
15       LLINK( $\alpha$ )  $\leftarrow$   $\tau$  ]
16  COUNT( $i$ )  $\leftarrow$  COUNT( $i$ ) + 1
17 end INSERT_PQLIST

```

Procedure 11.23 Selecting UPCAT degree program qualifiers

Summary

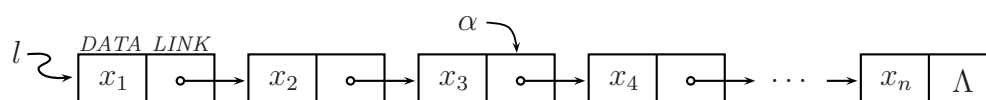
- Linear lists, implemented using the linked design, provide an efficient means of managing computer memory in applications where the amount of storage needed is not known *a priori* or changes unpredictably during program execution.
- There are several variations on the representation of linear lists as a collection of linked nodes — singly linked or doubly linked, straight or circular, with or without a list head. Which particular implementation is used depends primarily on the type and mix of operations performed on the list in a given application.
- The four applications of linear lists discussed at length in this session attest to the versatility of the linear list as an implementing data structure.
 - (a) polynomial arithmetic — polynomials are represented using a singly linked circular list with a list head
 - (b) multiple precision integer arithmetic — long integers are represented using a doubly linked circular list with a list head
 - (c) dynamic storage management — the free list(s) is(are) represented using a straight singly linked linear list with a list head or a doubly linked circular list with a list head
 - (d) UPCAT processing — degree program qualifiers' lists are represented using a doubly linked circular list with a list head
- All four applications of linear lists cited above utilize a list with a list head for one or more of the following reasons:
 - (a) a list head provides a well-defined starting and ending point as the list is traversed in cyclic fashion
 - (b) a list head allows the handling of the null list no differently from the non-null list, i.e., no special tests are needed for the case where the list is null thus resulting in neater code
 - (c) information about the list itself, say its length, is stored in the list head

Exercises

1. Write an EASY procedure DELETE_RIGHT to delete the rightmost element in a circular singly-linked linear list as depicted in Figure 11.4. What is the time complexity of your procedure?
2. Consider the straight singly-linked list l shown below. Write an EASY procedure to insert node β :

yy	
------	--

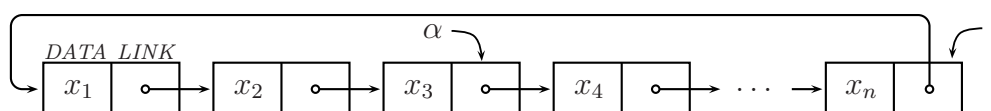
 immediately before node α in $O(1)$ time. Will your procedure work for *any* node α in l ?



3. Consider the straight singly-linked list in Item 2. Write an EASY procedure to delete a node given its address, e.g. node α , in $O(1)$ time. Will your procedure work for *any* node in the list?
4. Consider the circular singly-linked list shown below. Write an EASY procedure to insert node β :

yy	
------	--

 immediately before node α in $O(1)$ time. Will your procedure work for *any* node α in l ?



5. Consider the circular singly-linked list in Item 4. Write an EASY procedure to delete a node given its address, e.g. node α , in $O(1)$ time. Will your procedure work for *any* node in the list?
6. Let $l1$ and $l2$ be pointers to the list heads of two straight singly-linked lists (of the type depicted in Figure 11.3). Write an EASY procedure to test whether there is a sublist in $l1$ which is identical to the list $l2$.
7. Let $l1$ and $l2$ be pointers to the list heads of two doubly-linked circular lists (of the type depicted in Figure 11.6) which are to be concatenated. $l1$ will subsequently point to the concatenated lists and $l2$ to an empty list.
 - (a) Show *before* and *after* diagrams for all four possible cases.
 - (b) Write an EASY procedure to concatenate two doubly-linked lists as described.
8. Show the representation of the following polynomials according to the conventions adopted in section 11.4 and depicted in Figure 11.7.
 - (a) $x^5 - 6y^5 - 2x^4y + 5xy^4 + 3x^3y^2 - 4x^2y^3$
 - (b) $xyz^4 + 10x^2y^3z - 18y^4z^2 - 10x^3 + 25$
9. Using a language of your choice, transcribe procedures POLYREAD, POLYADD, POLYSUB and POLYMULT (Procedures 11.9 thru 11.12) into a running program.
10. Let

$$P = x^5y^2z - 10x^4y^2z^2 + 7x^3y^2 + 6x^3yz - 18$$

$$Q = x^5y^3z^2 - 10x^4y^2z^2 - 6x^3yz + yz + 15$$

$$S = x^3y^2z + 3x^2y^3z - 3xy^2z^3$$

Using your program in Item 9 find:

- | | | |
|-------------|-----------------|-------------------|
| (a) $P + P$ | (d) $P + Q$ | (g) $P * Q * S$ |
| (b) $P - P$ | (e) $P - Q$ | (h) $(P + Q) * S$ |
| (c) $P * P$ | (f) $P * Q + S$ | (i) $(Q - Q) * S$ |

11. Show the representation of the following integers according to the conventions adopted in section 11.5.5 and depicted in Figure 11.9.
 - (a) 132404746317716746, $r = 10^4$
 - (b) 132404746317716746, $r = 10^9$
 - (c) -15666365641302312516 , $r = 10^4$
 - (d) -15666365641302312516 , $r = 10^9$
12. Using a language of your choice, transcribe procedures READ_LONGINTEGER, ADD_LONGINTEGERS and MULTIPLY_LONGINTEGERS (Procedures 11.13 thru 11.15) into a running program.
13. Let $a = 243290200817664$, $b = 1099511627776$, $c = 1125899906842624$ and $r = 10^4$. Using your program in Item 12 find: (a) $a + b$ (b) $b - a$ (c) $b * c$
14. Assume that the *avail* list consists of three blocks of size 3000, 1400 and 2200 in that order. Give an example of a sequence of reservations which shows first-fit ‘superior’ to best-fit in that the latter can no longer satisfy a request whereas the former still can.
15. Assume that the *avail* list consists of three blocks of size 3000, 1400 and 2200 in that order. Give an example of a sequence of reservations which shows best-fit ‘superior’ to first-fit in that the latter can no longer satisfy a request whereas the former still can.
16. Assume that you are using the binary buddy system for DSM on a memory pool of size 2^9 and that all of it is initially free. The following reservations and liberations are then performed.

(a) reserve B1 (20 words)	(d) release B1	(g) release B3
(b) reserve B2 (50 words)	(e) reserve B4 (100 words)	
(c) reserve B3 (60 words)	(f) release B2	

Show the resulting lists of available blocks, both empty and non-empty, upon termination of: (i) operation (f) (ii) operation (g)

Bibliographic Notes

One of the earliest applications of linked lists is polynomial arithmetic; the subject is discussed in KNUTH1[1997], TREMBLAY[1976], WEISS[1997], among others. Procedures POLYADD and POLYMULT are EASY implementations of Algorithm **A** (*Addition of polynomials*) and Algorithm **M** (*Multiplication of polynomials*) given in KNUTH1[1997], pp. 276–277. Multiple-precision integer arithmetic is discussed in KNUTH2[1998], TREMBLAY[1976], TENENBAUM[1986], among others. Procedures ADD_LONGINTEGERS and MULTIPLY_LONGINTEGERS are implementations of Algorithm **A** (*Addition of nonnegative integers*) and Algorithm **M** (*Multiplication of nonnegative integers*) given in KNUTH2[1998], pp. 266–270; the algorithms have been generalized to handle signed

integers represented using linked lists. Dynamic memory management is discussed in KNUTH1[1997], TENENBAUM[1986], STANDISH[1980], HOROWITZ[1976], among others. Procedures FIRST_FIT_1 and FIRST_FIT_2 are implementations of Algorithm **A** (*First-fit method*) given in KNUTH1[1997], p.437–438, 607–608 (initial and improved versions). Procedures SORTED_LIST_TECHNIQUE and BOUNDARY_TAG_TECHNIQUE are EASY implementations of Algorithm **B** (*Liberation with sorted list*) and Algorithm **C** (*Liberation with boundary tags*) given in the same reference, pp. 440–442. Procedures BUDDY_SYSTEM_RESERVATION and BUDDY_SYSTEM_LIBERATION are EASY implementations of Algorithm **R** (*Buddy system reservation*) and Algorithm **S** (*Buddy system liberation*), also from KNUTH1[1997], pp. 443–444. The same algorithms, or variations thereof, may be found in the other cited references.

SESSION 12

Generalized lists

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define a generalized list and explain related concepts.
2. Show the representation of a generalized list using different representation schemes and explain the assumptions underlying each.
3. Write recursive procedures to perform simple operations on generalized lists, such as making a copy of a list, finding the depth of a list, determining whether two lists are exactly the same, and so on.
4. Explain the concept of automatic storage reclamation in a list processing environment and how it is implemented using the garbage-collection technique and the reference-counter technique.
5. Discuss the relative advantages and disadvantages of the garbage-collection and reference-counter techniques.
6. Explain the Schorr-Waite-Deustch algorithm and Wegbreit's algorithm for marking cyclic list structures.

READINGS KNUTH1[1998], pp. 408–421; STANDISH[1980], pp. 186–223; TENENBAUM[1986], pp. 646–692; TREMBLAY[1976], pp. 392–402.

DISCUSSION

In the applications of lists that we have encountered thus far, all the elements of a list are *atoms*. Here we think of an atom as any entity that is not a list. For instance, an atom may be a term in a polynomial, a piece of a long integer, a free block of memory, an UPCAT degree program qualifier, and so on. There are various other applications of lists, however, in which the elements of a list may themselves be lists yielding what is called a **generalized list** or a **list structure**.

The manipulation of generalized lists is called **list processing**. The development of list processing systems (e.g., IPL-V by Newell and Tonge, SLIP by J. Weizenbaum, LISP

by J. McCarthy) in the early 60's was a defining point in what was then the evolving field of information structures and non-numerical programming. The contemporary use of list processing systems, in which LISP and its offspring Scheme, play a dominant role is primarily in the field of **artificial intelligence** which includes such problem areas as computational linguistics, robotics, pattern recognition, expert systems, generalized problem solving, theorem proving, game playing, algebraic manipulation, and others.

12.1 Definitions and related concepts

1. A generalized list, say $L = (x_1, x_2, x_3, \dots, x_n)$, is a finite, ordered sequence of zero or more atoms or lists. Implied in this definition is the assumption that we can distinguish between an element which is a list and one which is not a list, i.e., an atom. To this end, we will assume that we have available a predicate, **atom**(x), which returns *true* if x is an atom, and returns *false*, otherwise.

If the element x_i is a list, we say that x_i is a *sublist* of L .

2. A list with no elements, $L = ()$, is said to be null. We assume that we have available a predicate, **null**(L), which returns *true* if L is null, and returns *false*, otherwise.
3. For any non-null List $L = (x_1, x_2, x_3, \dots, x_n)$, we define the **head** and **tail** of L as follows: **head**(L) = x_1 and **tail**(L) = (x_2, x_3, \dots, x_n) . Simply stated, the head of a list is its first element and the tail of a list is the list which results after the first element is removed. Note that the head of a list is either an atom or a list and that the tail is always a list.

The following examples should help clarify the above concepts.

(a) $L = (a, (b, c))$

head(L) = a

tail(L) = $((b, c))$

head(**tail**(L)) = (b, c)

tail(**tail**(L)) = $()$

head(**head**(**tail**(L))) = b

tail(**head**(**tail**(L))) = (c)

(b) $L = (a, ())$

head(L) = a

tail(L) = $(())$

head(**tail**(L)) = $()$

tail(**tail**(L)) = $()$

atom(**head**(L)) = *true*

null(**tail**(L)) = *false*

4. A generalized list has both **length** and **depth**. The length of a list is the number of elements at *level* 1 of the list, where the level of an element is the number of pairs of parentheses which enclose the element. For example, the elements at level 1 of the list $L = ((a, b), ((c, d), e))$ are the sublists (a, b) and $((c, d), e)$; thus L has length 2.

The depth of a list is the level of the element with maximum level. Formally:

depth[*atom*] = 0

depth[$()$] = 0

depth[(x_1, x_2, \dots, x_n)] = $1 + \max(\text{depth}[x_1], \text{depth}[x_2], \dots, \text{depth}[x_n])$, $n \geq 1$

For instance, the depth of the list $L = ((a, b), ((c, d), e))$ is found as follows:

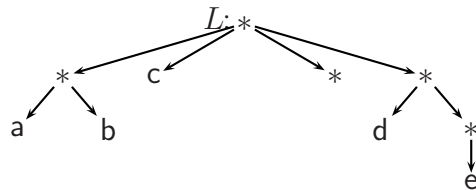
$$\begin{aligned}
 \text{depth}[L] &= 1 + \max(\text{depth}[(a,b)], \text{depth}[(c,d),e]) \\
 &= 1 + \text{depth}[(c,d),e] \\
 &= 1 + 1 + \max(\text{depth}[(c,d)], \text{depth}[e]) \\
 &= 1 + 1 + \text{depth}[(c,d)] \\
 &= 1 + 1 + 1 + \max(\text{depth}[c], \text{depth}[d]) \\
 &= 1 + 1 + 1 + \text{depth}[c] \\
 &= 1 + 1 + 1 + 0 = 3
 \end{aligned}$$

This is simply the number of pairs of parentheses which enclose the most deeply nested elements, viz., the atoms *c* and *d*.

5. Any generalized list can be represented by a graph. Such a representation may be used as a basis for classifying lists in their order of increasing complexity into **pure**, **reentrant** and **recursive** (or **cyclic**) lists.

- (a) pure list — a list whose graph corresponds to an ordered tree in which there is exactly one path between any two distinct nodes.

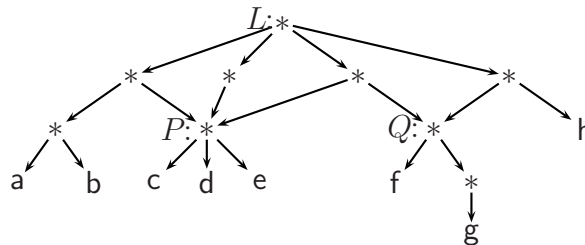
Example 12.1. $L = ((a,b),c,(d,e))$



Note: An asterisk indicates the instance of a list or sublist.

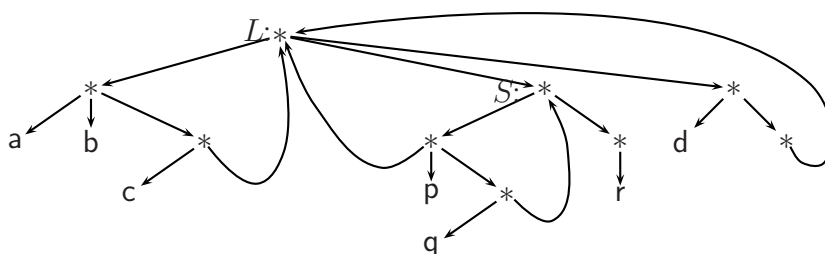
- (b) reentrant list — a list whose graph contains vertices accessible via two or more paths; such vertices represent *shared elements* of the list. All paths in the graph are of finite length.

Example 12.2. $L = (((a,b),P),(P),(P,Q),(Q,h))$ where $P = (c,d,e)$ and $Q = (f,g)$



- (c) cyclic or recursive list — a list whose graph contains cycles which correspond to instances in which the list or its sublists ‘contain themselves’.

Example 12.3. $L = ((a,b,(c,L),S,(d,(L))))$ where $S = ((L,p,(q,S)),(r))$



12.2 Representing lists in computer memory

Any list, whether pure, reentrant or recursive, can be represented in the memory of a computer using the node structure



The *DATA* field may contain **atomic data**, in which case *TAG* is set to 0, or it may contain **pointer data**, in which case *TAG* is set to 1. In the former case, the node is referred to as an **atomic node**; in the latter case, the node is referred to as a **list** (or **nonatomic**) **node**. The *DATA* field contains the head of a list or a sublist, if the head is an atom, or it contains a pointer to the head, if the head is a list. The *LINK* field always contains pointer data; it points to the tail of a list or a sublist. These conventions are depicted in Figure 12.1.



Figure 12.1 Nodes in a generalized list: (a) atomic node (b) list node

Example 12.4. $L = ((a,b),c,(()),((d,e),f))$

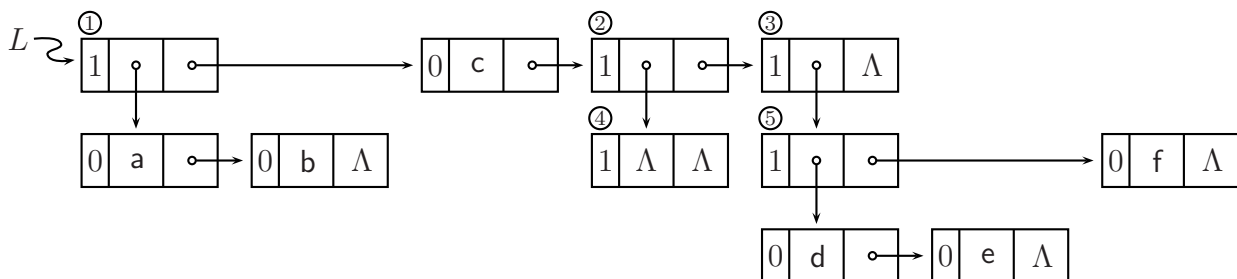


Figure 12.2 Representing a pure list using the node structure [*TAG*, *DATA*, *LINK*]

We take note of the following observations pertaining to Figure 12.2.

1. The list L is a pure list; there are no shared elements.
2. A list node indicates the instance of a sublist. The five list nodes, numbered 1 thru 5, indicate the instances of the sublists (a,b) , $(())$, $((d,e),f)$, $()$ and (d,e) , respectively.
3. A null sublist is represented by a list node with Λ in the *DATA* field.
4. The length of the list, 4, is the number of nodes in level 1; the depth of the list, 3, is the number of levels.

Example 12.5. $B = (((p, A, q), A), r, (A, (s)))$ where $A = ((a, b), ((c, d), e))$

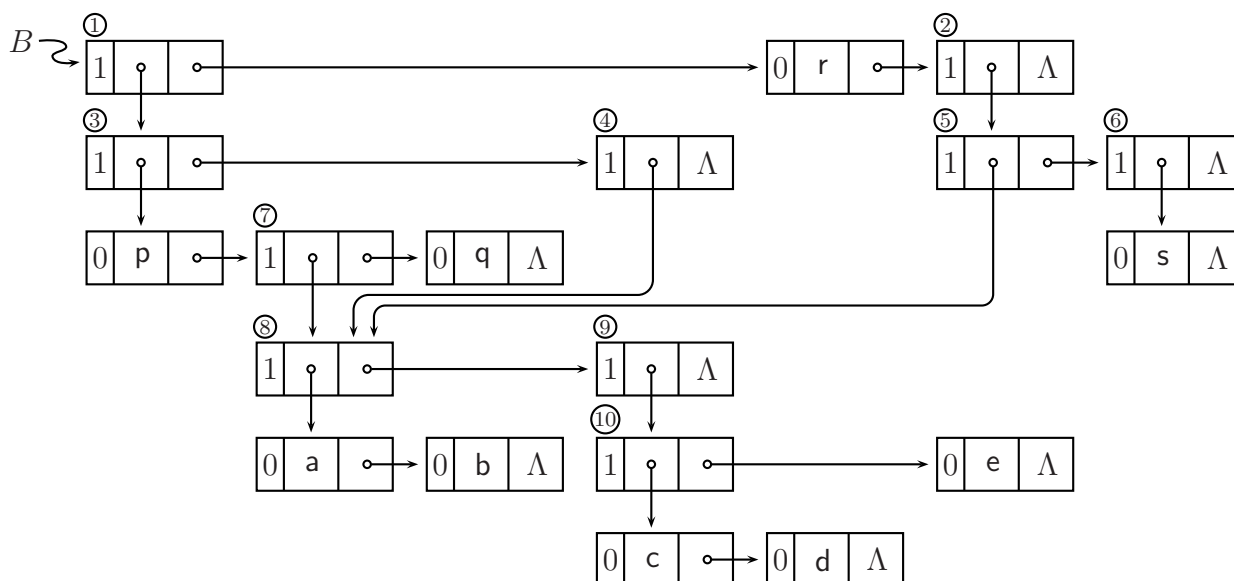


Figure 12.3 Representing a reentrant list using the node structure $[TAG, DATA, LINK]$

We take note of the following observations pertaining to Figure 12.3.

1. The list B is a reentrant list; the shared sublist is the list A . There is only one copy of the sublist A and all three references to A point to this copy.
2. A list node indicates the instance of a sublist. The list nodes numbered 1 thru 7 indicate the instances of the sublists $((p,A,q),A)$, $(A,(s))$, (p,A,q) , A , A , (s) and A , respectively, in B (which instance of A is referred to should be obvious). The list nodes numbered 8 thru 10 correspond to the sublists (a,b) , $((c,d),e)$ and (c,d) , respectively, in A .
3. The length of the list, 3, is the number of nodes in level 1; the depth of the list, 6, is the number of levels.

Representation using list heads

In certain applications of generalized lists a frequent operation is deleting the first element of a list or a sublist. Suppose we want to delete the first element of A in Example 12.5, viz., the sublist (a,b) . In Figure 12.3, this operation entails locating list nodes 4, 5 and 7 and setting the *DATA* field of each to point to list node 9; this is clearly *not* a neat and efficient way to perform the operation. To solve this problem, we simply use a *list head* to begin every list or sublist. Every reference to a list or sublist should then point to the list head. Figure 12.4 depicts the lists L and B of Figures 12.2 and 12.3 represented using list heads (shown as nodes with a shaded *TAG* field). Note that to delete the sublist (a,b) of the shared sublist A in list B , we only need to set the *LINK* field of the list head of sublist A to point to list node 9; we need not locate list nodes 4, 5 and 7 anymore.

A list head have other uses as well. For instance, information pertaining to a list, e.g., the number of references to the list, can be stored in the *DATA* field of the list head.

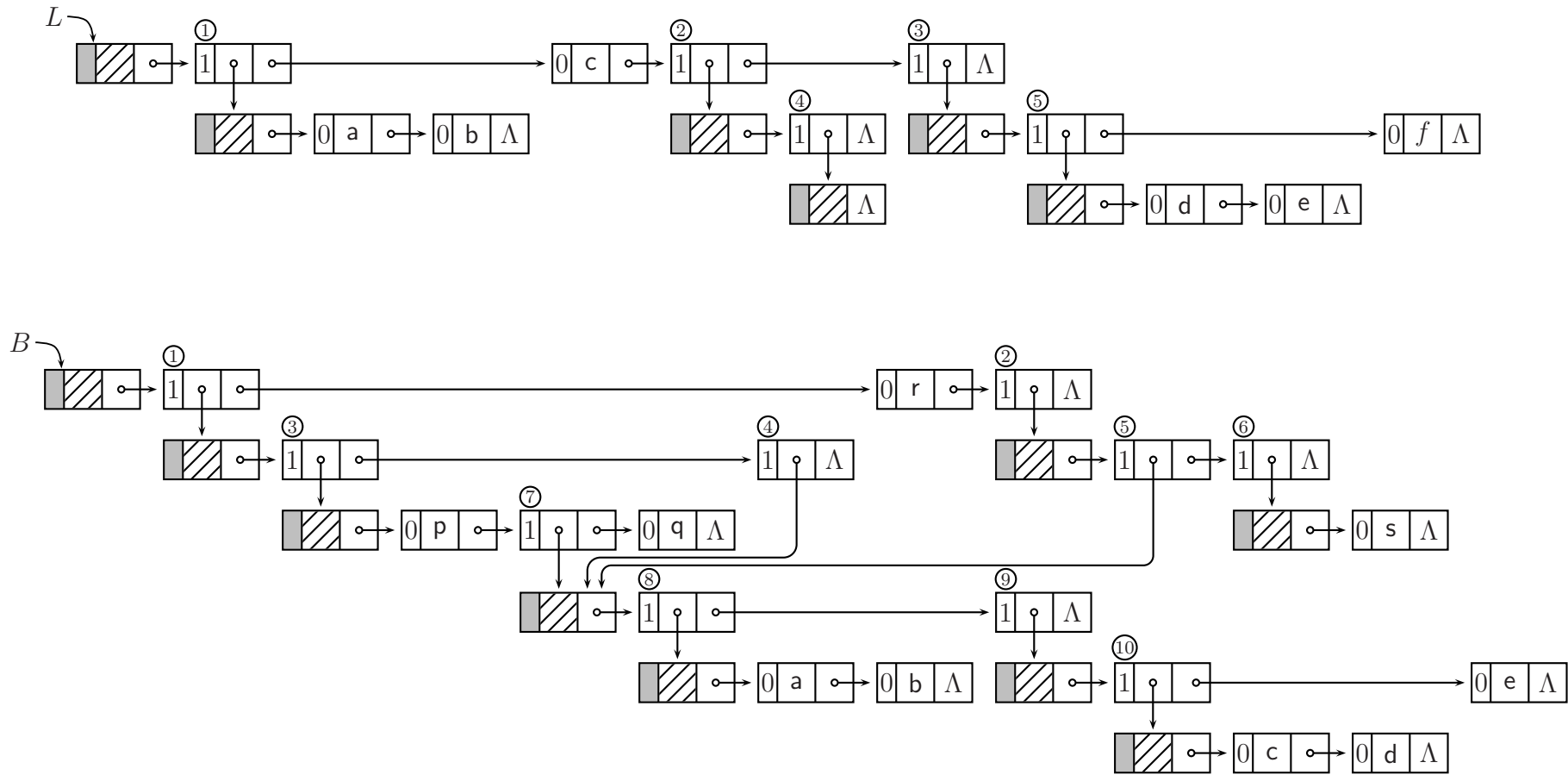


Figure 12.4 Alternative representations of the lists of Figures 12.2 and 12.3 using list heads

2. Case where a node holds

- (a) an atom only, or
- (b) two addresses only



This is a very probable scenario. A whole word is occupied by atomic data, e.g., a floating point number, or it is divided into two half-words where each contains addresses, most likely, relative addresses. In the absence of a tag bit within the node to distinguish between atomic and list nodes, other mechanisms must be devised.

- (a) Use **address range discrimination**. The idea is to place atomic nodes and list nodes in distinct regions of the list space, a 'list region' for list nodes and an 'info region' for atomic nodes. We may then determine whether a node is atomic or nonatomic by simply checking whether its address is in the 'info' or the 'list' region of the list space.

Example 12.7. $L = ((a,b),c,(()),((d,e),f))$

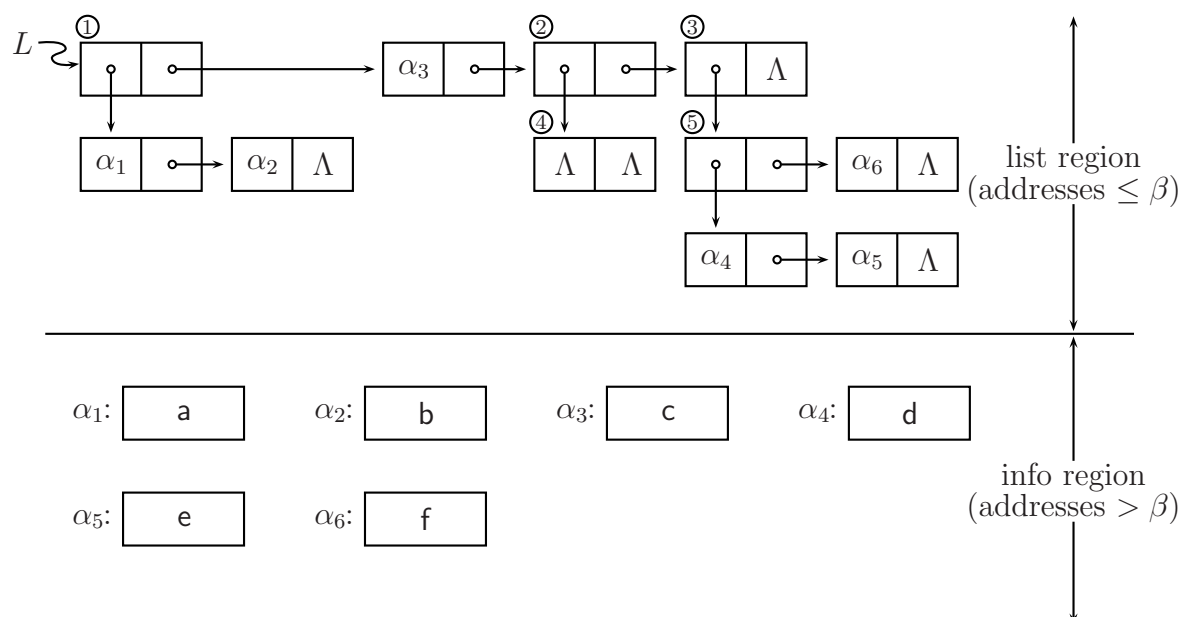


Figure 12.6 Representing a list using address range discrimination to distinguish between atomic nodes and list nodes

- (b) Use a **bit map**. The idea is to allocate a bit map with a 1 – 1 correspondence between nodes in the list space and bits in the bit map. For instance, if the i th node in list space is used as an atomic node, the i th bit in the bit map is set to 0; and if the j th node in the list space is used as a nonatomic node, then the j th bit in the bit map is set to 1. The relationship between the index i of the i th node and its address, say α , is easily established. If every node in list space consists of c addressable units and if the address of the first addressable unit in list space is σ , then we have $i = (\alpha - \sigma)/c + 1$.

12.3 Implementing some basic operations on generalized lists

Earlier we defined two entities called the head and tail of a non-null list, say $L = (x_1, x_2, x_3, \dots, x_n)$, to wit: the head of L is its first element, x_1 , which can be an atom or a list; and the tail of L , (x_2, x_3, \dots, x_n) , is the list that results after the head is removed. Given a list L , the procedures $\text{HEAD}(L)$ and $\text{TAIL}(L)$ return a pointer to the head and tail of L , respectively. The converse procedure, $\text{CONSTRUCT}(H, T)$, constructs and returns a pointer to the list whose head is H and whose tail is T .

In the implementation of the procedures HEAD , TAIL and CONSTRUCT given below, we assume that lists are represented using list heads, as depicted in Figure 12.4. For purposes of identifying a list head, we assume that its TAG field is set to -1 .

Finding the head of a non-null list

```

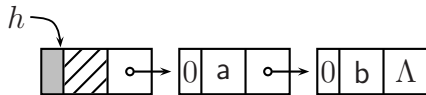
1  procedure HEAD( $L$ )
▷ Returns a pointer to the head of a non-null list  $L$ 
2  if  $L = \Lambda$  or  $\text{TAG}(L) \neq -1$  then [output 'Error: argument is not a valid list'; stop]
3   $\beta \leftarrow \text{LINK}(L)$ 
4  case
5    :  $\beta = \Lambda$ : [output 'Error: head is not defined for a null list'; stop]
6    :  $\beta \neq \Lambda$ : [if  $\text{TAG}(\beta) = 0$  then return( $\beta$ )
7                  else return( $\text{DATA}(\beta)$ )]
8  endcase
9  end HEAD

```

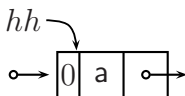
Procedure 12.1 Finding the head of a list

The procedure returns a pointer to an atomic node if the head is an atom (line 6) or a pointer to a list head if the head is a list (line 7). To illustrate, assume that L is the list shown in Figure 12.4; the following calls to HEAD will yield the indicated results.

(a) $h \leftarrow \text{HEAD}(L)$



(b) $hh \leftarrow \text{HEAD}(\text{HEAD}(L))$



(c) $hhh \leftarrow \text{HEAD}(\text{HEAD}(\text{HEAD}(L)))$

Error: argument is not a valid list.

Finding the tail of a non-null list

```

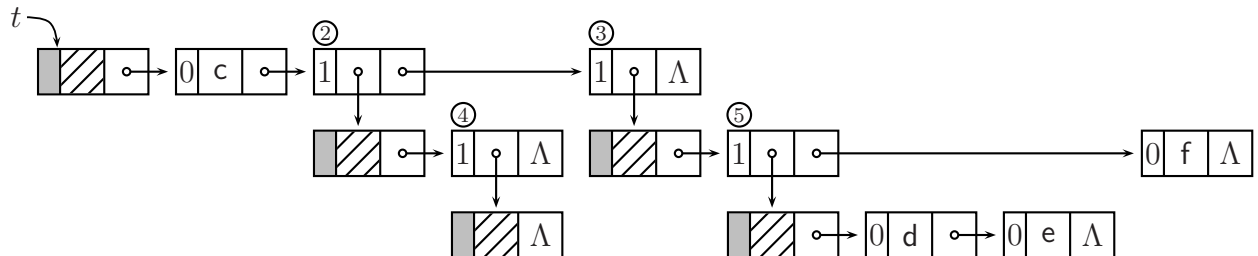
1  procedure TAIL(L)
  ▷ Returns a pointer to the tail of a non-null list L
2  if  $L = \Lambda$  or  $TAG(L) \neq -1$  then [output 'Error: argument is not a valid list'; stop]
3   $\beta \leftarrow LINK(L)$ 
4  case
5    :  $\beta = \Lambda$ : [output 'Error: tail is not defined for a null list'; stop]
6    :  $\beta \neq \Lambda$ : [call GETNODE( $\alpha$ );  $TAG(\alpha) \leftarrow -1$ ;  $LINK(\alpha) \leftarrow LINK(\beta)$ ; return( $\alpha$ )]
7  endcase
8  end TAIL

```

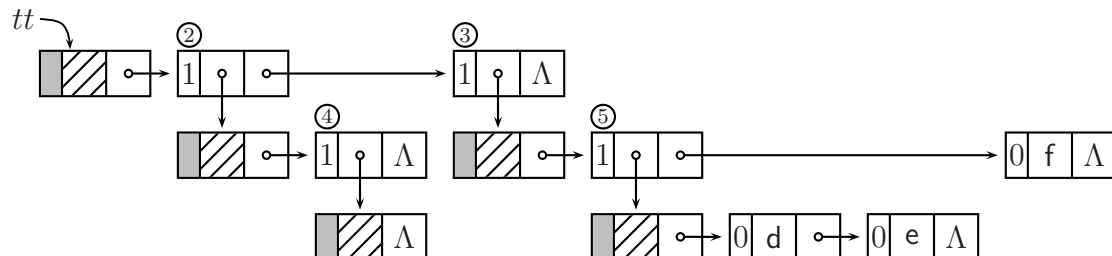
Procedure 12.2 Finding the tail of a list

Since the tail is always a list, the procedure creates a list head, sets its *LINK* field to point to the tail of the input list, and returns a pointer to the created list head (line 6). To illustrate, assume that L is the list shown in Figure 12.4; the following calls to TAIL will yield the indicated results.

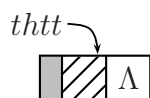
(a) $t \leftarrow TAIL(L)$



(b) $tt \leftarrow TAIL(TAIL(L))$



(c) $thtt \leftarrow TAIL(HEAD(TAIL(TAIL(L))))$



Constructing a list given its head and tail

The converse of finding the head and tail of a list is constructing a list given its head and its tail. The following examples illustrate the process.

P	Q	construct of P and Q
p	(i,l,a,r)	(p,i,l,a,r)
(p)	(i,l,a,r)	$((p),i,l,a,r)$
(p)	$((i,l,a,r))$	$((p),(i,l,a,r))$
p	$()$	(p)
$()$	$()$	$(())$

In procedure CONSTRUCT given below, the argument H is a pointer to an atomic node, if the head is an atom, or is a pointer to a list head, if the head is a list. The argument T is always a pointer to a list head.

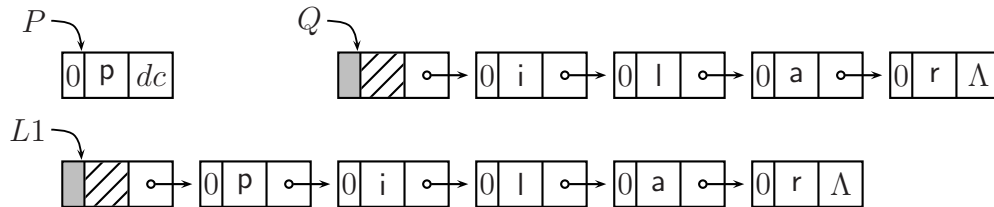
```

1  procedure CONSTRUCT( $H, T$ )
  ▷ Constructs a list whose head is  $H$  and whose tail is  $T$  and returns a pointer to the
  ▷ list head of the constructed list.
2  if  $H = \Lambda$  or  $T = \Lambda$  then [ output 'Error: invalid pointers'; stop ]
3  if  $TAG(T) \neq -1$  then [ output 'Error: the tail is not a list'; stop ]
4  call GETNODE( $\beta$ )
5  case
6    :  $TAG(H) = 0$  : [  $TAG(\beta) \leftarrow 0$ ;  $DATA(\beta) \leftarrow DATA(H)$  ]      ▷ head is an atom
7    :  $TAG(H) = -1$  : [  $TAG(\beta) \leftarrow 1$ ;  $DATA(\beta) \leftarrow H$  ]      ▷ head is a list
8  endcase
9   $LINK(\beta) \leftarrow LINK(T)$ 
10 call GETNODE( $\alpha$ );  $TAG(\alpha) \leftarrow -1$ ;  $LINK(\alpha) \leftarrow \beta$       ▷ list head of resulting list
11 return( $\alpha$ )
12 end CONSTRUCT
    
```

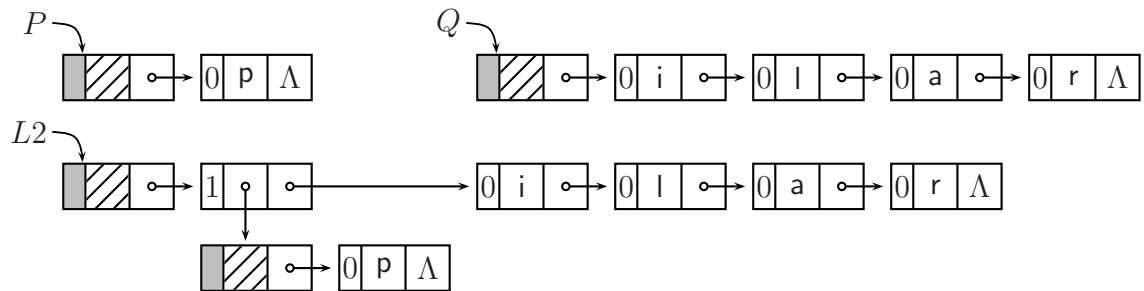
Procedure 12.3 Constructing a list given its head and tail

The node obtained in line 4 becomes an atomic node (line 6) or a list node (line 7); either way, its $LINK$ field points to the tail of the constructed list (line 9). The following examples should suffice to illustrate how the procedure works.

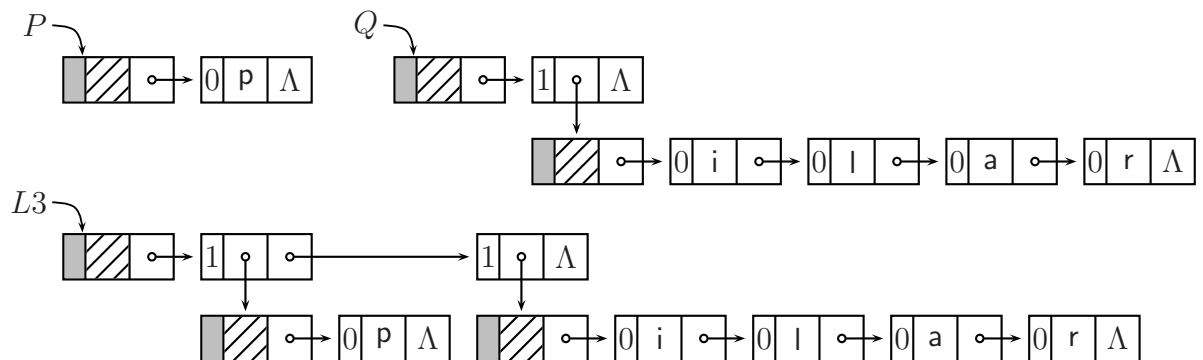
(a) $L1 \leftarrow CONSTRUCT(P, Q)$ where $P = p$ and $Q = (i,l,a,r)$



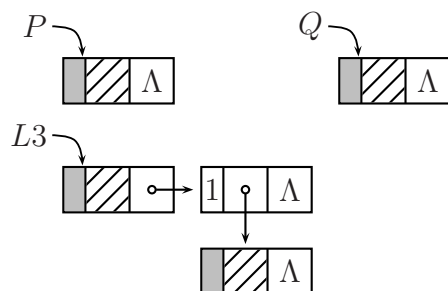
(b) $L2 \leftarrow \text{CONSTRUCT}(P, Q)$ where $P = (p)$ and $Q = (i, l, a, r)$



(c) $L3 \leftarrow \text{CONSTRUCT}(P, Q)$ where $P = (p)$ and $Q = ((i, l, a, r))$



(d) $L4 \leftarrow \text{CONSTRUCT}(P, Q)$ where $P = ()$ and $Q = ()$



Two other basic procedures on generalized lists, which we leave uncoded, are:

$\text{ATOM}(X)$ – returns *true* if X is an atom, and *false* otherwise

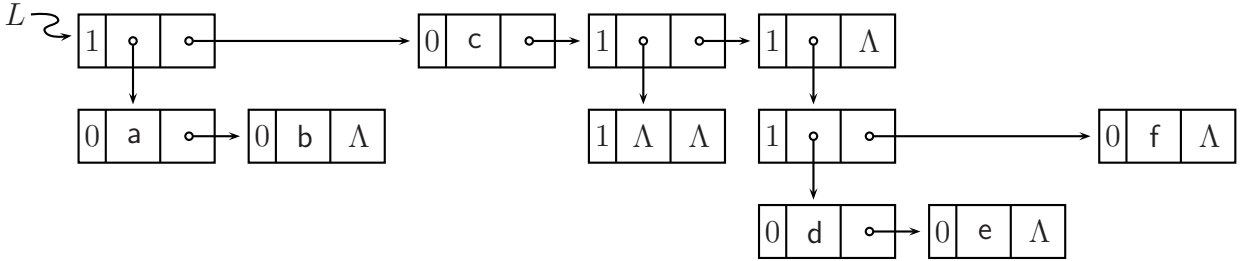
$\text{EQUAL}(X, Y)$ – returns *true* if X and Y are the same atom, and *false* otherwise

The programming language LISP provides five ‘pure’ procedures called CAR, CDR, CONS, ATOM and EQ which correspond to HEAD, TAIL, CONSTRUCT, ATOM and EQUAL (not necessarily as implemented above). With just these five pure functions, it is actually possible to perform complex manipulations of list structures (more of this in your course on Programming Languages).

12.4 Traversing pure lists

A recurring theme in our study of nonlinear structures is that of *traversal*. As with binary trees, trees, and graphs, we need to develop algorithms for systematically visiting the nodes of a generalized list. For now, we will consider the case in which the list is pure (no reentrancies nor cycles); in a subsequent section on ‘Automatic storage reclamation’, we will tackle the more complicated, albeit common, case in which the list is cyclic.

Consider the list $L = ((a,b),c,()),((d,e),f))$ of Figure 12.2, reprinted below for quick reference. Suppose we want to process the data stored in the atomic nodes of L ; one obvious sequence is a, b, c, d, e, f . If now we tilt the figure by 45° , it will resemble a binary tree, which, if traversed in *preorder*, will yield the desired sequence. Essentially, we ‘traverse down’ before we ‘traverse right’. Procedure TRAVERSE_PURE_LIST formalizes the process.



```

1  procedure TRAVERSE_PURE_LIST(L)
2  if L = Λ then return
3  call InitStack(S)
4  α ← L
5  loop
6  1:  if TAG(α) = 0 then call VISIT(DATA(α))
7      else [if DATA(α) = Λ then goto 2
8              else [call PUSH(S, α)
9                      α ← DATA(α); goto 1]]
10 2:  α ← LINK(α)
11    if α = Λ then if IsEmptyStack(S) then return
12        else [call POP(S, α); goto 2]
13  forever
14  end TRAVERSE_PURE_LIST
  
```

Procedure 12.4 Traversing a pure list

List nodes with a non-null *DATA* field are stored in a stack (line 8) so that after traversal of the structure below (i.e., the head) is done, traversal of the structure to the right (i.e., the tail) can be completed (line 12). For instance, assuming that α_i denotes the address of the list node numbered i in the figure above, the stack will take on the following states: *empty* $\Rightarrow (\alpha_1) \Rightarrow$ *empty* $\Rightarrow (\alpha_2) \Rightarrow$ *empty* $\Rightarrow (\alpha_3) \Rightarrow (\alpha_3, \alpha_5) \Rightarrow (\alpha_3) \Rightarrow$ *empty* (procedure terminates). Assuming that the call to VISIT in line 6 takes $O(1)$ time, the procedure clearly executes in $O(n)$ time for a list on n nodes [verify].

12.5 Automatic storage reclamation

An important task in a list processing environment is that of reclaiming nodes that are no longer in use. What makes the task somewhat more complicated than a simple call to `RETNODE`, which returns an unused node to the *avail* list, is that with generalized lists in which reentrancies and cycles may exist, there could be several references to any given node. Thus it is not always clear just when some such node is no longer needed and can already be reclaimed. In the early list processing systems, e.g., IPL-V, the task of determining when a node is no longer in use and returning it to the *avail* list is left to the programmer, since the programmer is supposed to be well acquainted with the lists he is using. This is *not* a good policy, for the following reasons. First, there are more important tasks than this housekeeping chore that the programmer must attend to, such as solving the problem at hand. Second, the programmer may not in fact be competent to do storage reclamation considering the complexities that may arise in processing cyclic list structures.

In contemporary list processing systems, including LISP, reclaiming unused nodes is a system function. To put the problem in perspective, imagine the following simple scenario. A user program, say *P*, is currently executing in a list processing environment. The runtime system, say *S*, maintains a list of available nodes (the *avail* list) from where *P* obtains the building blocks for the lists it uses. At any point during the execution of *P*, there are three types of nodes, namely: (a) *free* nodes, which are the nodes on the *avail* list; (b) *used* nodes, which are the nodes which comprise the lists being processed by *P* and are accessible to *P*; and (c) *unused* nodes, which are nodes no longer accessible to *P*. A fundamental question in automatic (read: system-initiated) storage reclamation is: ‘When should *S* reclaim unused nodes?’ To which there are two diametrically opposed answers, to wit: ‘Reclaim unused nodes when the *avail* list becomes empty’, and ‘Reclaim unused nodes the moment they are created.’ Two principal methods for automatic storage reclamation, which are based on these strategies are, respectively, the **garbage-collection technique** and the **reference-counter technique**.

Garbage-collection technique

In this method, unused nodes are allowed to accumulate (to become ‘garbage’) until a request for a node can no longer be satisfied because the *avail* list is empty; at this point normal list processing is interrupted and ‘garbage collection’ commences. Garbage collection consists of two phases, namely, **marking** and **gathering**. Initially, all nodes in list space are marked as ‘not in use’. Then, in the *marking phase*, all nodes accessible to the running programs are marked as ‘in use’; all other nodes not thereby marked are considered as ‘not in use’. In the *gathering phase*, the entire list space is scanned, and all nodes marked ‘not in use’ are linked together to comprise the *avail* list. Those nodes marked as ‘in use’ are marked as ‘not in use’ in preparation for the next marking session.

We note at this point some important aspects of the method. Actual procedures to implement marking and gathering will be subsequently discussed.

1. The idea of letting garbage pile up has merits. Consider again the scenario presented above. It could be that there are enough nodes in the *avail* list for *P* to execute successfully without reusing any node. Thus reclaiming nodes which won’t be used anyway is wasted effort.

2. To implement the marking step, a *mark bit* is associated with every node in list space. The mark bit may be stored in a *MARK* field in the node itself, or it may be maintained in a bit map stored elsewhere in memory.
3. Marking must be thorough. For instance, a program might be in the process of constructing a list structure when a request for a node finds the *avail* list empty triggering a call to the garbage collector. It is extremely important that the marking algorithm mark the nodes in such a partially constructed list as ‘in use’; otherwise, they will be gathered and returned to the free list, unbeknown to the program constructing the list. This causes the program to behave erratically, and debugging can be difficult since the problem is not so much with the program itself as it is with the garbage collection.
4. Garbage collection runs very slowly when most of list space is in use. The marking algorithm has more nodes to mark; the gathering algorithm scans all of list space only to collect a few unused nodes. Soon the *avail* list is again empty, and the process repeats. This can lead to a phenomenon called *thrashing* in which a system routine, in this case the garbage collector, takes up almost all of CPU time with little left for normal user processing. One way to avoid this is to terminate processing altogether if the garbage collector cannot find more than a specified minimum number of free nodes; this minimum can be user-specified.
5. Normal list processing must be suspended while marking or gathering is in progress. In certain real-time applications, such interruptions may not be permissible, thus making the garbage-collection technique unsuitable.

The above considerations are from the system’s end. From the user’s end, it is important to see to it that lists are well-formed and that pointer fields contain valid pointer data at all times (‘don’t cares’ won’t do). Otherwise, even the best written garbage collector may end up generating garbage!

Reference-counter technique

In this method, a reference count field, say *REF*, is maintained in each node in list space in which the number of references to the node is stored. A node, say node α , whose *REF* field becomes zero is no longer in use and is returned to the *avail* list. The reference count of the nodes referred to by node α is then decremented by 1; some of these nodes may have their reference count drop to zero. These nodes are likewise returned to the free list, and the reference count of the nodes they point to decreased by 1. And so on. The idea is to reclaim a node the moment it is found to be *unused*, i.e., its *REF* field is zero.

We take note of the following important points regarding this method.

1. The size of the *REF* field must be chosen carefully. If it is too large, a lot of space is wasted; if it is too small, overflow may occur and the method bogs down. In any event, the total amount of space allocated for reference counts can be significant. One way to reduce this amount is by using a list head to begin every list or sublist and requiring that only list heads may be multiply referenced. With this restriction, only list heads will need a reference count. Once the reference count of a list head drops to zero, all the nodes on its list can automatically be reclaimed.

- Freeing a single node may lead to a cascade of other nodes being freed also. For instance, if the external pointer to a *pure* list is removed, then all the nodes in the list will have their reference count drop to zero and each will be reclaimed one after the other in a manner somewhat akin to a chain reaction. This can lead to some significant delay in normal list processing.

To correct this, Weizenbaum proposed a variation on the reference-counter technique that is essentially a compromise between the policy of letting unused nodes pile up and the policy of reclaiming unused nodes immediately. The strategy goes like this: when a node, say node α , whose reference count becomes zero is reclaimed it is attached at the rear of the *avail* list *with the contents of its pointer fields kept intact and unprocessed*. It is only when node α gets to the front of the *avail* list and is allocated that the nodes it points to are finally located and their reference count decremented. This delayed processing of reference counts prevents the occurrence of nodes being reclaimed in spurts, thus avoiding undue interruptions to normal list processing. Likewise, some efficiency is gained by not immediately reclaiming nodes which may not be reused anyway.

There is however one problem that arises with this strategy, namely: how do we link together the nodes on the *avail* list if the contents of their pointer fields, say *CAR* and *CDR*, must be preserved and cannot be used for this purpose. One obvious solution is to define another pointer field; another is to harness the *REF* field, which is now irrelevant, as a pointer field.

- With cyclic list structures, it is possible to have collections of nodes with nonzero reference counts but which are nevertheless inaccessible. Figure 12.8 shows such an ‘island of unused but unreclaimed nodes’. These nodes are unused because they are not accessible; however, they cannot be reclaimed because their reference count is not zero.

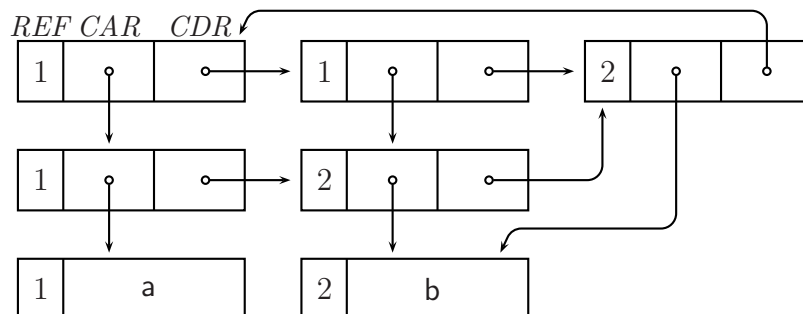


Figure 12.8 Unused but unreclaimed nodes (reference-counter technique)

12.5.1 Algorithms for marking and gathering

In the rest of this session, we will look more closely at the garbage-collection technique for automatic storage reclamation, in particular at the algorithms which implement the marking phase. It is important to appreciate the conditions under which a marking algorithm is invoked, namely, normal list processing is suspended because space on the *avail* list is exhausted. Thus a marking algorithm must execute very fast (so that user processing can resume as soon as possible) and should require as little additional space as possible (because there is very little space available in which it can execute). We can state this more formally as follows: The *ideal* marking algorithm should execute in linear time, in bounded workspace, and without the need for auxiliary tag bits. Unfortunately, none of the actual marking algorithms that have been devised thus far satisfies all three requirements. The marking algorithms discussed below, and implemented as procedures MARK_LIST_1, MARK_LIST_2 and MARK_LIST_3, constitute a progression towards this ideal although none quite makes it, as we will see shortly.

We make the following general assumptions with respect to all three marking procedures.

1. The list to be marked is cyclic, which subsumes both the pure and reentrant kind.
2. List nodes have two pointer fields named *CAR* and *CDR* which contain pointer data only; atomic nodes contain atomic data only.
3. Each node has an associated *MARK* bit; whether this is stored in a bit field or a bit map is irrelevant. Initially, all nodes are marked ‘not in use’ (mark bit = 0).
4. There is available a procedure, $\text{ATOM}(\alpha)$, which returns *true* if α points to an atomic node, and *false*, otherwise. By what mechanism it recognizes an atom (address-range discrimination, bit map, etc) is irrelevant.
5. Λ is the address of an atom such that $\text{MARK}(\Lambda) = 1$.

Now the procedures.

Marking a cyclic list structure using a pointer stack

```

1  procedure MARK_LIST_1(L)
2  call InitStack(S)
3   $\alpha \leftarrow L$ 
4  loop
5     $\text{MARK}(\alpha) \leftarrow 1$ 
6    if not ATOM( $\alpha$ ) then [call PUSH(S, CDR( $\alpha$ )); call PUSH(S, CAR( $\alpha$ ))]
7  1: if IsEmptyStack(S) then return
8    else [call POP(S,  $\alpha$ ); if MARK( $\alpha$ ) = 1 then goto 1]
9  forever
10 end MARK_LIST_1
```

Procedure 12.6 Marking a cyclic list using a stack of pointers

Procedure MARK_LIST_1 is a generalization of procedure TRAVERSE_PURE_LIST, which traverses a pure list in preorder, for the case in which the list is cyclic. Essentially, MARK_LIST_1 performs a preorder traversal of a *spanning tree* of the cyclic list structure being marked. It marks the immediately accessible node L as ‘in use’ and then proceeds to similarly mark all nodes accessible from L via chains of *CAR* and *CDR* links. It uses an auxiliary stack \mathbb{S} to store pointers to nodes awaiting marking, processing *CAR* chains before *CDR* chains. Procedure MARK_LIST_1 takes $O(n)$ time to mark n nodes, which is as good as it can get. However, it is not too useful in practice, since when it is invoked, there is hardly any available space for the pointer stack.

Marking a cyclic list structure using link inversion and auxiliary tag bits

In Session 6, we saw that by inverting links we can traverse a binary tree without using a stack, albeit at the expense of an additional tag bit per node (you may want to review section 6.6.1 at this point). The application of this idea to generalized lists yields the

```

1  procedure MARK_LIST_2( $L$ )
  ▷ Implements the Schorr-Waite-Deutsch algorithm for marking cyclic list structures using
  ▷ link inversion and auxiliary tag bits. Procedure marks node  $L$  and all nodes accessible
  ▷ from  $L$  via any chain of pointers in the CAR and CDR fields of nonatomic nodes.
2    if ATOM( $L$ ) then [ MARK( $L$ )  $\leftarrow$  1; return ]
3     $\beta \leftarrow \Lambda$ ;  $\alpha \leftarrow L$ 
  ▷ Mark node
4    1: MARK( $\alpha$ )  $\leftarrow$  1
  ▷ Process CAR path
5     $\sigma \leftarrow \text{CAR}(\alpha)$ 
6    case
7      : MARK( $\sigma$ ) = 1 : exit
8      : ATOM( $\sigma$ )    : [ MARK( $\sigma$ ) = 1; exit ]
9      : else        : [ CAR( $\alpha$ )  $\leftarrow \beta$ ;  $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; goto 1 ]
10   endcase
  ▷ Process CDR path
11  2:  $\sigma \leftarrow \text{CDR}(\alpha)$ 
12   case
13     : MARK( $\sigma$ ) = 1 : exit
14     : ATOM( $\sigma$ )    : [ MARK( $\sigma$ ) = 1; exit ]
15     : else        : [ TAG( $\alpha$ )  $\leftarrow$  1; CDR( $\alpha$ )  $\leftarrow \beta$ ;  $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; goto 1 ]
16   endcase
  ▷ Ascend via inverted links
17  3: if  $\beta = \Lambda$  then return
18     else if TAG( $\beta$ ) = 0 then [  $\sigma \leftarrow \text{CAR}(\beta)$ ; CAR( $\beta$ )  $\leftarrow \alpha$ 
19                                $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; goto 2 ]
20     else [  $\sigma \leftarrow \text{CDR}(\beta)$ ; CDR( $\beta$ )  $\leftarrow \alpha$ 
21            $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; TAG( $\beta$ )  $\leftarrow$  0; goto 3 ]
22  end MARK_LIST_2

```

Procedure 12.7 Schorr-Waite-Deutsch algorithm for marking cyclic list structures

Schorr-Waite-Deustch algorithm for marking cyclic list structures which is implemented as procedure `MARK_LIST_2`. The additional tag bit per node, which may be stored in a bit field or a bit map, is used to indicate whether it is the *CAR* link or the *CDR* link of the node which is inverted at some instance during the marking process. Initially, all nodes have a tag of 0; subsequently, the tag is kept at 0 when a *CAR* link is inverted and is set to 1 when a *CDR* link is inverted. When marking reaches the end of a *CAR* or *CDR* chain, the setting of the tag bit indicates whether ascent is from below (lines 18–19) or from the right (lines 20–21), after which the *CAR* or *CDR* link, as the case may be, is restored.

The structure of `MARK_LIST_2` is very similar to that for traversing a binary tree in preorder by link inversion (Procedure 6.14), which we have described at length in section 6.6.1. However, there are two conditions which are peculiar to the present case which must be considered, namely:

1. A node may be encountered several times during the traversal because of reentrancies or cycles. To avoid infinite looping, a node is marked on first encounter and is ignored on subsequent encounters (line 7 and line 13); note that the **exit** in lines 7 and 13 means exit the **case** statement, not the procedure!
2. A node may have no link fields (it is atomic) and as such there are no links to invert; in this case, the node is simply marked as ‘in use’ (line 8 and line 14). Note that this code also applies to the case where the *CAR* or *CDR* field is null, since we consider Λ to be an atom. In this case, $MARK(\Lambda)$ is redundantly set to 1.

Procedure `MARK_LIST_2` executes in linear time and in bounded workspace, but it requires an additional tag bit per node; thus it falls short of the requisites of an ideal marking algorithm. In any event, it is a vast improvement over `MARK_LIST_1`. Can we do better still?

Marking a cyclic list structure using link inversion and a bit stack

The Schorr-Waite-Deutsch algorithm for marking cyclic list structures requires one bit per node whose setting, 0 or 1, indicates which pointer in the node is currently inverted. However, it is extremely unlikely that *all* list nodes will have one link inverted at any one time during the marking process. While some list nodes will have one link inverted, the rest will have their links in the normal position. Furthermore, the last link inverted is the first link to be restored (as you may easily verify by walking through `MARK_LIST_2` using a sample list). Thus it is not really necessary to allocate a tag bit per node to indicate which link is inverted; it suffices to use a bit stack in which there is an entry for each inverted link *only*, say a 0 for an inverted *CAR* link and a 1 for an inverted *CDR* link. Upon ascent, the entry at the top of the bit stack will indicate whether ascent is from below (if 0) or from the right (if 1), and which link field can now be restored.

This is the idea behind **Wegbreit’s algorithm** which is implemented as procedure `MARK_LIST_3`. If we appear to be too stingy with bits, remember that when a marking algorithm is invoked, there is very little space for it to operate in.

```

1  procedure MARK_LIST_3( $L$ )
  ▷ Implements Wegbreit's algorithm for marking cyclic list structures using link
  ▷ inversion without tag bits but with a bit stack. Procedure marks node  $L$  and all
  ▷ nodes accessible from  $L$  via any chain of pointers in the CAR and CDR fields
  ▷ of nonatomic nodes.
2    if ATOM( $L$ ) then [ MARK( $L$ )  $\leftarrow$  1; return ]
3    call InitStack( $\mathbb{S}$ )
4     $\beta \leftarrow \Lambda$ ;  $\alpha \leftarrow L$ 
  ▷ Mark node
5    1: MARK( $\alpha$ )  $\leftarrow$  1
  ▷ Process CAR path
6     $\sigma \leftarrow$  CAR( $\alpha$ )
7    case
8      : MARK( $\sigma$ ) = 1 : exit
9      : ATOM( $\sigma$ )      : [ MARK( $\sigma$ ) = 1; exit ]
10     : else           : [ call PUSH( $\mathbb{S}$ , 0); CAR( $\alpha$ )  $\leftarrow$   $\beta$ ;  $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; goto 1 ]
11  endcase
  ▷ Process CDR path
12  2:  $\sigma \leftarrow$  CDR( $\alpha$ )
13  case
14    : MARK( $\sigma$ ) = 1 : exit
15    : ATOM( $\sigma$ )      : [ MARK( $\sigma$ ) = 1; exit ]
16    : else           : [ call PUSH( $\mathbb{S}$ , 1); CDR( $\alpha$ )  $\leftarrow$   $\beta$ ;  $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \sigma$ ; goto 1 ]
17  endcase
  ▷ Ascend via inverted links
18  3: if  $\beta = \Lambda$  then return
19      else [ call POP( $\mathbb{S}$ ,  $flag$ )
20              if  $flag = 0$  then [  $\sigma \leftarrow$  CAR( $\beta$ ); CAR( $\beta$ )  $\leftarrow$   $\alpha$ 
21                                   $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; goto 2 ]
22                                  else [  $\sigma \leftarrow$  CDR( $\beta$ ); CDR( $\beta$ )  $\leftarrow$   $\alpha$ 
23                                           $\alpha \leftarrow \beta$ ;  $\beta \leftarrow \sigma$ ; goto 3 ] ]
24  end MARK_LIST_3

```

Procedure 12.8 Wegbreit's algorithm for marking cyclic list structures

Procedure MARK_LIST_3 executes in linear time, in unbounded workspace (because it uses a stack), without an additional tag bit per node; as such, it falls short of the requisites of an ideal marking algorithm. In any case, it is an improvement over MARK_LIST_2 in terms of space utilization.

In summary, we have:

	MARK_LIST_1	MARK_LIST_2	MARK_LIST_3
Executes in linear time	Yes	Yes	Yes
Executes in bounded workspace	No	Yes	No
Uses no tag bits	Yes	No	Yes

Gathering

Once all accessible nodes have been marked as ‘in use’ (mark bit = 1) by a marking algorithm, gathering those that are not (mark bit = 0) is a straightforward process. Assume that the list space is from address $m1$ through $m2$, and that each node consists of c addressable units. The following procedure performs a linear sweep of the list space and reconstitutes the *avail* list by linking together all unused nodes using the *CDR* field. It resets to 0 the the mark bit of used nodes in preparation for the next marking session.

```

1  procedure GATHER
2     $avail \leftarrow \Lambda$ 
3    for  $\alpha \leftarrow m2 - c + 1$  to  $m1$  by  $-c$  do
4      if  $MARK(\alpha) = 0$  then [ $CDR(\alpha) \leftarrow avail$ ;  $avail \leftarrow \alpha$ ]
5      else  $MARK(\alpha) \leftarrow 0$ 
6    endfor
7    end GATHER

```

Summary

- Of all the ADT’s discussed in this book the generalized list is the most abstruse. The ordinary programmer who wants to use generalized lists on, say, an artificial intelligence application would normally use a programming language such as LISP or Scheme in which the generalized list is the fundamental data type. He will be least concerned with the issues addressed in this session, such as representing lists in computer memory or storage reclamation, as this will be handled by the runtime system.
- The material in this session, although brief, is for the serious student of Computer Science who may want to develop his own list processing system. Then the issues and concepts discussed here will be pertinent to the task at hand.
- A recurring theme in the study of data structures and algorithms is how techniques developed for one structure carry over to other structures. For instance, traversal by link inversion discussed earlier in connection with binary trees is the template for the Schorr-Waite-Deutsch algorithm for marking cyclic list structures.

Exercises

1. Given the generalized list $L = (a, (b, (c, (d, (e))))$, what is:

- | | | |
|------------------------|-----------------------------------|---|
| (a) $\text{length}(L)$ | (d) $\text{tail}(L)$ | (g) $\text{atom}(\text{head}(\text{head}(\text{tail}(L))))$ |
| (b) $\text{depth}(L)$ | (e) $\text{head}(\text{tail}(L))$ | (h) $\text{null}(\text{tail}(\text{head}(\text{tail}(L))))$ |
| (c) $\text{head}(L)$ | (f) $\text{tail}(\text{tail}(L))$ | |

2. Show the representation of the list $L = ((a, (b, (c, d))), (c, d), ((), e, (c, d)))$ as a *reentrant list* using the node structure:

- (a) $[TAG, DATA, LINK]$ for both atomic nodes and list nodes
 - (b) $[TAG, DATA]$ for atomic nodes and $[TAG, CAR, CDR]$ for list nodes
 - (c) $[DATA]$ for atomic nodes and $[CAR, CDR]$ for list nodes with address range discrimination to distinguish between the two
3. Using the node structure $[TAG, DATA, LINK]$ for both atomic nodes and list nodes, show the representation of the list $L = (((a, b), P), P, (P, Q), (Q, h))$, where $P = (c, d, e)$ and $Q = (f, g)$, as a reentrant list
 - (a) without list heads
 - (b) with list heads
 4. Let $L = (x_1, x_2, x_3, \dots, x_n)$. Write a recursive EASY procedure CREATE which takes in L as input and generates the internal representation of L as a *pure* list using the node structure $[TAG, DATA, LINK]$. The procedure should return a pointer to the list (as in Figure 12.2) if L is non-null; or Λ otherwise.
 5. Let P point to a pure list represented using the node structure $[TAG, DATA, LINK]$, e.g., a list generated by procedure CREATE in Item 4. Write a recursive EASY procedure COPYLIST which makes, and returns a pointer to, an exact copy of list P .
 6. Let P point to a pure list represented using the node structure $[TAG, DATA, LINK]$, e.g., a list generated by procedure CREATE in Item 4. Write a recursive EASY procedure DEPTH which finds the depth of list P .
 7. Let P and Q point to pure lists represented using the node structure $[TAG, DATA, LINK]$. Write a recursive EASY procedure EQUIV which returns *true* if P and Q are exactly the same, and *false* otherwise.
 8. Let P point to a pure list represented using the node structure $[TAG, DATA, LINK]$. Write a recursive EASY procedure INVERT which inverts *in-place* the elements of list P . For instance, the pure list $(a, (b, c), (d, e, (f, g, h)), (i, j))$ inverted becomes the list $((i, j), (d, e, (f, g, h)), (b, c), a)$.
 9. Let P point to a pure list represented using the node structure $[TAG, DATA, LINK]$. Write a recursive EASY procedure REVERSE which reverses *in-place* the elements of list P . For instance, the pure list $(a, (b, c), (d, e, (f, g, h)), (i, j))$ reversed becomes the list $((j, i), ((h, g, f), e, d), (c, b), a)$.
 10. Using the node structure $[TAG, DATA, LINK]$, show the representation of the list $(a, (b, c), (d, e, (f, g, h)), (i, j))$ as a threaded pure list.
 11. Using the node structure $[REF, DATA]$ for atomic nodes and $[REF, CAR, CDR]$ for list nodes, show a cyclic list structure on 10 nodes which constitute an ‘island of unused but unreclaimable nodes’ by the reference-counter technique.

Bibliographic Notes

KNUTH1[1997], STANDISH[1980] and TENENBAUM[1986] are excellent sources of the basic material on generalized lists discussed in this session. Procedures HEAD, TAIL and CONSTRUCT are implementations in EASY of Algorithms HEAD, TAIL and CONSTRUCT given in TREMBLAY[1976]. Procedure TRAVERSE_PURE_LIST and procedure TRAVERSE_THREADED_PURE_LIST are implementations of Algorithm 5.2 (*Traverse X using a pushdown stack A*) and Algorithm 5.3 (*Traverse X using threads in bounded workspace*) given in STANDISH[1980]; the two algorithms are shown to take exactly the same number of steps, viz., $1 + 2 * n$, where n is the number of nodes. Procedures MARK_LIST 1, MARK_LIST 2 and MARK_LIST 3 are implementations of Algorithm 5.4 (*Marking cyclic list structures with a pointer stack*), Algorithm 5.5 (*Schorr-Waite-Deutsch marking of cyclic list structures*) and Algorithm 5.6 (*Wegbreit marking with a bit stack*), respectively, given in STANDISH[1980]. KNUTH1[1997], pp. 415–420, presents a progression of four marking algorithms (Algorithms A through D) which culminates in Algorithm E (*Marking*) which is the Schorr-Waite-Deutsch algorithm. TENENBAUM[1986], pp. 682–683, gives a Pascal version of the same algorithm.

STANDISH[1980], pp. 80–82, discusses another marking algorithm, after Lindstrom, which uses no stack or auxiliary tag bits; however it executes in $O(n \log n)$ time.

SESSION 13

Sequential tables

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define the problem of ‘searching’ in computer applications and cite typical instances of the problem.
2. Describe the table ADT and explain the type of operations supported by it.
3. Describe the different methods for searching a sequential table and give an analysis of the performance of each.

READINGS KNUTH3[1998], pp. 392–422; STANDISH[1980], pp. 132–141; CORMEN[2001], pp.222–223.

DISCUSSION

In this and the next two sessions we will consider data structures and associated algorithms to solve a simple, albeit important, problem which arises in various computer applications. This is the problem of finding the data, stored somewhere in the memory of a computer, given some information which identifies the desired data. We have already encountered instances of this problem in the previous sessions. For example, in topological sorting (Session 5), we wanted to find, given the label of an object, the data pertaining to the object: its count of direct predecessors and its list of direct successors. In UPCAT processing (Session 11), we wanted to find, given an applicant’s choice of degree course, the data pertaining to the course: its predictor, its quota, and its count and list of current qualifiers. Other instances of the problem readily come to mind. For instance, in a computerized SRS, we may want to find the data (student name, address, course, college, grades, and so forth) given a student’s student number. A compiler stores somewhere in computer memory the attributes of user-defined identifiers during compilation; the runtime system needs to find such information when a reference is made to these identifiers during program execution. And so on.

To put the problem in a general setting, we assume that we have a collection of n records, where each record is an aggregate of data items with an associated *key*. We assume that the n keys are distinct, so that each record is uniquely identified by its key.

Such a collection of records is called a **table** or a **file** depending on its size (small or large), lifetime (short-lived or long-lived) and storage medium (internal or external memory).

In general, we think of a file as a possibly large collection of records that resides more or less permanently in external memory. In contrast, we think of a table as a not-so-large collection of records that resides in internal memory and which exists only during the execution of a particular program. While our use of these terms is not always precise nor consistent (e.g., we might save a table onto a diskette and then call it a file), it is well to remember the distinction between the two. Differences in the size or life span of, or speed of access to, the data are clearly important considerations in determining how to represent the data in computer memory and implement the required operations.

In this and the next two sessions we will be concerned mainly with tables.

13.1 The table ADT

Figure 13.1 shows our view of a table in the abstract.

KEY	DATA
k_1	d_1
k_2	d_2
k_3	d_3
\vdots	\vdots
k_i	d_i
\vdots	\vdots
k_{n-1}	d_{n-1}
k_n	d_n

Figure 13.1 A table as a collection of records

In Figure 13.1, (k_i, d_i) is a record, and d_i represents the aggregation of data associated with, or identified by, the key k_i . We shall refer to the collection of records (k_i, d_i) , $i = 1, 2, \dots, n$ as the table T . Some of the operations usually performed on T include:

1. Given a search argument K , find the record (k_j, d_j) in T such that $K = k_j$. If such a record is found, the search is said to be successful; otherwise, the search is said to be unsuccessful.
2. Insert a new record (k_j, d_j) into T . An insert operation is normally preceded by an unsuccessful search operation, i.e., a new record with key k_j is inserted into T only if there is no record yet in T with key k_j . In effect, we perform a search-insert operation.
3. Delete the record (k_j, d_j) from T .
4. Find the record with the smallest, or largest, key.
5. Given the key k_j , find the record with the next smaller, or next larger, key.

Then, of course, we have the usual operations such as initializing T , determining whether T is empty, and so forth.

As with any ADT, there are different ways of implementing the table T . We can use sequential or linked allocation, or both. Which representation to use depends on such factors as:

1. the size of the key space U_k , i.e., the number of possible distinct keys; for instance, the keys may be any of the integers 1 through 1000, or the keys may be identifier names of at most six characters from the character set A through Z, 0 through 9, such that the first character is alphabetic (1,617,038,306 possible keys in all)
2. whether the table is static or dynamic, i.e., whether or not it is subject to insertions and deletions while in use
3. the type and mix of operations performed on the table, e.g., an implementation of the table may support operations 1, 2 and 3 but not 4 and 5

13.2 Direct-address table

If the key space U_k is of size m , where m is not too large, and if we have the option of choosing what keys to use, then we can implement a table simply as an array of m cells, and use array indices as keys such that $k_i = i$. This data structure is called a **direct-address table**.

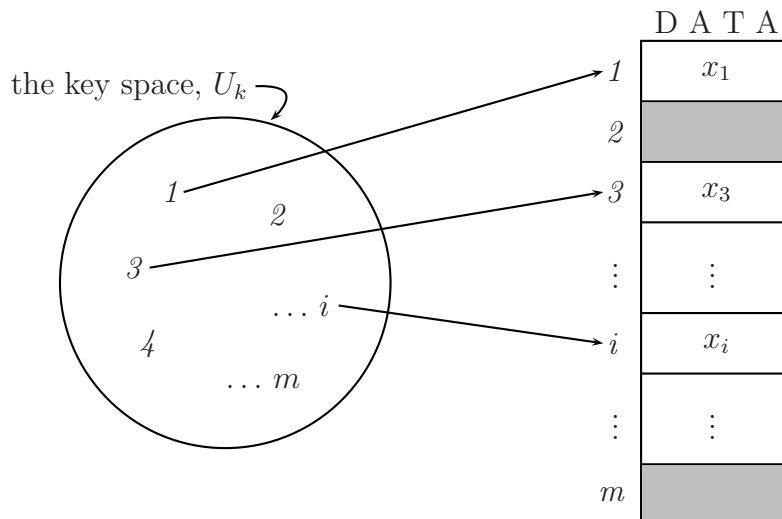


Figure 13.2 A direct-address table (array indices serve as keys)

Every key in U_k is assigned a cell, or slot, in the table in which the data associated with the key is stored. We take note of the following considerations when using a direct-address table.

1. There is no need to explicitly store the key k_i in the table since the cell index i already indicates the position in the table where the associated data d_i is stored.

2. If there is not enough room in a cell for the data d_i , we can store d_i in some structure external to the table and plant a pointer to this structure in cell i .
3. Whether d_i or a pointer to d_i is stored in cell i , we must have some way of indicating unused cells, shown shaded in the figure, which correspond to unused keys from U_k .

In a direct-address table, searching is eliminated, since cell i contains d_i or a pointer to d_i . Likewise, since every possible key has its preassigned cell, insertion and deletion are trivially easy to implement.

An example of the use of a direct-address table can be found in Session 5 in Knuth's solution to the topological sorting problem. In this example, the objects to be sorted are labeled $1, 2, \dots, n$, and these labels serve as the index, or the key, to the data associated with each object, namely, the number of direct predecessors and the labels of the direct successors of the object. The table in this case is implemented using both sequential and linked allocation. The former utilizes two parallel arrays of size $m \geq n$ named *COUNT* and *LIST*; the latter utilizes a singly-linked linear list where each node in the list consists of two fields named *SUCC* and *NEXT*. For any object i , $COUNT(i)$ contains the number of direct predecessors of object i , and $LIST(i)$ contains a pointer to a linked list in which the direct successors of object i are stored. Thus, a portion of the satellite data d_i is stored in cell i of the table and the rest is stored in a linked list which is directly accessible from cell i . Unused cells in the table are indicated by an entry of -1 in the *COUNT* field.

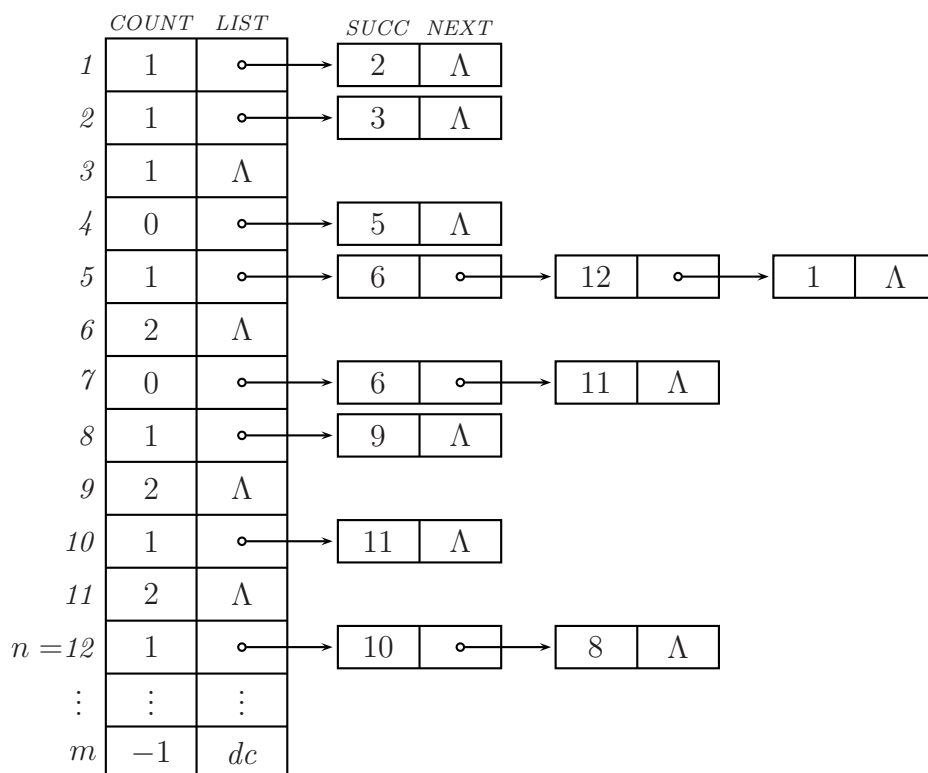


Figure 13.3 An example of a direct-address table

13.3 Static sequential tables

When the key space U_k is large it becomes impractical, if not impossible, to preassign each possible key from U_k a specific cell in a table and use direct addressing to access the data associated with a key. A more rational approach is to use a table whose size depends on the number of keys from U_k that are actually used, and to store a record such as (k_i, d_i) in some available, not preassigned, slot in the table. The records in such a table may be stored in T

1. in essentially random order
2. according to frequency of access, i.e., the more frequently accessed records are placed at the beginning of the table
3. in increasing or decreasing order of the keys
4. according to some function $h(k_i)$ which maps k_i into some slot in the table

Clearly, direct addressing can no longer be used to locate records in such a table; instead, we search the table by *comparing keys*.

We consider first the case in which the table is *static*, i.e., we do not insert records into, nor delete records from, the table while it is in use. The only operation of interest to us is searching the table. There are important applications which satisfy these assumptions. For instance, in the selection phase of UPCAT processing discussed in Session 11, we needed to search the course table for a UP campus (see Figure 11.27) when assigning campus qualifiers to the different degree programs within the campus. To determine whether applicant x may be assigned to his or her choice of degree program, say p , we searched the course table using as key the course code for p in order to access the data pertaining to p , such as its predictor, quota, current number of qualifiers, and so on. We updated part of the satellite data which constituted a record, but we did not insert records into nor delete records from the table during the entire processing.

Figure 13.4 depicts a static sequential table T which consists of n records (k_i, d_i) stored in an array, or parallel arrays, of size $m \geq n$. The table may be unordered or ordered. We assume that it is represented using the data structure $\mathbb{T} = [KEY(1:m), DATA(1:m), n]$. In each of the procedures given below, we assume that a pointer to \mathbb{T} is passed to the procedure thus giving it access to the records in T .

	KEY	DATA
1	k_1	d_1
2	k_2	d_2
\vdots	\vdots	\vdots
i	k_i	d_i
\vdots	\vdots	\vdots
n	k_n	d_n
\vdots	\vdots	\vdots
m	dc	dc

Figure 13.4 A static sequential table

13.3.1 Linear search in a static sequential table

Given a search key, say K , with the instruction to find the record in T whose key matches K , we proceed as follows: we compare K with K_1 ; if $K = K_1$, the search is done; otherwise, we compare K with K_2 ; if $K = K_2$, the search is done; otherwise, we compare K with K_3 ; and so on. This technique, called *sequential* or *linear search*, terminates in one of two ways: we find the desired record, or we discover that what we are looking for is not in the table. Procedure LINEAR_SEARCH_1 formalizes this process.

The input to the procedure is the table T , represented by the data structure \mathbb{T} , and a search key K . The procedure returns the index of the record in T whose key matches K , if such a record exists; otherwise, the procedure returns 0.

```

1  procedure LINEAR_SEARCH_1( $\mathbb{T}$ ,  $K$ )
2  for  $i \leftarrow 1$  to  $n$  do
3    if  $K = \text{KEY}(i)$  then return( $i$ )     $\triangleright$  successful search
4  endfor
5  return(0)                                 $\triangleright$  unsuccessful search
6  end LINEAR_SEARCH_1

```

Procedure 13.1 Linear search

For now, let us take note of two things about this simple procedure.

- (a) Only the *KEY* component of a record is needed to perform the search for the record whose key matches K . We assume that once we have found the desired record, we gain access to all the data which comprise the record. How this data is used is a different matter altogether and is not relevant to the search procedure. This observation generally applies to all the other search algorithms to be subsequently discussed.
- (b) It is possible to substantially reduce the execution time of the procedure. (Think about it before reading on.)

To see how (b) can be accomplished, let us rewrite the code so that we make explicit what the **for** statement actually does:

```

 $i \leftarrow 1$                                  $\triangleright$  initialization
loop
  if  $K = \text{KEY}(i)$  then return( $i$ )     $\triangleright$  test for matching keys
   $i \leftarrow i + 1$                      $\triangleright$  incrementation
  if  $i > n$  then return(0)             $\triangleright$  test for table exhaustion
forever

```

The main computational effort is incurred within the loop by the two tests: the test for matching keys and the test for table exhaustion. It is clear that we cannot do away with the match test; hence, we must find a way to eliminate the test for table exhaustion. How? By making sure that the match test always succeeds.

Procedure LINEAR_SEARCH_2 implements this idea through the simple technique of adding a dummy $(n + 1)$ th record to the table such that $\text{KEY}(n + 1) = K$.

```

1  procedure LINEAR_SEARCH_2( $\mathbb{T}, K$ )
2     $i \leftarrow 1$ 
3     $KEY(n+1) \leftarrow K$ 
4    loop
5      if  $K = KEY(i)$  then exit
6       $i \leftarrow i + 1$ 
7    forever
8    if  $i \leq n$  then return( $i$ )     $\triangleright$  successful search
9      else return(0)     $\triangleright$  unsuccessful search
10   end LINEAR_SEARCH_2

```

Procedure 13.2 Linear search (improved implementation)

Analysis of linear search

Let us now analyze the performance of linear search. In an unsuccessful search, the number of key comparisons performed is clearly n (for version 1) and $n+1$ (for version 2). In a successful search, the minimum and maximum number of key comparisons performed are obviously 1 and n , respectively. What about the *average* number of comparisons in a successful search? Let C_n denote this quantity. Clearly, the value of C_n depends on the probability, say p_i , that k_i is the search key for $i = 1, 2, \dots, n$.

If each key in the table is equally likely to be a search key, i.e., $p_i = 1/n$, $i = 1, 2, \dots, n$, then the average number of key comparisons performed in a successful search is

$$C_n = \frac{1 + 2 + 3 + 4 + \dots + n}{n} = \frac{n+1}{2} \quad (13.1)$$

The numerator in Eq.(13.1) is simply the sum of the number of comparisons performed if the search key were k_1 (1 comparison), if the search key were k_2 (2 comparisons), and so on. This sum, divided by the number of search keys which result in a successful search, yields the average. On the average, then, we need to search half the table.

While the assumption of equal probabilities simplifies the analysis, a more realistic scenario is one in which certain keys are more likely than others to be the search key. In general, if p_1 is the probability that the search key is k_1 , p_2 is the probability that the search key is k_2 , and so on, where $p_1 + p_2 + \dots + p_n = 1$, then we have

$$C_n = 1p_1 + 2p_2 + 3p_3 + \dots + np_n \quad (13.2)$$

Note that Eq.(13.2) reduces to Eq.(13.1) if $p_1 = p_2 = \dots = p_n = 1/n$, i.e., if each of the n keys in the table is equally likely to be a search key.

It is also clear from Eq.(13.2) that C_n will be minimum if

$$p_1 \geq p_2 \geq p_3 \geq \dots \geq p_n \quad (13.3)$$

that is, if the records in the table are arranged from the most frequently accessed to the least frequently accessed. Following Knuth (KNUTH3[1998], pp. 399-401) let us now investigate what C_n is for certain probability distributions assuming that conditions (13.3) obtain.

1. Binary probability distribution — the i th record is accessed twice as often as the $(i + 1)$ th record

$$\begin{aligned} p_i &= \frac{1}{2^i} & 1 \leq i \leq n-1 \\ p_n &= \frac{1}{2^{n-1}} & (\text{so that } p_1 + p_2 + \dots + p_n = 1) \end{aligned} \quad (13.4)$$

Substituting into Eq.(13.2), we obtain

$$\begin{aligned} C_n &= 1 \cdot \frac{1}{2^1} + 2 \cdot \frac{1}{2^2} + 3 \cdot \frac{1}{2^3} + \dots + (n-1) \cdot \frac{1}{2^{n-1}} + n \cdot \frac{1}{2^{n-1}} \\ &= \sum_{i=1}^{n-1} \frac{i}{2^i} + \frac{n}{2^{n-1}} \\ &= \underbrace{2 - \frac{(n-1)+2}{2^{n-1}}}_{\text{cf. Eq.(2.32)}} + \frac{n}{2^{n-1}} \\ &= 2 - 2^{1-n} \end{aligned} \quad (13.5)$$

Hence, the average number of comparisons in a successful search is less than 2 whatever n is. Fine, but then this probability distribution is quite artificial and is unlikely to ever occur in practice. The next two are more realistic, being based on empirically observed behavior.

2. Zipf distribution (after G. K. Zipf, who observed that *the n th most common word in natural language text seems to occur with a frequency inversely proportional to n*)

$$p_i = \frac{1/H_n}{i} \quad 1 \leq i \leq n \quad (13.6)$$

where

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \text{\textit{n}th harmonic number of order 1}$$

Substitution into Eq.(13.3) yields

$$\begin{aligned} C_n &= 1 \cdot \frac{1}{H_n} + 2 \cdot \frac{1}{2H_n} + 3 \cdot \frac{1}{3H_n} + \dots + n \cdot \frac{1}{nH_n} \\ &= \frac{n}{H_n} \end{aligned} \quad (13.7)$$

For instance, $C_n = (3.41, 19.28, 133.59)$ for $n = (10, 100, 1000)$.

3. ‘80-20’ rule (commonly observed in commercial file-processing applications in which 80% of the transactions deal with the most active 20% of the records in a file, and the same rule applies to this 20%, so that 64% of the transactions deal with the most active 4% of the records, and so on). A probability distribution which approximates this rule is

$$p_i = \frac{1/H_n^{1-\theta}}{i^{1-\theta}} \quad 1 \leq i \leq n \quad (13.8)$$

where

$$H_n^{1-\theta} = \frac{1}{1^{1-\theta}} + \frac{1}{2^{1-\theta}} + \frac{1}{3^{1-\theta}} + \dots + \frac{1}{n^{1-\theta}} = n\text{th harmonic number of order } 1-\theta$$

$$\theta = \frac{\log_{10} 0.80}{\log_{10} 0.20} = 0.1386$$

Substitution into Eq.(13.3) yields

$$\begin{aligned} C_n &= 1 \cdot \frac{1}{1^{1-\theta} H_n^{1-\theta}} + 2 \cdot \frac{1}{2^{1-\theta} H_n^{1-\theta}} + 3 \cdot \frac{1}{3^{1-\theta} H_n^{1-\theta}} + \dots + n \cdot \frac{1}{n^{1-\theta} H_n^{1-\theta}} \\ &= \frac{1}{H_n^{1-\theta}} \left[\frac{1}{1^{1-\theta}} + \frac{2}{2^{1-\theta}} + \frac{3}{3^{1-\theta}} + \dots + \frac{n}{n^{1-\theta}} \right] \\ &= \frac{1}{H_n^{1-\theta}} \underbrace{\left[\frac{1}{1^{-\theta}} + \frac{1}{2^{-\theta}} + \frac{1}{3^{-\theta}} + \dots + \frac{1}{n^{-\theta}} \right]}_{H_n^{-\theta}} \\ &= \frac{H_n^{-\theta}}{H_n^{1-\theta}} \end{aligned} \quad (13.9)$$

For instance, $C_n = (3.70, 23.76, 188.43)$ for $n = (10, 100, 1000)$.

The interested student is referred to KNUTH3[1998], pp. 401–404 for the analysis of other techniques related to sequential search.

13.3.2 Binary search in an ordered static sequential table

When a table is small or if it is to be searched only a few times, linear search usually suffices. However, if a table is to be searched again and again (e.g., in Session 11 we discussed UPCAT processing, in which we needed to search the course table literally thousands of times), then it will be more efficient if we maintain the records (k_i, d_i) in T such that $k_1 < k_2 < k_3 < \dots < k_n$, yielding an **ordered table**. With linear search in an unordered table, the result of a comparison leaves us with two alternatives:

- (a) $K = k_i$ — stop, we already found what we want
- (b) $K \neq k_i$ — try the next record

432 SESSION 13. Sequential tables

In an ordered table, the result of a comparison leaves us with three alternatives:

- (a) $K = k_i$ — stop, we already found what we want
- (b) $K < k_i$ — continue the search, but only on the records with keys k_1, k_2, \dots, k_{i-1}
- (c) $K > k_i$ — continue the search, but only on the records with keys $k_{i+1}, k_{i+2}, \dots, k_n$

It is clear that if we properly choose k_i , each comparison performed will substantially reduce the amount of work yet to be done. We eliminate from further consideration not only one (as in linear search), but substantially more, records.

An obvious choice of k_i is the middle key. Then, if $K < k_i$, we continue the search on the upper half of the table, and if $K > k_i$, we continue the search on the lower half of the table, using the same technique iteratively. Thus, we discard half the remaining records after every comparison. This procedure is aptly called *binary search*. One other way to choose k_i is based on some simple properties of the Fibonacci numbers, yielding a procedure called *Fibonacci search*.

Let's consider binary search first. There are actually various ways of implementing the algorithm; KNUTH3[1998] presents no less than six variations on the theme! We will consider the first and last of these. Procedure `BINARY_SEARCH`, which we briefly encountered in Session 2, is by far the most familiar implementation of the method.

```

1  procedure BINARY_SEARCH( $T, K$ )
2     $l \leftarrow 1; \quad u \leftarrow n$ 
3    while  $l \leq u$  do
4       $i \leftarrow \lfloor (l + u)/2 \rfloor$ 
5      case
6        :  $K = \text{KEY}(i)$ : return( $i$ )       $\triangleright$  successful search
7        :  $K > \text{KEY}(i)$ :  $l \leftarrow i + 1$    $\triangleright$  discard upper half
8        :  $K < \text{KEY}(i)$ :  $u \leftarrow i - 1$    $\triangleright$  discard lower half
9      endcase
10   endwhile
11   return(0)                         $\triangleright$  unsuccessful search
12 end BINARY_SEARCH

```

Procedure 13.3 Binary search (conventional implementation)

The input to the procedure is an ordered table T represented by the data structure $\mathbb{T} = [\text{KEY}(1:m), \text{DATA}(1:m), n]$, and a search argument K . The procedure returns the index of the record in T whose key matches K if it exists, or zero if no such record exists. At any time during the search, the two pointers l and u demarcate the portion of the table still to be searched such that if K is in the table, then $k_l \leq K \leq k_u$. The condition $u < l$ indicates that K is not in the table.

Analysis of binary search

Let us now analyze the performance of binary search, as implemented by Procedure 13.3. It is clear that the minimum number of key comparisons in a successful search is 1; this occurs if K happens to match the middle key in the whole table. What about the maximum number of comparisons in a successful search? In an unsuccessful search? What is the average number of comparisons in a successful search? In an unsuccessful search? To answer these questions, let us construct the binary tree representation of binary search to expose the inner workings of the algorithm. For $n = 16$, this is depicted in Figure 13.5 below.

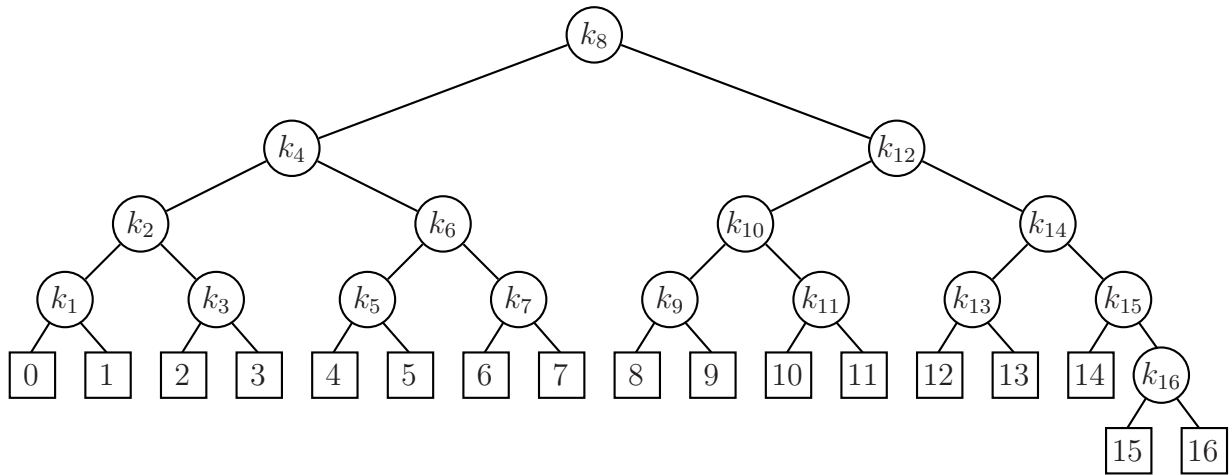


Figure 13.5 Binary tree representation of binary search for $n = 16$

We take note of the following observations pertaining to Figure 13.5.

1. The binary tree is an *extended binary tree* (see section 6.1); the circular nodes are called *internal nodes* and the square nodes, which represent null sons, are called *external nodes*.
2. The internal nodes contain the keys $k_i, i = 1, 2, \dots, n$; an inorder traversal of the binary tree yields an ascending order of the keys.
3. The external nodes are labeled 0 through n from left to right.
4. For $n = 16$, procedure `BINARY_SEARCH` initially computes the middle index i as $\lfloor (1 + 16)/2 \rfloor = 8$, thus the search key K is first compared with k_8 ; this is the key in the root node of the binary tree.
5. If $K < k_8$ the next comparison is with the key with index $\lfloor (1 + 7)/2 \rfloor = 4$; this is the key stored in the left son of the root node. On the other hand, if $K > k_8$, the next comparison is with the key with index $\lfloor (8 + 16)/2 \rfloor = 12$; this is the key stored in the right son of the root node.
6. Let node α be the node whose key is k_i . Then in general, if $K < k_i$, we next compare K with the left son of node α , and if $K > k_i$, we next compare K with the right son of node α .

7. A path in the binary tree represents a sequence of comparisons. A successful search terminates in an internal node; e.g., the path $k_8 - k_4 - k_6 - k_5$ represents the sequence of comparisons $K < k_8, K > k_4, K < k_6, K = k_5$. An unsuccessful search terminates in an external node, e.g., the path $k_8 - k_4 - k_2 - k_3 - 2$ represents the sequence of comparisons $K < k_8, K < k_4, K > k_2, K < k_3$, thus terminating in the external node labeled 2.

In Figure 13.5, the longest path terminating in an internal node is $k_8 - k_{12} - k_{14} - k_{15} - k_{16}$ (when $K = k_{16}$), and this represents the maximum number of comparisons in a successful search (which is 5). Similarly, the longest path terminating in an external node is $k_8 - k_{12} - k_{14} - k_{15} - k_{16} - 15$ (when $k_{15} < K < k_{16}$) or $k_8 - k_{12} - k_{14} - k_{15} - k_{16} - 16$ (when $K > k_{16}$), and this represents the maximum number of comparisons in an unsuccessful search (which is also 5). In general, the *maximum* number of comparisons performed by binary search on a table of size n in either a successful or unsuccessful search is $\lceil \log_2(n+1) \rceil$. [Verify].

To determine the *average* number of comparisons performed by binary search, we define two quantities called the **internal path length**, I , and the **external path length**, E , of an extended binary tree on n internal nodes, as follows:

$$I = \text{sum of the lengths of the paths from the root} \quad (13.10)$$

to each of the n internal nodes

$$E = \text{sum of the lengths of the paths from the root} \quad (13.11)$$

to each of the $n+1$ external nodes

The following simple relationship between I and E is readily verified:

$$E = I + 2n \quad (13.12)$$

For instance, for the binary tree of Figure 13.5, we have:

$$\begin{aligned} I &= 1(0) + 2(1) + 4(2) + 8(3) + 1(4) = 38 \\ E &= 15(4) + 2(5) = 70 \\ &= 38 + 2(16) = 70 \end{aligned}$$

If we add 1 to each of the numbers inside the pair of parentheses in the expression for I in the above example, i.e., $1(0+1) + 2(1+1) + 4(2+1) + 8(3+1) + 1(4+1)$, we obtain $54 = 38 + 16$, or $I + n$. This quantity represents the total number of comparisons performed in a sequence of n successful searches terminating in each of the n internal nodes. Assuming that each of the n keys in the table is equally likely to be a search key, then the average number of comparisons, C_n , performed by binary search on a table of size n in a successful search is simply

$$C_n = \frac{I + n}{n} \quad (13.13)$$

In a similar manner, the quantity E represents the total number of comparisons performed in a sequence of $n + 1$ unsuccessful searches terminating in each of the $n + 1$ intervals between keys (i.e., $K < k_1, k_1 < K < k_2, \dots, k_{15} < K < k_{16}, K > k_{16}$). Assuming that all of these intervals between keys are equally probable, then the average number of comparisons, C'_n , performed by binary search on a table of size n in an unsuccessful search is simply

$$C'_n = \frac{E}{n + 1} \quad (13.14)$$

In Eq.(13.13) and Eq.(13.14), the numerator is the total number of comparisons and the denominator is the total number of equally likely targets. Thus the average number of comparisons performed by binary search on a table of size $n = 16$ is $C_n = \frac{38+16}{16} = 3.375$ and $C'_n = \frac{70}{16+1} = 4.118$ for successful and unsuccessful searches, respectively. This is as best as it can get for searching based solely on comparing keys.

13.3.3 Multiplicative binary search

On some computers, division is a more expensive operation than multiplication. Thus an implementation of binary search which performs a multiplication in lieu of a division operation to find the middle key in that portion of the table still to be searched would run faster on such computers. The catch, however, is that we need to first rearrange the records in the table, as follows:

Algorithm REARRANGE_FOR_MBS

1. Assign the keys $k_1 < k_2 < k_3 < \dots < k_n$, to the nodes of a *complete* binary tree on n nodes in *inorder*.
2. Arrange the records in the table according to the sequence of the keys in the binary tree when enumerated in *level order*.

The figure below shows the rearranged table for $n = 16$.

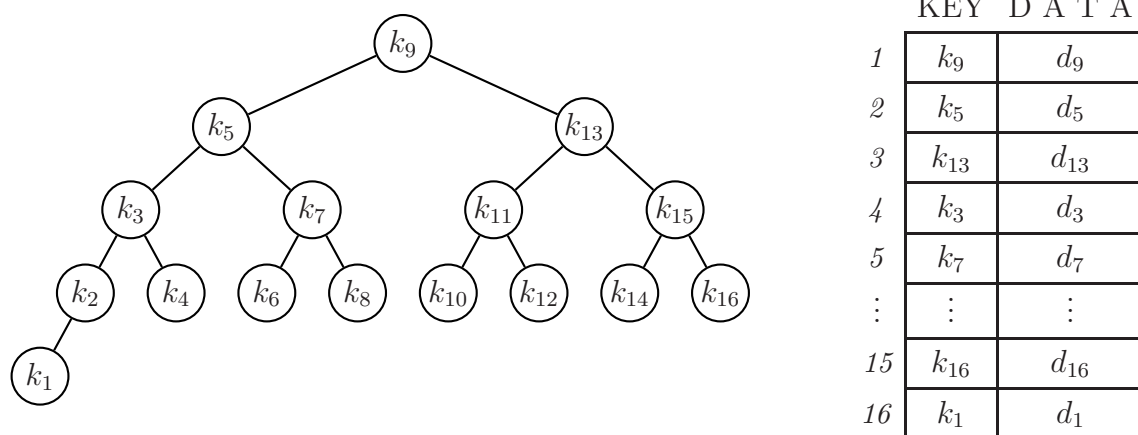


Figure 13.6 Rearranging the records from an increasing order of the keys into the order required by multiplicative binary search

Comparison begins with $KEY(i) = KEY(1) = k_9$, which is the middle key in the original table if we take the ceiling, instead of the floor, in calculating the middle index. If $K < k_9$ we next compare K with k_5 , which is the middle key in the upper half of the original table; in the reordered table, this key is found in cell $2 * i = 2 * 1 = 2$. If $K > k_9$ we next compare K with k_{13} , which is the middle key in the lower half of the original table; in the reordered table, this key is found in cell $2 * i + 1 = 2 * 1 + 1 = 3$. And so on. Procedure MULTIPLICATIVE_BINARY_SEARCH given below captures the whole process in a very neat way.

```

1  procedure MULTIPLICATIVE_BINARY_SEARCH(T, K)
2     $i \leftarrow 1$ 
3    while  $i \leq n$  do
4      case
5        : $K = KEY(i)$ : return( $i$ )           ▷ successful search
6        : $K < KEY(i)$ :  $i \leftarrow 2 * i$      ▷ go left
7        : $K > KEY(i)$ :  $i \leftarrow 2 * i + 1$  ▷ go right
8      endcase
9    endwhile
10   return(0)                             ▷ unsuccessful search
11 end BINARY_SEARCH

```

Procedure 13.4 Binary search (multiplicative version)

Generating the reordered table for multiplicative binary search

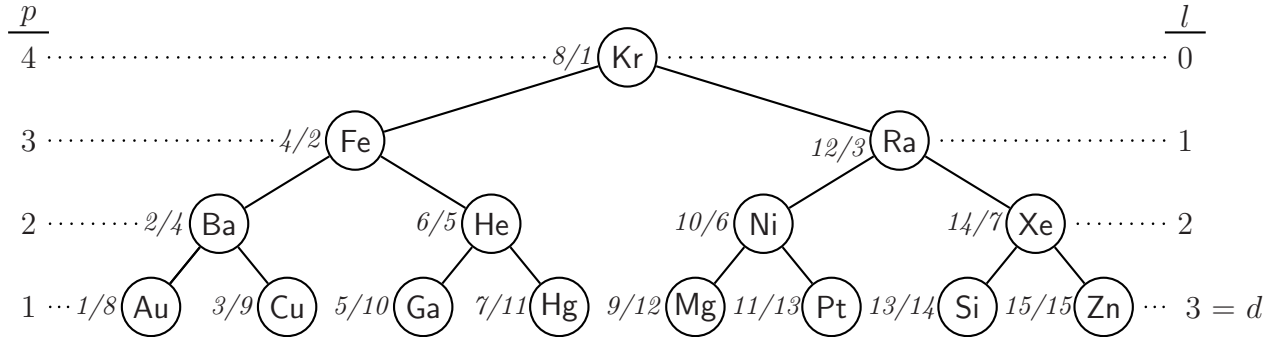
As with many of the algorithms we have encountered so far, algorithm REARRANGE_FOR_MBS is readily applied ‘by hand’. The challenge is to implement the algorithm on a computer such that the records in the original ordered table T are rearranged *in place* and in *linear time* in the number of records n . You may want to think about the problem and take up the challenge before reading on.

Given an ordered sequential table with records stored in ascending sequence of the keys $k_1 < k_2 < k_3 < \dots < k_n$, we will now devise a linear time procedure to rearrange the records in-place in the order required for MBS. We consider first the case in which $n = 2^m - 1$ for some positive integer m , which means that the binary tree is *full*. To this end, consider the table T shown in Figure 13.7(a), which contains the names of elements in the periodic table as alphabetic keys and the corresponding atomic weight as the satellite data. Figure 13.7(b) shows the keys assigned to the nodes of a complete, and in this case also full, binary tree in inorder. The numbers i/j placed alongside each node represent the index of the cell in T where k_i is stored, which is index i , and the index j of the cell in the reordered table T' where k_i is moved to. Note that T and T' are *not* separate tables; the records are simply rearranged within the table T to yield T' . For instance, the record (Au, 79) in cell 1 of T is moved to cell 8; (Kr, 36) in cell 8 is moved to cell 1; (Ba, 56) in cell 2 is moved to cell 4; and so on.

i	KEY	DATA
1	Au	79
2	Ba	56
3	Cu	29
4	Fe	26
5	Ga	31
6	He	2
7	Hg	80
8	Kr	36
9	Mg	12
10	Ni	28
11	Pt	78
12	Ra	88
13	Si	14
14	Xe	54
15	Zn	30

(a) The original table, T

j	KEY	DATA
1	Kr	36
2	Fe	26
3	Ra	88
4	Ba	56
5	He	2
6	Ni	28
7	Xe	54
8	Au	79
9	Cu	29
10	Ga	31
11	Hg	80
12	Mg	12
13	Pt	78
14	Si	14
15	Zn	30

(c) The reordered table, T' (b) A complete binary tree B with the keys assigned to the nodes in inorder**Figure 13.7** Generating the reordered table for MBS (binary tree is full)

At the heart of the algorithm to rearrange T into T' is a function which maps the address (index) i in T of a record to its new address j in T' . To derive this function let:

- k_i be the key to be moved from its old to its new position in the table
- i be the index of the cell in T where k_i resides
- j be the index of the cell in T' where k_i will reside
- p be the number of times i is divided by 2 before a nonzero remainder results (or equivalently, p is the *smallest* integer such that 2^p is *not* a divisor of i)
- l be the level of the node in B which contains k_i
- d be the level of the bottommost node(s) in B , i.e., $d = \lfloor \log_2 n \rfloor$

Then we have:

$$l = (d + 1) - p \quad (13.15)$$

$$j = 2^l + \left\lfloor \frac{i}{2^p} \right\rfloor \quad (13.16)$$

Figure 13.8 shows the values of the above quantities for the binary tree of Figure 13.7(b) with the keys enumerated in level order. It is clear that Eq.(13.15) and Eq.(13.16) produce the desired mapping from i to j when the binary tree is full.

k_i	i	p	l	$2^l + \left\lfloor \frac{i}{2^p} \right\rfloor$	j
Kr	8	4	0	1 + 0	1
Fe	4	3	1	2 + 0	2
Ra	12	3	1	2 + 1	3
Ba	2	2	2	4 + 0	4
He	6	2	2	4 + 1	5
Ni	10	2	2	4 + 2	6
Xe	14	2	2	4 + 3	7
Au	1	1	3	8 + 0	8
Cu	3	1	3	8 + 1	9
Ga	5	1	3	8 + 2	10
Hg	7	1	3	8 + 3	11
Mg	9	1	3	8 + 4	12
Pt	11	1	3	8 + 5	13
Si	13	1	3	8 + 6	14
Zn	15	1	3	8 + 7	15

Figure 13.8 Applying Eq.(13.16) to the table of Figure 13.7

When the binary tree is complete but not full, Eq.(13.16) no longer applies to some of the nodes in the tree. Figure 13.9 shows T , B and T' for the case in which $n = 10$. You may readily verify that Eq.(13.16) does not apply anymore to the black nodes with indices $s < i \leq n$, where

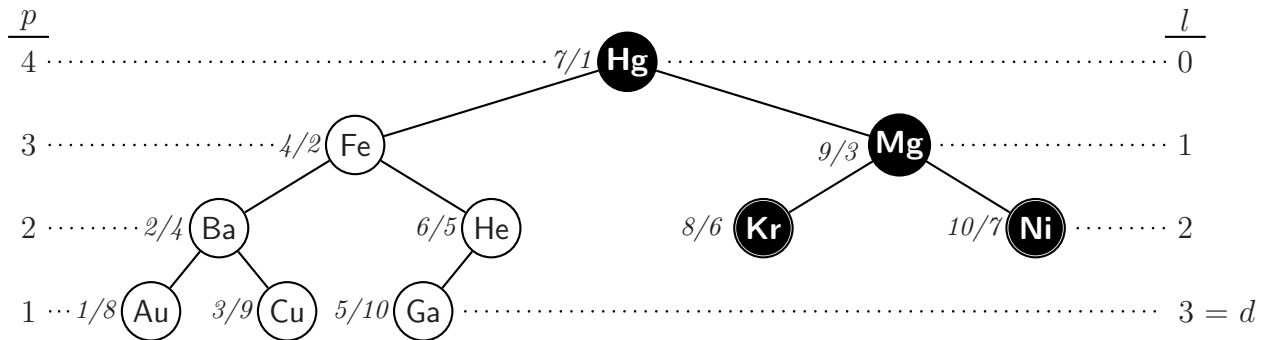
$$s = 2(n + 1) - 2^{d+1}. \quad (13.17)$$

To make the equation applicable to the black nodes as well, we simply replace i in Eq.(13.16) by $2i - s$, which would have been the value of i if the binary tree were full.

i	KEY	DATA
1	Au	79
2	Ba	56
3	Cu	29
4	Fe	26
5	Ga	31
6	He	2
7	Hg	80
8	Kr	36
9	Mg	12
10	Ni	28

(a) The original table, T

j	KEY	DATA
1	Hg	80
2	Fe	26
3	Mg	12
4	Ba	56
5	He	2
6	Kr	36
7	Ni	28
8	Au	79
9	Cu	29
10	Ga	31

(c) The reordered table, T' (b) A complete binary tree B with the keys assigned to the nodes in inorder**Figure 13.9** Generating the reordered table for MBS (binary tree is not full)

For the table given above we find that $s = 2(n+1) - 2^{d+1} = 2(10+1) - 2^{3+1} = 6$. This means that each of the records in cells $i = 1, 2, 3, 4, 5, 6$ of T is moved to its new address j in T' as prescribed by Eq.(13.16), as you may readily verify. For the records in cells $i = 7, 8, 9, 10$ of T , the value of i must be replaced by $2i - s$ before Eq.(13.16) is applied. For instance, for the record (**Hg**, 80) in cell 7 of T we have $i = 2 * i - s = 2 * 7 - 6 = 8$; substituting $i = 8$ in Eq.(13.16) gives $j = 1$ which is where (**Hg**, 80) is in T' . You may easily verify that similar computations correctly find the new address in T' of each of the remaining records in T .

Procedure REARRANGE_FOR_MBS rearranges *in-situ* the records in a sequential table T from an initially ascending sequence of the keys into the order required by multiplicative binary search as described above. The input to the procedure is the table T represented by the data structure $\mathbb{T} = [KEY(1:m), DATA(1:m), n]$; for brevity, we will use the notation $R(i)$ to represent the record $(KEY(i), DATA(i))$. The output consists of the same table T with the records rearranged for MBS. The procedure calls on function NEW_INDEX which accepts the current address of a record and returns the new address of the record in the reordered table.

```

1  procedure REARRANGE_FOR_MBS(T)
2  array moved(1:n)
3  moved  $\leftarrow$  0
4  d  $\leftarrow$   $\lfloor \log_2 n \rfloor$ 
5  s  $\leftarrow$   $2 * (n + 1) - 2^{(d + 1)}$ 
   for i  $\leftarrow$  1 to n do
6     if moved(i) = 0 then [ record_out  $\leftarrow$  R(i)
7                               j  $\leftarrow$  i
8                               loop
9                                   moved(j)  $\leftarrow$  1
10                                  j  $\leftarrow$  NEW_INDEX(j, d, s)
11                                  record_in  $\leftarrow$  record_out
12                                  record_out  $\leftarrow$  R(j)
13                                  R(j)  $\leftarrow$  record_in
14                                  if moved(j) = 1 then exit
15                               forever ]
16 endfor
17 end REARRANGE_FOR_MBS

```

```

1  procedure NEW_INDEX(i, d, s)
2  if i > s then i  $\leftarrow$   $2 * i - s$ 
3  p  $\leftarrow$  1
4  q  $\leftarrow$  i
5  loop
6      qdiv2  $\leftarrow$   $\lfloor q/2 \rfloor$ 
7      r  $\leftarrow$  q -  $2 * qdiv2$ 
8      if r  $\neq$  0 then return ( $2^{(d + 1 - p)} + \lfloor i/2^p \rfloor$ )
9      else [p  $\leftarrow$  p + 1; q  $\leftarrow$  qdiv2 ]
10 forever
11 end NEW_INDEX

```

Procedure 13.5 Rearranging an ordered table for multiplicative binary search

If you ‘walk through’ the above procedure you will note that rearranging the records in *T* proceeds in ‘cycles’, where a cycle completes when the first vacated cell is occupied by the last displaced record. For instance, rearranging the records in the table of Figure 13.9(a) takes three cycles:

1. (Au,79) from cell 1 moves to cell 8; (Kr,36) from cell 8 moves to cell 6; (He,2) from cell 6 moves to cell 5; (Ga,31) from cell 5 moves to cell 10; (Ni,28) from cell 10 moves to cell 7; (Hg,80) from cell 7 moves to cell 1; cycle is complete.
2. (Ba,56) from cell 2 moves to cell 4; (Fe,26) from cell 4 moves to cell 2; cycle is complete.
3. (Cu,29) from cell 3 moves to cell 9; (Mg,12) from cell 9 moves to cell 3; cycle is complete.

The rearrangement is now finished giving the table in Figure 13.9(c). The function of the *moved* array in procedure REARRANGE_FOR_MBS is to avoid going through a cycle in endless cycles. A record that is already in its new position in the rearranged table should not be moved out once again.

Analysis of procedure REARRANGE_FOR_MBS

Each of the n records in the table is moved exactly once from its original position in cell i to its new position in cell j of the table. Each such move requires evaluating Eq.(13.16) to find j given i . Evaluating Eq.(13.16) requires finding p , where p is the number of times i is divided by 2 before a non-zero remainder results. Hence to determine the time complexity of procedure REARRANGE_FOR_MBS, we find the total number, say S , of such divisions performed. For the full binary tree of Figure 13.7 we find that 2^3 values of i (viz., 1, 3, 5, 7, 9, 11, 13 and 15) require 1 division by 2 before a nonzero remainder results, 2^2 values of i (viz., 2, 6, 10 and 14) require 2, 2^1 (viz., 4 and 12) require 3, and 2^0 value of i (viz., 8) requires 4. Hence we have

$$S = 1 \cdot 2^3 + 2 \cdot 2^2 + 3 \cdot 2^1 + 4 \cdot 2^0$$

In general, we have

$$S = \sum_{i=1}^k i \cdot 2^{k-i} = \sum_{i=1}^k i \cdot \frac{2^k}{2^i} = 2^k \sum_{i=1}^k \frac{i}{2^i} = 2^k \underbrace{\left[2 - \frac{k+2}{2^k} \right]}_{\text{cf. Eq.(2.32)}} < 2^{k+1} \quad (13.18)$$

where $k = d + 1 = \lfloor \log_2 n \rfloor + 1$. Since for any binary tree on n nodes we have $2^d \leq n$ (the equality obtains when there is only one node at the bottommost level), it follows that $2^{k+1} = 2^{d+2} \leq 4n$. Hence $S < 4n$. Therefore, procedure REARRANGE_FOR_MBS executes in linear time in the number of records rearranged.

13.3.4 Fibonacci search in an ordered sequential table

We pointed out earlier that in searching an ordered sequential table, an appropriate choice of the key k_i with which to compare the search key K can lead to a substantial reduction in the amount of work still to be done. In binary search, we choose k_i to be the middle key in the current search range, and in the event that $K \neq k_i$, we discard the upper half (if $K > k_i$) or the lower half (if $K < k_i$), of the current search area.

Fibonacci search uses some simple properties of the Fibonacci numbers F_k (see section 2.1.6) to characterize the search. Specifically, we initially choose k_i such that $i = F_k$ where F_k is the largest Fibonacci number less than or equal to the table size, n . Subsequently, we discard all the keys with indices less than i if $K > k_i$, or all the keys with indices greater than i if $K < k_i$. The next value of i is computed using two auxiliary variables, say p and q , which are consecutive Fibonacci numbers.

442 SESSION 13. Sequential tables

Assume, for the moment, that we have a table of size $n = F_{k+1} - 1$, i.e., $n + 1$ is a Fibonacci number, say $n = 12$. Then, i , p and q will initially be as indicated below:

				q	p	i						
				↓	↓	↓						
0	1	1	2	3	5	8	13	21	34	55	...	
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	...	

Note that $i = 8$ is the largest Fibonacci number less than or equal to $n = 12$. We now compare K with k_i or k_8 . If $K = k_i$, the search is done. Otherwise:

(a) if $K < k_i$, then we set $i \leftarrow i - q$ and shift p and q one place left, as shown

				q	p							
				↓	↓							
0	1	1	2	3	5	8	13	21	34	55	...	
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	...	

(b) if $K > k_i$, then we set $i \leftarrow i + q$ and shift p and q two places left, as shown

				q	p							
				↓	↓							
0	1	1	2	3	5	8	13	21	34	55	...	
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	...	

The search terminates unsuccessfully if for some i , $K < k_i$ and $q = 0$ or $K > k_i$ and $p = 1$. Procedure FIBONACCIAN_SEARCH formalizes the process described above.

```

1  procedure FIBONACCIAN_SEARCH( $\mathbb{T}$ ,  $K$ )
2   $i \leftarrow F_k$ ;  $p \leftarrow F_{k-1}$ ;  $q \leftarrow F_{k-2}$ 
3  loop
4    case
5      : $K = KEY(i)$ : return( $i$ )
6      : $K < KEY(i)$ : [ if  $q = 0$  then return(0)
7                    else [ $i \leftarrow i - q$ ;  $t \leftarrow p$ ;  $p \leftarrow q$ ;  $q \leftarrow t - q$ ] ]
8      : $K > KEY(i)$ : [ if  $p = 1$  then return(0)
9                    else [ $i \leftarrow i + q$ ;  $p \leftarrow p - q$ ;  $q \leftarrow q - p$ ] ]
10   endcase
11 forever
12 end FIBONACCIAN_SEARCH

```

Procedure 13.6 Fibonacci search

In the initializations in line 2, F_k is the largest Fibonacci number less than or equal to n where $n + 1$ is a Fibonacci number. These initializations may be done through table look-up, function call, etc.

To fully understand how the procedure works, let us construct the binary tree representation of Fibonacci search; call this a **Fibonacci tree**. Fibonacci trees are classified by *order*, such that the Fibonacci tree of order k has $F_{k+1} - 1$ internal nodes, or equivalently, F_{k+1} external nodes. A Fibonacci tree is constructed as follows:

- (a) If $k = 0$ or $k = 1$, the tree is simply $\boxed{0}$.
- (b) If $k \geq 2$, then the root is F_k ; the left subtree is the Fibonacci tree of order $k - 1$; and the right subtree is the Fibonacci tree of order $k - 2$ with all node labels increased by F_k .

Figure 13.10 shows the Fibonacci tree of order 6, constructed by applying the above rules.

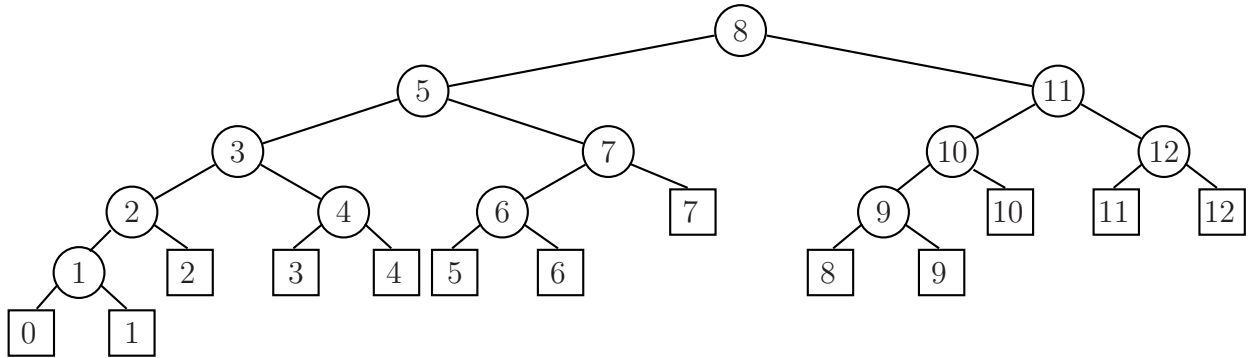


Figure 13.10 The Fibonacci tree of order 6

As with the binary tree representation of binary search, a path in a Fibonacci tree which terminates in an internal node represents a sequence of comparisons in a successful search, and a path which terminates in an external node represents a sequence of comparisons in an unsuccessful search. Comparison starts with the key at the root, and the search continues on the left or the right subtree, depending on whether K is less than or greater than this key. The auxiliary variables p and q are all that we need to locate the left son or the right son, as needed, as the index i traces a path in the tree. With Figure 13.10 as an aid, go through the actions of procedure FIBONACCIAN_SEARCH. Try both successful and unsuccessful searches, and verify the termination conditions for the latter case.

In procedure FIBONACCIAN_SEARCH given above, it is assumed that F_k is the largest Fibonacci number less than or equal to the table size n , and that $n + 1$ is the Fibonacci number F_{k+1} . For this value of n , all possible successful and unsuccessful searches are represented by some path in the corresponding Fibonacci tree, and the procedure correctly traces all such possible paths. However, if $F_k < n < F_{k+1} - 1$ and K is greater than the key at index F_k , the procedure may search beyond the table following a non-existent path, yielding spurious results. So that the procedure works for all $n \geq 1$, the initialization part should be modified as follows:

2 $i \leftarrow F_k; p \leftarrow F_{k-1}; q \leftarrow F_{k-2}$
 2a $m \leftarrow F_{k+1} - 1 - n; \text{ if } K > k_i \text{ then } i \leftarrow i - m$

Incorporate line 2a into procedure FIBONACCIAN_SEARCH so that it becomes applicable for any ordered table of size $n \geq 1$. Now assume that $n + 1$ is not a Fibonacci number, and go through the actions of the procedure for some $K > k_i$. Better yet, transcribe the procedure into a running program (with a trace of the key comparisons performed) and run it for different values of K for different values of n .

Summary

- If we have prior information as to which records in a table are more likely to be searched for, linear search attains its best average performance in successful searches if we arrange the records from the most frequently to the least frequently accessed. Equations 13.7 and 13.9 give a measure of such performance for two empirically observed probability distributions. In the absence of such prior information, we simply assume that each record in the table is equally likely to be chosen. In this case it no longer matters how the records are arranged in the table; on the average, linear search will look at half the records in a successful search.
- Binary search and Fibonacci search require that the table be ordered. Generating an ordered table, say by sorting, requires additional computational effort. Unless the table is to be searched repeatedly, keeping it ordered may not be worth the cost.
- When the table size is small, linear search may outperform binary search because program overhead in the former is minimal. However, on larger tables, and assuming that each key in the table is equally likely to be a search key, binary search performs best on the average among all comparison-based search methods. This is readily inferred from the binary tree representation of binary search in which external nodes appear on at most two adjacent levels, thus yielding the minimum C_n and C'_n .

Exercises

1. Assume that the records in a sequential table are accessed according to the ‘80-20’ rule and that they are ordered from the most frequently to the least frequently accessed. What is the average number of comparisons performed by linear search if there are

(a) 50 records
(b) 500 records
(c) 5000 records

 in the table?
2. Show the binary tree representation of binary search on an ordered table of size $n = 20$.
3. Prove that the *maximum* number of comparisons performed by binary search on an ordered table of size n in either a successful or an unsuccessful search is $\lceil \log_2(n+1) \rceil$.
4. What is the *average* number of comparisons performed by binary search on an ordered table of size $n = 20$ in a successful search? In an unsuccessful search? What conditions have been tacitly assumed in arriving at these numbers?
5. Using a language of your choice, transcribe procedure `BINARY_SEARCH` (Procedure 13.3) into a running program.

6. Knuth writes: ‘Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried.’ (KNUTH3[1998], p. 410). Using your program in Item 5, investigate the effect of the following changes to procedure `BINARY_SEARCH` (*ibid*, p. 423):
 - (a) change line 7 to: $l \leftarrow i$
 - (b) change line 8 to: $u \leftarrow i$
 - (c) change lines 7 and 8 to: $l \leftarrow i$ and $u \leftarrow i$, respectively
7. Suppose that the variables l and u are initialized as

$$l \leftarrow 0; u \leftarrow n + 1$$
 in line 2 of procedure `BINARY_SEARCH`. How should the test for termination in line 3 be modified and how should l and u be updated in lines 7 and 8 so that the procedure executes correctly?
8. Twelve of the 16 known moons of Jupiter are IO, EUROPA, GANYMEDE, CALLISTO, LEDA, HIMALIA, LYSITHEA, ELARA, ANANKE, CARME, PASIPHAE and SINOPE. Taking these names as alphabetic keys, arrange the names in a sequential table of size 12 in the order required by multiplicative binary search.
9. Using a language of your choice, transcribe procedures `REARRANGE_FOR_MBS` and `NEW_INDEX` (Procedure 13.5) into a running program. Test your program using the tables in Figure 13.7(a) and Figure 13.8(a). Using your program, solve Item 8 anew.
10. Show the Fibonacci tree representation of Fibonaccian search on an ordered table of size $n = 20$.
11. Using a language of your choice, transcribe procedure `FIBONACCIAN_SEARCH` (Procedure 13.6) into a running program. Test your program on an ordered table of size n such that $n + 1$ is a Fibonacci number, for instance, a table with the 12 moons of Jupiter in Item 8 as keys. Verify that it yields the correct results for both successful and unsuccessful searches.
12. Run your program in Item 11 on an ordered table of size n where $n + 1$ is *not* a Fibonacci number, for instance, the table in Figure 13.7(a). Using each of the keys in the table as the search key, is the search always successful?
13. Modify your program to incorporate line 2a. Redo Item 12 on your modified program.

Bibliographic Notes

Linear search and binary search are well known algorithms and can be found in most books on Data Structures or on Algorithms. The analysis of linear search for certain probability distributions (Zift, ‘80-20’, etc.) is from KNUTH3[1998], pp. 399–401. The

analysis of binary search which is based on the binary tree representation of the algorithm is from the same reference (pp. 411–414) where other implementations of the algorithm are also given and analyzed in detail (pp. 414–417). Procedure `BINARY_SEARCH` is an implementation of Algorithm B (*Binary search*) given in KNUTH3[1998], p. 410.

The multiplicative version of binary search is discussed in STANDISH[1980], pp. 138–139. Procedure `MULTIPLICATIVE_BINARY_SEARCH` is an implementation of Algorithm 4.3 (*Multiplicative binary search*) given in this reference, p. 139.

Fibonacci search is discussed in KNUTH3[1998], pp. 417–419 and also in STANDISH[1980], pp. 139–141. Procedure `FIBONACCIAN_SEARCH` is an implementation of Algorithm F (*Fibonacci search*) given in the former, p. 418.

SESSION 14

Binary search trees

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Define a binary search tree and differentiate between the static and dynamic type.
2. Explain how the basic operations of searching, insertion and deletion are performed on a BST.
3. Give an analysis of BST search for the minimal, maximal and average cases.
4. Define an AVL tree and explain why it is suitable to use in a dynamic environment.
5. Illustrate the different types of rotations on AVL trees.
6. Construct an AVL tree on a given set of keys.
7. Define an optimum BST and explain how the dynamic programming technique is applied to find one.
8. Construct an optimum BST on a given set of keys.

READINGS KNUTH3[1998], pp. 426–442, 458–471; STANDISH[1994], pp. 365–388; CORMEN[2001], pp. 253–264; WEISS[1997], pp. 100–123, 387–390.

DISCUSSION

In the preceding session, we studied two search algorithms which can be modeled by a binary tree, viz., binary search and Fibonaccian search. The binary tree representations of these search methods, Figures 13.5 and 13.8 of Session 13, are actually special cases of an ADT called a **binary search tree**. Our implementations of binary search and Fibonaccian search, procedure `BINARY_SEARCH` and procedure `FIBONACCIAN_SEARCH`, are for a static, sequentially allocated table. Thus the BST's which model these procedures are static: no insertions nor deletions of records, searches only. Essentially, we have here a case in which we have all the records on hand and the luxury of arranging these in a table in the required order, before we perform any searching.

In this session, we will consider the case in which the table is dynamic, with insertions and deletions comingling with searches. A dynamic binary search tree provides a way of efficiently implementing such a table.

14.1 Binary search trees

DEFINITION 14.1. A **binary search tree** is a binary tree with keys assigned to its nodes such that if k is the key at some node, say node α , then all the keys in the left subtree of node α are less than k and all the keys in the right subtree of node α are greater than k .

Figure 14.1 shows a binary search tree in which names of Old Testament prophets are the (alphabetic) keys. Note that if we take *any* node, say $\llbracket \text{HABAKKUK} \rrbracket$, then all the names in the left subtree of $\llbracket \text{HABAKKUK} \rrbracket$ are alphabetically less than HABAKKUK and all the names in the right subtree of $\llbracket \text{HABAKKUK} \rrbracket$ are alphabetically greater than HABAKKUK.

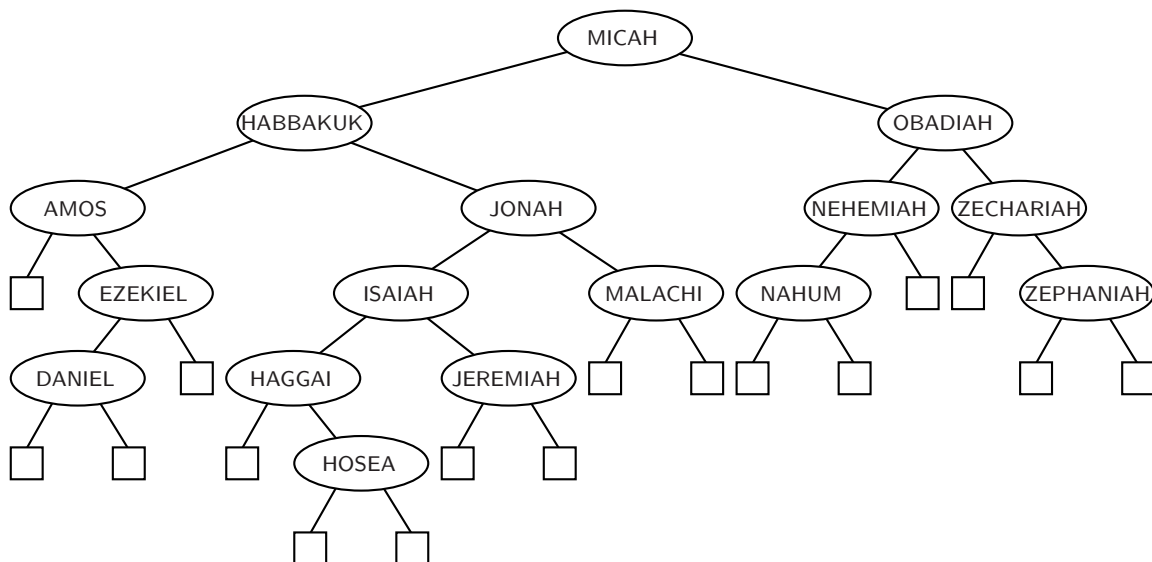


Figure 14.1 A binary search tree

For now let us take note of the following facts about the BST of Figure 14.1, or about any BST in general.

1. An inorder traversal of a BST yields an ascending sequence of its keys. For instance, traversing the BST shown above in inorder we obtain AMOS, DANIEL, EZEKIEL, HABAKKUK, ..., OBADIAH, ZECHARIAH, ZEPHANIAH.
2. It follows from 1 that the leftmost node in a BST contains the smallest key, and the rightmost node contains the largest key. Likewise, given any node, say node α with key k , the inorder predecessor of node α contains the next smaller key and the inorder successor of node α contains the next larger key.
3. A path in a BST which starts at the root and terminates in an internal node represents a successful search. A path in a BST which starts at the root and terminates in an external node represents an unsuccessful search.

For instance, to search for ISAIAH in the BST of Figure 14.1, we start by comparing ISAIAH with MICAH. Since $\text{ISAIAH} < \text{MICAH}$, we continue the search in the left

subtree of $\llbracket \text{MICAH} \rrbracket$. Next we compare ISAIAH with HABAKKUK . Since $\text{ISAIAH} > \text{HABAKKUK}$, we continue the search in the right subtree of $\llbracket \text{HABAKKUK} \rrbracket$. Now we compare ISAIAH with JONAH . Since $\text{ISAIAH} < \text{JONAH}$, we ‘go left’ to the subtree rooted at $\llbracket \text{ISAIAH} \rrbracket$. One final comparison yields $\text{ISAIAH} = \text{ISAIAH}$, and the search is done. In an actual application, we now access the data associated with the key ISAIAH .

A search for JOEL yields the sequence of key comparisons $\text{JOEL} < \text{MICAH}$ (go left), $\text{JOEL} > \text{HABAKKUK}$ (go right), $\text{JOEL} < \text{JONAH}$ (go left), $\text{JOEL} > \text{ISAIAH}$ (go right), $\text{JOEL} > \text{JEREMIAH}$ (go right); since the right subtree of $\llbracket \text{JEREMIAH} \rrbracket$ is null (i.e., an external node), the search is unsuccessful. Note that if we are inserting a node for JOEL into the BST, we follow the same search sequence from MICAH to JEREMIAH ; then we attach $\llbracket \text{JOEL} \rrbracket$ as the right son of $\llbracket \text{JEREMIAH} \rrbracket$. Thus an insert operation is always preceded by an unsuccessful search operation.

4. A BST on n internal nodes has $n + 1$ external nodes. To see this, start with a BST with $n = 1$ internal node and $n_{\text{ext}} = n + 1 = 2$ external nodes. Subsequently, each time an internal node is inserted into the BST the number of external nodes decreases by one (because the newly inserted internal node replaces an external node) and then increases by two (representing the null left and right sons of the newly inserted internal node), or a net increase of one external node for each new internal node. Thus the initial relation $n_{\text{ext}} = n + 1$ is maintained.
5. The height of a BST is defined as the length of the longest path from the root to an external node. The *minimum* height of a BST on n internal nodes is $\lfloor \log_2(n + 1) \rfloor$; this is attained in a BST in which each level has its full complement of internal nodes (1, 2, 4, 8, 16, ...), except possibly the bottommost level. The *maximum* height of a BST on n internal nodes is n ; this is attained in a BST in which each level has only one internal node.
6. There are b_n distinct BST’s on n distinct keys, where

$$b_n = \frac{1}{n+1} \binom{2n}{n} = \frac{1}{n+1} \frac{2n(2n-1)(2n-2) \cdots (n+1)}{n(n-1)(n-2) \cdots 1} \quad (14.1)$$

is the number of distinct binary trees on n unlabeled nodes (see Eq.(6.3) and Figure 6.3 in Session 6). To see this, imagine constructing the b_n binary trees and then assigning to each the keys $k_1 < k_2 < k_3 < \dots < k_n$ in inorder.

7. There are $n!$ ways in which n distinct keys can be inserted into an initially empty BST. Since $n! > b_n$ for $n > 2$, two or more orderings of the n keys may yield the same BST. For instance we can insert the keys $k_1 < k_2 < k_3$ into an initially empty BST in six different ways, viz., k_1, k_2, k_3 ; k_1, k_3, k_2 ; k_2, k_1, k_3 ; k_2, k_3, k_1 ; k_3, k_1, k_2 and k_3, k_2, k_1 . The sequences k_2, k_1, k_3 and k_2, k_3, k_1 yield the same BST and we obtain $b_3 = 5$ distinct BST’s.

14.1.1 Implementing some basic operations for dynamic tables

In the previous session we listed some basic operations performed on a table, to wit: (a) *search* for a record in the table whose key matches a given search key, (b) *insert* a new record into the table, (c) *delete* a record from the table, (d) *find* the record with the smallest or largest key, and (e) given some key k , *find* the record with the next smaller or next larger key. The following EASY procedures implement these operations on a dynamic table with the table represented as a binary search tree. Since the BST is dynamic, changing both in size and shape, we will use the linked representation of a binary tree. Each node of the binary tree corresponds to a record, and will be represented using the node structure

<i>LSON</i>	<i>KEY</i>	<i>DATA</i>	<i>RSON</i>
	k	d	

where d denotes the aggregation of data associated with the key k , or it denotes a pointer to some structure which contains this data. We assume that the BST is represented using the data structure

$$\mathbb{B} = [T, (LSON, KEY, DATA, RSON)]$$

The first component of \mathbb{B} contains the address of the root node of the BST; the condition $T = \Lambda$ means the BST is empty.

Searching for the node (record) whose key matches a given search key K

```

1  procedure BST_SEARCH( $\mathbb{B}, K$ )
  ▷ Given a BST  $\mathbb{B}$  and a search argument  $K$ , procedure searches for the node in  $\mathbb{B}$ 
  ▷ whose key matches  $K$ . If found, procedure returns the address of the node found;
  ▷ otherwise, it returns  $\Lambda$ .
2     $\alpha \leftarrow T$ 
3    while  $\alpha \neq \Lambda$  do
4      case
5        : $K = KEY(\alpha)$ : return( $\alpha$ )           ▷ successful search
6        : $K < KEY(\alpha)$ :  $\alpha \leftarrow LSON(\alpha)$    ▷ go left
7        : $K > KEY(\alpha)$ :  $\alpha \leftarrow RSON(\alpha)$    ▷ go right
8      endcase
9    endwhile
10   return( $\alpha$ )           ▷ unsuccessful search
11 end BST_SEARCH

```

Procedure 14.1 Searching in a binary search tree

We have seen that a path in a BST represents a sequence of comparisons in a search. Since the length of a path is clearly $O(h)$, procedure `BST_SEARCH` executes in $O(h)$ time on a BST of height h .

Searching for the node (record) with the minimum or maximum key

```

1  procedure BST_MINKEY( $\mathbb{B}$ )
  ▷ Given a BST  $\mathbb{B}$ , procedure searches for the node in  $\mathbb{B}$  with the smallest
  ▷ key and returns the address of the node found.
2     $\beta \leftarrow \alpha \leftarrow T$ 
3    while  $\alpha \neq \Lambda$  do
4       $\beta \leftarrow \alpha$ 
5       $\alpha \leftarrow LSON(\alpha)$ 
6    endwhile
7    return( $\beta$ )
8  end BST_MINKEY

```

Procedure 14.2 Finding the record with smallest key

Procedure finds the leftmost node in the BST and returns a pointer to it. Procedure BST_MAXKEY is symmetric to procedure BST_MINKEY; we simply replace line 5 with $\alpha \leftarrow RSON(\alpha)$. Both procedures execute in $O(h)$ time on a BST of height h .

Inserting a new record (k, d) into a BST

```

1  procedure BST_INSERT( $\mathbb{B}, k, d$ )
  ▷ Given a BST  $\mathbb{B}$  and a record  $(k, d)$ , procedure inserts a new node into the BST
  ▷ to store the record. If there is already a record in  $\mathbb{B}$  with the same key, procedure
  ▷ issues an error message and terminates execution.
2     $\alpha \leftarrow T$ 
3    while  $\alpha \neq \Lambda$  do
4      case
5        :  $k = KEY(\alpha)$ : [output ‘Duplicate key found.’; stop]
6        :  $k < KEY(\alpha)$ : [ $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow LSON(\alpha)$ ]
7        :  $k > KEY(\alpha)$ : [ $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow RSON(\alpha)$ ]
8      endcase
9    endwhile
  ▷ Exit from the loop means unsuccessful search; insert new node where unsuccessful
  ▷ search ended.
10   call GETNODE( $\tau$ )
11    $KEY(\tau) \leftarrow k$ ;  $DATA(\tau) \leftarrow d$ 
12   case
13     :  $T = \Lambda$  :  $T \leftarrow \tau$ 
14     :  $k < KEY(\beta)$ :  $LSON(\beta) \leftarrow \tau$ 
15     :  $k > KEY(\beta)$ :  $RSON(\beta) \leftarrow \tau$ 
16   endcase
17   return
18 end BST_INSERT

```

Procedure 14.3 Inserting a new record into a binary search tree

The action taken in line 5 may be modified according to the requirements of a given application. Procedure BST_INSERT executes in $O(h)$ time on a BST of height h .

Deleting a node (record) from a BST

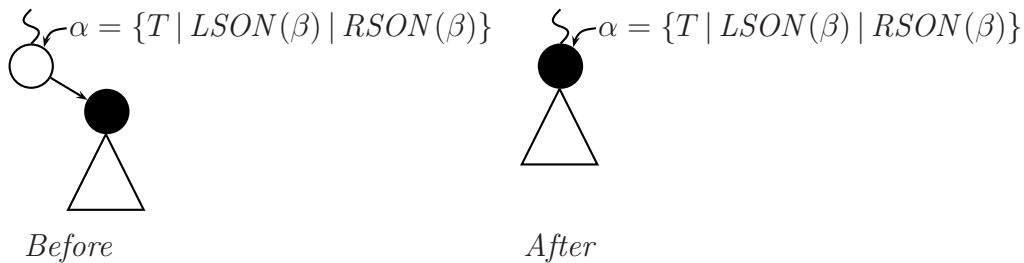
Deletion from a BST is a somewhat more complicated operation than insertion. Basically, the question is: How do we maintain the BST property after a node is deleted? It is easy to delete a leaf node, such as **[[DANIEL]]** or **[[HOSEA]]** in Figure 14.1. But consider deleting **[[EZEKIEL]]** or **[[OBADIAH]]** or **[[HABAKKUK]]**. Which node replaces the deleted node? How do we reset the links so that the result is still a BST?

Suppose that we want to delete node α from a BST, where node α is

- (a) the root node, i.e., $\alpha = T$
- (b) the left son of its father, say node β , i.e., $\alpha = LSON(\beta)$
- (c) the right son of its father, say node β , i.e., $\alpha = RSON(\beta)$

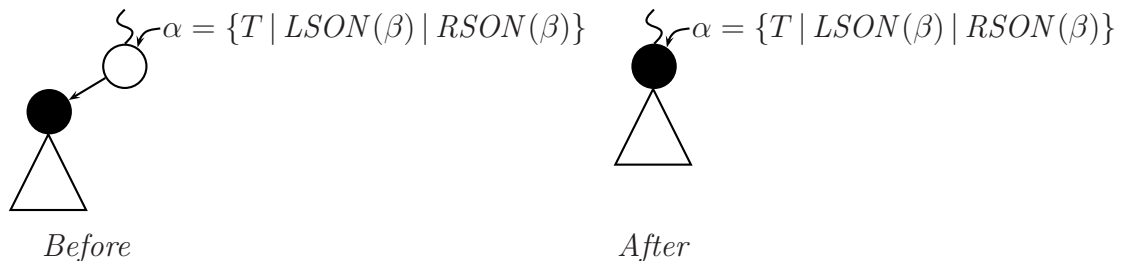
After node α is deleted, T or $LSON(\beta)$ or $RSON(\beta)$, as the case may be, will point to whichever node in the new BST replaces node α . To determine which node this is and how the links are reset, let us consider the different cases that may arise.

Case 1. Node α has no left son



In this case, the right son of node α replaces node α . For instance, if **[[AMOS]]** is deleted in the BST of Figure 14.1, then **[[EZEKIEL]]** replaces **[[AMOS]]**, i.e., **[[EZEKIEL]]** becomes the left son of **[[HABAKKUK]]**.

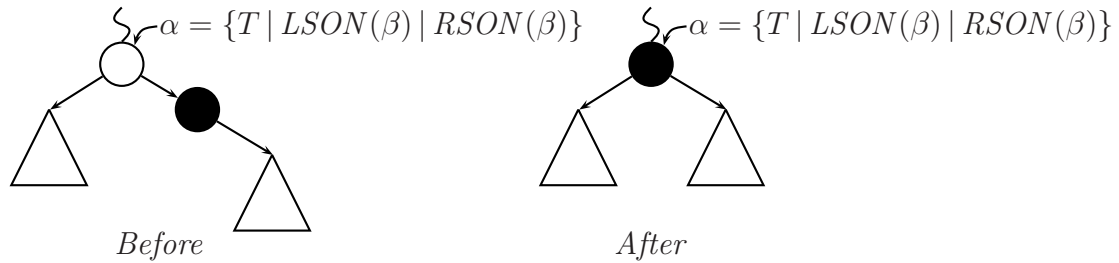
Case 2. Node α has no right son



In this case, the left son of node α replaces node α . For instance, if **[[EZEKIEL]]** is deleted from the BST of Figure 14.1, then **[[DANIEL]]** replaces **[[EZEKIEL]]**, i.e., **[[DANIEL]]** becomes the right son of **[[AMOS]]**.

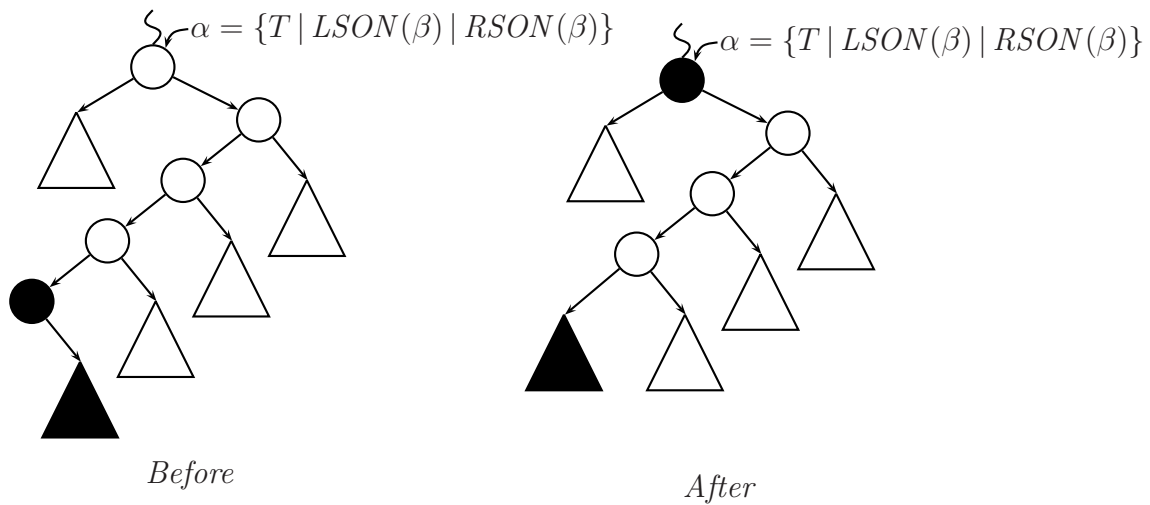
Case 3. Node α has a right subtree

(a) The right son of node α has no left subtree



In this case, the right son of node α replaces node α . For instance, if **[[OBADIAH]]** is deleted from the BST of Figure 14.1, then **[[ZECHARIAH]]** replaces **[[OBADIAH]]**, i.e., **[[OBADIAH]]** becomes the right son of **[[MICAH]]** and the father of **[[NEHEMIAH]]**.

(b) The right son of node α has a left subtree



In this case, the *leftmost* node of the right subtree of node α (i.e., the *inorder successor* of node α) replaces node α . For instance, if **[[HABAKKUK]]** is deleted from the BST of Figure 14.1, then **[[HAGGAI]]** replaces **[[HABAKKUK]]**, i.e., **[[HAGGAI]]** becomes the left son of **[[MICAH]]** and the father of **[[AMOS]]** and **[[JONAH]]**. In addition, **[[HOSEA]]**, who loses **[[HAGGAI]]** as a father, becomes the right son of **[[ISAIAH]]**.

Note that in all these cases, the inorder sequence of the nodes is preserved. This is readily verified by traversing both *Before* and *After* diagrams in inorder and comparing the resulting sequences.

The following EASY procedure formalizes the process described above. It is organized according to the four cases listed above. Unlike the preceding three procedures whose actions we can readily follow, it may take a little more mental effort to follow the inner workings of **BST_DELETE**. The *Before* and *After* diagrams given above should help provide the needed insight.

```

1  procedure BST_DELETE( $\mathbb{B}, \alpha$ )
  ▷ Deletes node  $\alpha$  from a BST where  $\alpha = \{T \mid LSON(\beta) \mid RSON(\beta)\}$ , i.e., node  $\alpha$  is the
  ▷ root of the entire BST or is either the left or the right son of some node  $\beta$  in the BST.
2     $\tau \leftarrow \alpha$ 
3    case
4      :  $\alpha = \Lambda$  : return
5      :  $LSON(\alpha) = \Lambda$  :  $\alpha \leftarrow RSON(\alpha)$ 
6      :  $RSON(\alpha) = \Lambda$  :  $\alpha \leftarrow LSON(\alpha)$ 
7      :  $RSON(\alpha) \neq \Lambda$  : [ $\gamma \leftarrow RSON(\alpha)$ 
8                              $\sigma \leftarrow LSON(\gamma)$ 
9                             if  $\sigma = \Lambda$  then [ $LSON(\gamma) \leftarrow LSON(\alpha)$ ;  $\alpha \leftarrow \gamma$ ]
10                            else [while  $LSON(\sigma) \neq \Lambda$  do
11                                 $\gamma \leftarrow \sigma$ 
12                                 $\sigma \leftarrow LSON(\sigma)$ 
13                                endwhile
14                                 $LSON(\gamma) \leftarrow RSON(\sigma)$ 
15                                 $LSON(\sigma) \leftarrow LSON(\alpha)$ 
16                                 $RSON(\sigma) \leftarrow RSON(\alpha)$ 
17                                 $\alpha \leftarrow \sigma$ ] ]
18    endcase
19    call RETNODE( $\tau$ )
20    return
21  end BST_DELETE

```

Procedure 14.4 Deleting a record from a binary search tree

In line 2 we keep in τ the address of the node to be deleted so that it can be returned to the memory pool (line 19) after the deletion. Line 4 handles the case in which α points to a null BST or it points to an external node; in this case there is nothing to delete and the procedure simply returns.

Lines 5 and 6 handle Case 1 and Case 2, respectively. Lines 7–17 handle Case 3; specifically, line 9 takes care of Case 3(a). The loop in lines 10–13 finds the inorder successor of node α , namely node σ . Lines 14–17 resets the links as indicated in the diagrams for Case 3(b).

Procedure BST_DELETE executes in $O(h)$ time on a BST of height h .

Finding the node with the next smaller or next larger key

We have noted earlier that an inorder enumeration of the nodes of a BST yields an ascending sequence of its keys. Hence, given node α with key k_i in a BST, the node with the next smaller or next larger key is simply the inorder predecessor (say, α^-) or inorder successor (say, α^+) of node α , respectively. In Session 6, we studied procedures to find α^- and α^+ in an inorder threaded binary tree. In the absence of threads, α^- and α^+ can be found by enumerating the nodes of the BST in inorder and comparing k_i with the key in each node. Clearly, this will take $O(n)$ time. It is possible to find α^- and α^+ in $O(h)$ time and without comparing keys if we add a *FATHER* field in each node of the BST. These are left as exercises (cf. procedures INSUC and INPRED in Session 6).

14.1.2 Analysis of BST search

Given a BST on n internal nodes, what is the average number of comparisons, C_n , in a successful search? What is the average number of comparisons, C'_n , in an unsuccessful search? As in the case of binary search on a sequential table which we considered in Session 13, the answers to these questions depend on the probabilities associated with each of the internal and external nodes of the BST. For purposes of the present analysis, we assume that each internal node is an equally likely target in a successful search, and that each external node is an equally likely target in an unsuccessful search. Under these assumptions, we have:

$$C_n = \frac{I + n}{n} \quad (14.2)$$

$$C'_n = \frac{E}{n + 1} \quad (14.3)$$

where

$$I = \begin{array}{l} \text{sum of the lengths of the paths from the root} \\ \text{to each of the } n \text{ internal nodes} \end{array} \quad (14.4)$$

$$E = \begin{array}{l} \text{sum of the lengths of the paths from the root} \\ \text{to each of the } n + 1 \text{ external nodes} \end{array} \quad (14.5)$$

$$= I + 2n \quad (14.6)$$

Note that these are the same equations we used to describe the average performance of binary search in Session 13. This is to be expected since binary search is really just a special case of BST search. We already discussed at length the meaning of these quantities in Session 13; you are encouraged to go through the pertinent material before reading on.

Let us now compute these quantities for the BST of Figure 14.1.

$$I = 0 + 2(1) + 4(2) + 5(3) + 3(4) + 1(5) = 42$$

$$E = 3(3) + 7(4) + 5(5) + 2(6) = 74$$

$$C_n = \frac{I + n}{n} = \frac{42 + 16}{16} = 3.625$$

$$C'_n = \frac{E}{n + 1} = \frac{74}{16 + 1} = 4.353$$

Thus, assuming that we are as likely to be looking for **AMOS** as we would be for **DANIEL** and for each of the other names, then we will be performing 3.625 comparisons on the average in a successful search. Similarly, assuming that we are as likely to be looking for a name less than **AMOS** or greater than **ZEPHANIAH** as we would be for a name between **AMOS** and **DANIEL** and for names in each of the other intervals between keys in the BST, then we will be performing 4.353 comparisons on the average in an unsuccessful search. These values are for the the BST of Figure 14.1. For *other* BST's on the *same* set of keys **AMOS** through **ZEPHANIAH**, the average values will be different. So it is pertinent to ask for any BST on n distinct keys:

- (a) What is the *minimum average* number of comparisons in a successful search? In an unsuccessful search?
- (b) What is the *maximum average* number of comparisons in a successful search? In an unsuccessful search?
- (c) What is the *average average* number of comparisons in a successful search? In an unsuccessful search?

We will now consider each of these cases in turn. Note that it suffices to find C_n in each case; applying Eq.(14.7), which is easily obtained by combining Eq.(14.2), Eq.(14.3) and Eq.(14.6), gives us the corresponding C'_n .

$$C'_n = \frac{n}{n+1} (C_n + 1) \quad (14.7)$$

Minimal BST on n internal nodes

It is clear from Eq.(14.2) and Eq.(14.3) that C_n and C'_n are minimum if I and E are minimum. Since I and E differ only by a constant, viz., $2n$, a BST which minimizes I also minimizes E . We call this a **minimal BST**. The following statements are equivalent ways of describing a minimal BST.

- (a) It has minimum height.
- (b) Each level of the binary tree, except possibly the lowest, has its full complement of internal nodes: 1, 2, 4, 8, 16, ...
- (c) All external nodes appear on at most two adjacent levels, viz., the bottommost and next to the bottommost.

From statement (b), it follows that the internal path length of a minimal BST on n internal nodes is

$$\begin{aligned} I &= 0 + \underbrace{1+1}_{2 \text{ 1's}} + \underbrace{2+2+2+2}_{4 \text{ 2's}} + \underbrace{3+3+\dots+3}_{8 \text{ 3's}} + \underbrace{4+4+\dots+4}_{16 \text{ 4's}} + \dots \\ &= \sum_{i=1}^n \lfloor \log_2 i \rfloor = (n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2 \end{aligned} \quad (14.8)$$

Hence, the *minimum average* number of comparisons in a successful search is

$$\text{Min. } C_n = \frac{I + n}{n} = \frac{(n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2}{n} + 1 \quad (14.9)$$

$$\approx \log_2 n + 1 \quad \text{for large enough } n \quad (14.10)$$

Figure 14.2 shows a minimal BST on the 16 keys AMOS, ..., ZEPHANIAH. There are 15 more minimal BST's on these set of 16 keys. For each of these 16 minimal BST's we have: $I = 38$, $E = 70$, $C_n = 3.375$ and $C'_n = 4.118$, as you may easily verify. These are the same values that we obtained for binary search on a table of size 16 (see section

13.3.2). Comparing Figure 13.5 and Figure 14.2, we find that binary search on an ordered sequential table is equivalent to BST search on a minimal BST.

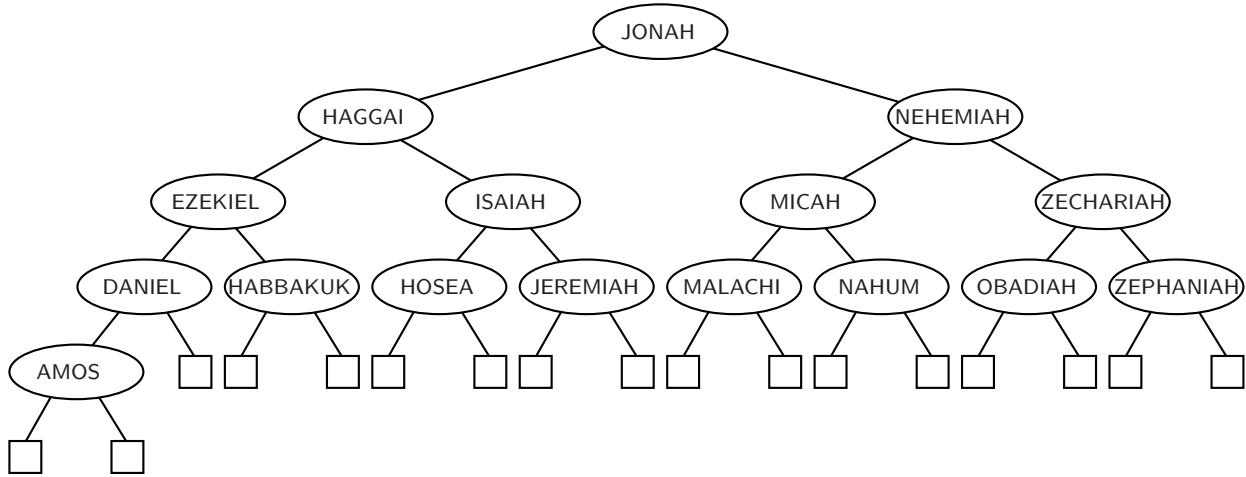


Figure 14.2 A minimal BST on the 16 keys AMOS, ..., ZEPHANIAH

Maximal BST on n internal nodes

It is clear from Eq.(14.2) and Eq.(14.3) that C_n and C'_n are maximum if I and E are maximum. We call a BST for which I and E are maximum a **maximal BST**. The following statements are equivalent ways of describing a maximal BST.

- (a) It has maximum height.
- (b) Each level of the binary tree has exactly one internal node. (Such a binary tree is said to be *degenerate*.)

From statement (b), it follows that the internal path length of a maximal BST on n internal nodes is

$$I = 0 + 1 + 2 + 3 + \dots + (n-1) = \frac{(n-1)n}{2} \quad (14.11)$$

Hence, the *maximum average* number of comparisons in a successful search is

$$\text{Max. } C_n = \frac{I + n}{n} = \frac{\frac{(n-1)n}{2} + n}{n} = \frac{n+1}{2} \quad (14.12)$$

This is the same as the average number of comparisons for linear search on a sequential table of size n (see Eq.(13.1)). Thus, for successful searches, BST search on a maximal BST is equivalent to linear search.

Figure 14.3 shows a maximal BST on the 16 keys AMOS, ..., ZEPHANIAH. There are $2^{15} - 1 = 32,767$ more maximal BST's on these set of 16 keys. For each of these 32,768 maximal BST's we have: $I = 120$, $E = 152$, $C_n = 8.500$ and $C'_n = 8.941$, as you may easily verify.

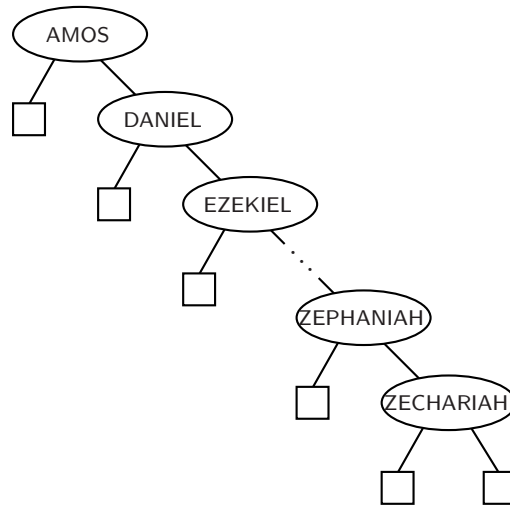


Figure 14.3 A maximal BST on the 16 keys AMOS, \dots , ZEPHANIAH

Average BST on n internal nodes

Given n keys, there are $n!$ ways in which we can insert the n keys into an initially empty BST. Now imagine taking the n keys and constructing BST's considering each possible permutation of the n keys as equally likely. Some would be minimal BST's while others would be maximal; we already considered these cases. Now, what is the average number of comparisons in the *average* BST? The analysis is based on the following easily verifiable facts.

- (a) Inserting a key into a BST is always preceded by an unsuccessful search.
- (b) The number of comparisons needed to find a key in a successful search is one more than the number of comparisons performed in an unsuccessful search when the key was inserted.

Now, assume that the key K is the j th key to be inserted into some BST; at this point, there will already be $i = j - 1$ nodes in the BST. The average number of comparisons performed to insert K into the BST is C'_i , which is the average number of comparisons in an unsuccessful search in a BST on i nodes. Subsequently, when K is searched for, $C'_i + 1$ comparisons will be needed. Summing these quantities for $0 \leq i \leq n - 1$ gives the total number of comparisons performed in an average BST on n nodes if each one of the n keys were used as a search key. Dividing this total by n yields the average number of comparisons in an average BST.

$$\text{Ave. } C_n = \frac{\sum_{i=0}^{n-1} (C'_i + 1)}{n} = 1 + \frac{C'_0 + C'_1 + C'_2 + \dots + C'_{n-1}}{n} \quad (14.13)$$

In its present form Eq.(14.13) is not readily computed; to obtain an expression that is, we proceed as follows. Combining Eq.(14.7) and Eq.(14.13) we get

$$(n + 1) C'_n = 2n + C'_0 + C'_1 + \dots + C'_{n-2} + C'_{n-1} \quad (a)$$

Replacing n by $n - 1$ in (a) we have

$$nC'_{n-1} = 2(n-1) + C'_0 + C'_1 + \dots + C'_{n-3} + C'_{n-2} \quad (b)$$

Subtracting (b) from (a) and simplifying we obtain the recurrence equation

$$C'_n = C'_{n-1} + \frac{2}{n+1} \quad (c)$$

Unrolling (c) and noting that $C'_1 = 1$ we obtain

$$\begin{aligned} C'_n &= C'_{n-1} + \frac{2}{n+1} \\ &= C'_{n-2} + \frac{2}{n} + \frac{2}{n+1} \\ &= C'_{n-3} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= C'_2 + \frac{2}{3+1} + \frac{2}{4+1} + \dots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= C'_1 + \frac{2}{2+1} + \frac{2}{3+1} + \frac{2}{4+1} + \dots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= 1 + \underbrace{\frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \dots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}}_{2 \sum_{j=1}^{n+1} \frac{1}{j} - 2} \\ &= 2(H_{n+1} - 1) \end{aligned} \quad (d)$$

where H_{n+1} is the $(n+1)$ th harmonic number (cf. Eq.(2.29) in Session 2). Substituting (d) in Eq.(14.7) we obtain the desired expression for the *average* number of comparisons in a successful search in an *average* BST, thus

$$\begin{aligned} \text{Ave. } C_n &= \left(\frac{n+1}{n}\right) C'_n - 1 \\ &= 2 \left(\frac{n+1}{n}\right) (H_{n+1} - 1) - 1 \\ &= 2 \left(\frac{n+1}{n}\right) \left(H_n + \frac{1}{n+1} - 1\right) - 1 \\ &= 2 \left(\frac{n+1}{n}\right) H_n - 3 \end{aligned} \quad (14.14)$$

Eq.(14.14) gives an exact value of C_n . For sufficiently large n we can use the approximation $H_n \approx \log_e n + \gamma$ where $\gamma = 0.577215\dots$ is Euler's constant, to get

$$\text{Ave. } C_n \approx 2 \log_e n \approx 1.386 \log_2 n \quad (14.15)$$

Figure 14.4 summarizes the results we obtained for the analysis of BST search when the search is successful. The corresponding equations or values for unsuccessful searches are obtained by applying Eq.(14.7). It is well to reiterate at this point that all these results are valid only if each internal node in the BST is an equally likely target in a successful search and each external node is an equally likely target in an unsuccessful search. The approximate expressions yield good approximations for large enough n .

	Average number of comparisons (successful search), C_n	
	Exact value	Approximate value
Minimal BST	$\frac{(n+1)\lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2}{n} + 1 \quad (14.9)$	$\log_2 n + 1 \quad (14.10)$
Average BST	$2 \left(\frac{n+1}{n} \right) \left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right) - 3 \quad (14.14)$	$1.386 \log_2 n \quad (14.15)$
Maximal BST	$\frac{n+1}{2} \quad (14.12)$	—

Figure 14.4 Average number of comparisons for BST search (successful search)

Comparing Eqs. (14.10) and (14.15), we find that the average number of comparisons in a successful search in the average BST is only about 38% more than the corresponding number in a minimal BST. Put another way, the average BST is only about 38% worse than the best BST. In plain English: *if keys are inserted into a BST in random order, well-balanced trees are common and degenerate ones are very rare* (KNUTH3[1998], p.430).

n	Minimal BST		Average BST		Maximal BST
	Eq.(14.9)	Eq.(14.10)	Eq.(14.14)	Eq.(14.15)	Eq.(14.12)
16	3.38	5.00	4.18	5.54	8.50
128	6.07	8.00	7.95	9.70	64.50
1024	9.01	11.00	12.03	13.86	512.50
5000	11.36	13.29	15.19	17.03	2500.50
10000	12.36	14.29	16.58	18.42	5000.50
100000	15.69	17.61	21.18	23.02	50000.50
1000000	18.95	20.93	25.78	27.62	500000.50

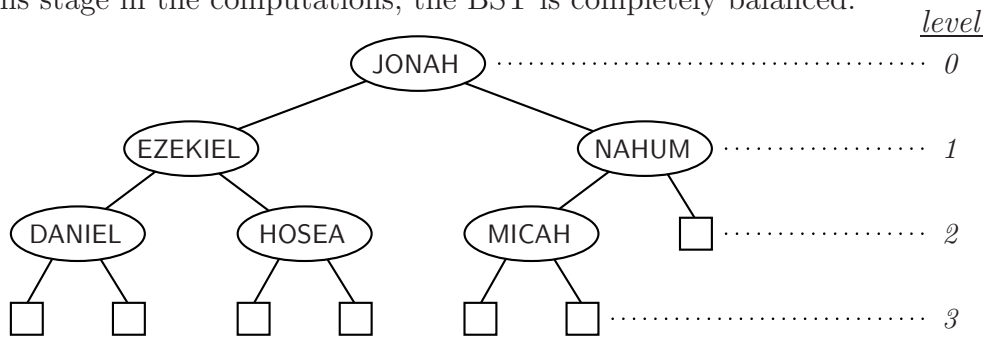
Figure 14.5 Values of C_n for selected values of n

The table above shows values of C_n for selected values of n . The entries in the last column should alert us to the fact that although degenerate BST's are rare, things can get very bad when they do occur. For instance, it takes only 11.36 comparisons on the average to find a record in a minimal BST on 5000 keys, 15.19 comparisons in an average BST of the same size and 2500 comparisons in a maximal BST on the same set of keys. This suggests that it is worthwhile to find ways of preventing the occurrence of degenerate BST's. This brings us to the next section.

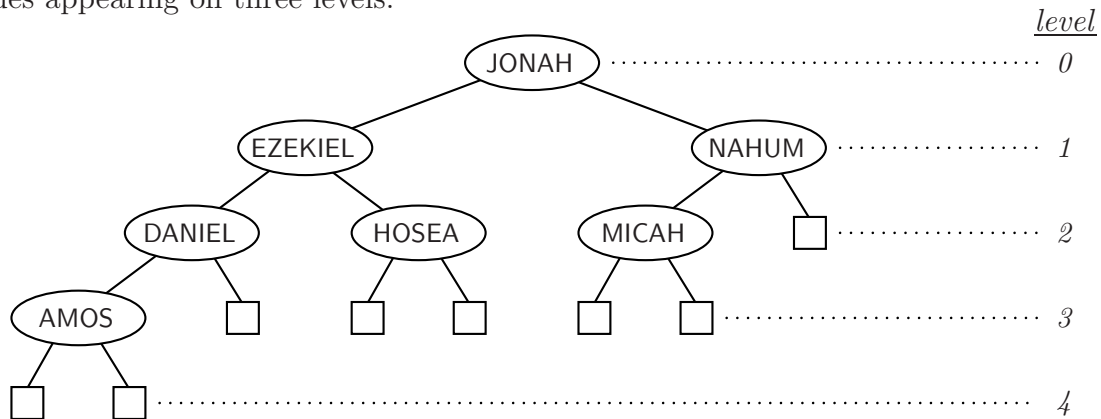
14.2 Height-balanced binary search trees

We have seen in the preceding section that if each key in a BST is equally likely to be a search key, then a BST in which external nodes appear on at most two adjacent levels minimizes the average number of comparisons, or average search time, in both successful and unsuccessful searches. Earlier we used the term *minimal* to describe such a BST; another way to characterize such a BST is to say that it is *completely balanced*. What does it take to keep a BST completely balanced under dynamic conditions in which searches, insertions and deletions commingle? Consider the series of figures shown below which depict the effect of a single insertion into a completely balanced BST.

(a) At this stage in the computations, the BST is completely balanced.



(b) After [AMOS] is inserted, the BST ceases to be completely balanced, with external nodes appearing on three levels.



(c) Rebalancing the BST to restore complete balance causes every node to change position in the BST.

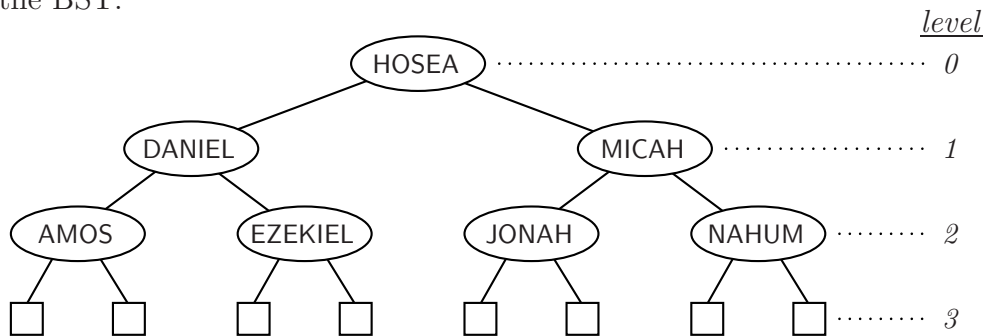


Figure 14.6 Maintaining complete balance under dynamic conditions

This example shows that keeping a BST completely balanced under dynamic conditions can be very expensive in that it may require resetting practically every link in the entire BST. The computational cost of maintaining complete balance may completely negate the expected benefits in searching a minimal BST.

As it is with most situations in an imperfect world, we must settle for a compromise. We will not insist that a BST be always minimal; however, we will not allow it to become maximal (degenerate) either. A class of BST's which implement this idea are the so-called **height-balanced binary trees** or **HB(k)-trees**.

DEFINITION 14.2. A BST is said to be an HB(k)-tree if the height of the left subtree of every node differs by at most k from the height of its right subtree.

The most common variety of HB(k)-trees is the HB(1)-tree, also called an **AVL tree**, named after the two Russian mathematicians G.M. Adel'son-Vel'skii and E.M. Landis who first defined it. The figure below shows two BST's: a non-AVL tree and an AVL tree.

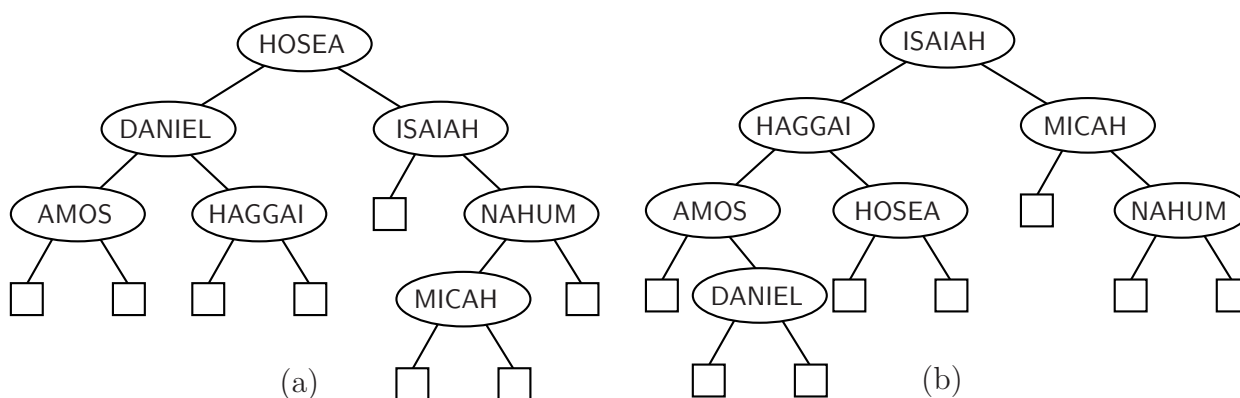
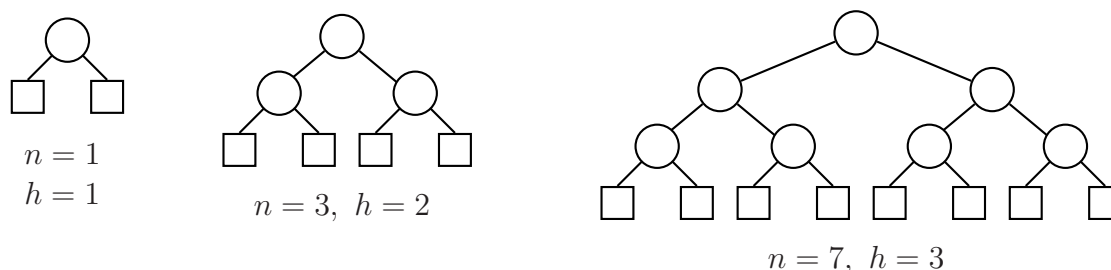


Figure 14.7 (a) A non-AVL tree (b) An AVL tree

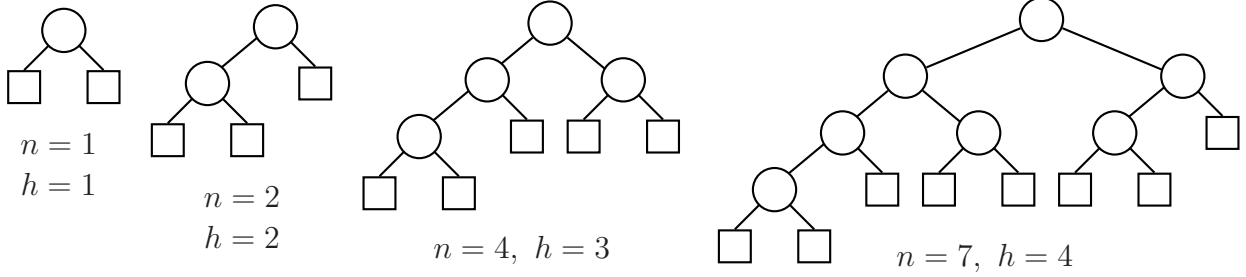
Let us now look at the features of AVL trees which make them suitable BST's in a dynamic environment. The first important characteristic of an AVL tree is embodied in the following theorem after Adel'son-Vel'skii and Landis (1962).

THEOREM 14.1 (Adel'son-Vel'skii and Landis). The height of an AVL tree on n internal nodes lies between $\log_2(n+1)$ and $1.4404 \log_2(n+2) - 0.328$.

Proof: To establish the lower bound, we construct AVL trees with the most number of nodes for a given height h . Shown below are three such best-case AVL trees. Clearly, a binary tree of height h has at most 2^h external nodes, i.e., $n+1 \leq 2^h$. Hence, $h \geq \lceil \log_2(n+1) \rceil$.



Next, to establish the upper bound, we construct AVL trees with the least number of nodes for a given height h . We do this by making the height of the right subtree (left subtree) of any node strictly one less than the height of the left subtree (right subtree) of the node. The figure below shows four such worst-case AVL trees.



We recognize these worst-case AVL trees as Fibonacci trees, sans the node labels, of order 2, 3, 4 and 5, respectively, or as Fibonacci trees of order $h+1$, where h is the height of the AVL tree. To determine the relationship between h and n for this case, we make use of the following facts:

- (a) The number of external nodes in an AVL tree of height h is at least equal to the number of external nodes in a Fibonacci tree of order $h+1$.
- (b) The Fibonacci tree of order k has F_{k+1} external nodes (see section 13.3.4 of Session 13).
- (c) Let $\phi = \frac{1}{2}(1 + \sqrt{5})$. This quantity is called the golden ratio or the divine proportion. From Eq.(2.15) in Session 2 we have

$$F_k > \frac{\phi^k}{\sqrt{5}} - 1$$

Putting (a), (b) and (c) together we obtain

$$\begin{aligned} n+1 &\geq F_{h+2} \\ &> \frac{\phi^{h+2}}{\sqrt{5}} - 1 \end{aligned}$$

or

$$n+2 > \frac{\phi^{h+2}}{\sqrt{5}}$$

Taking logarithms of both sides, we have

$$\begin{aligned} \log_\phi(n+2) &> \log_\phi \frac{\phi^{h+2}}{\sqrt{5}} \\ &> h+2 - \log_\phi \sqrt{5} \end{aligned}$$

or

$$h < \log_\phi(n+2) + \log_\phi \sqrt{5} - 2$$

Finally, using the identity $\log_\phi x = \log_2 x / \log_2 \phi$, we get

$$h < 1.4404 \log_2(n+2) - 0.328$$

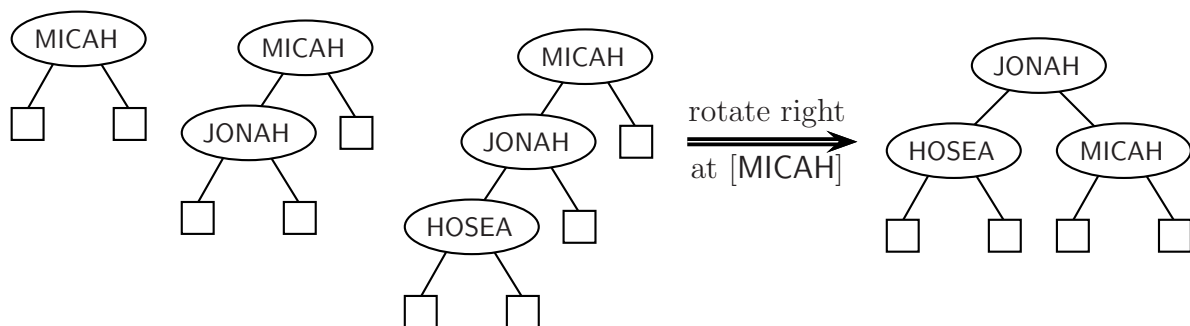
Hence we have $\lfloor \log_2(n+1) \rfloor \leq h < 1.4404 \log_2(n+2) - 0.328$, thus establishing the theorem. Essentially what the theorem says is that the maximum number of comparisons performed in searching an AVL tree is never more than 45% more than the minimum. Put another way, a worst-case AVL tree is never more than 45% worse, in terms of average search times, than a best-case or minimal BST.

We now ask the question: What does it take to maintain the AVL property in a dynamic environment? We expect that it should not be too expensive computationally, since we have agreed to permit some imbalance in the tree. The answer to this question lies in a basic operation on AVL trees called a *rotation*.

14.3 Rotations on AVL trees

When an AVL tree loses the AVL property due to an *insertion*, the AVL property is restored by performing *one* of four types of rotations. (We will consider the case of deletion in the next section.) The following figures illustrate each type of rotation in its simplest form.

Case 1. AVL property restored by a **right rotation**



Case 2. AVL property restored by a **left rotation**

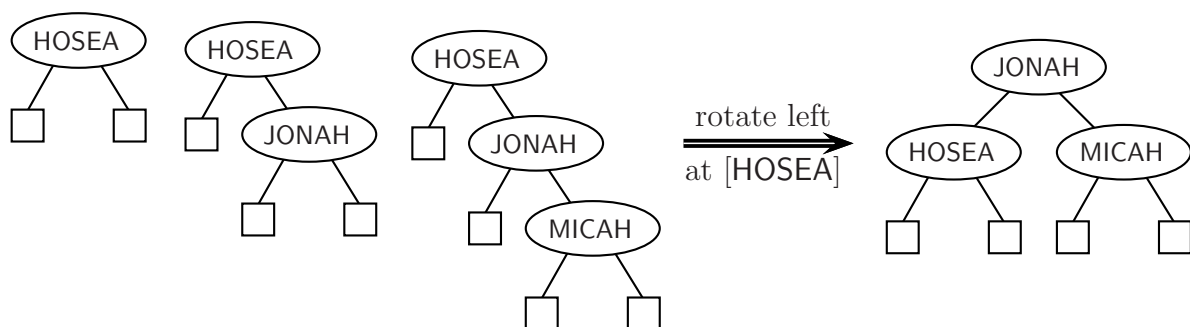
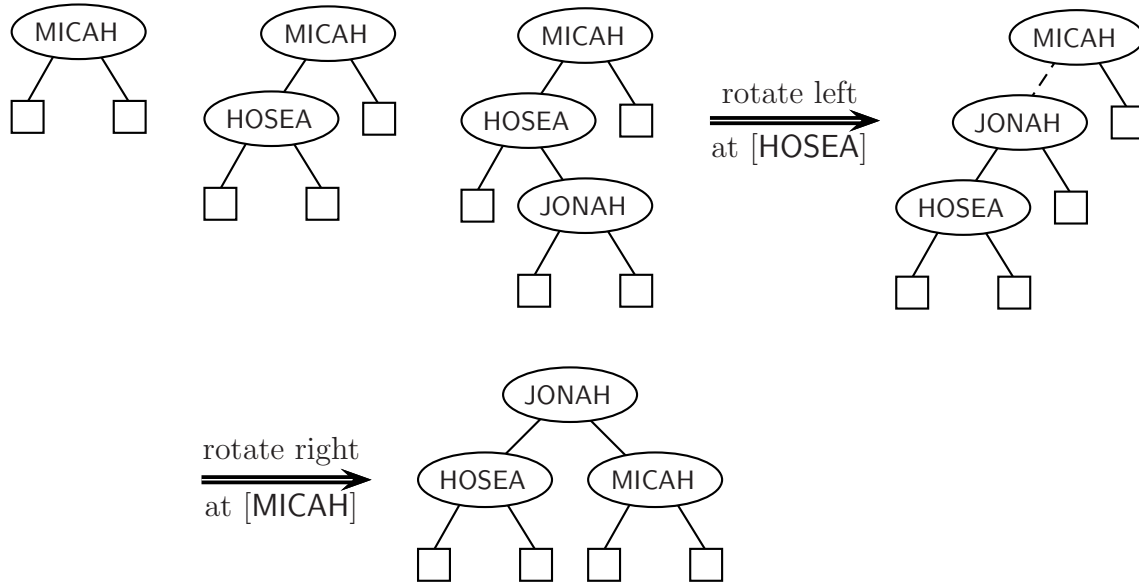


Figure 14.8 The four basic types of rotations on AVL trees (in their simplest form)
(continued on next page)

Case 3. AVL property restored by a **left-right** rotation



Case 4. AVL property restored by a **right-left** rotation

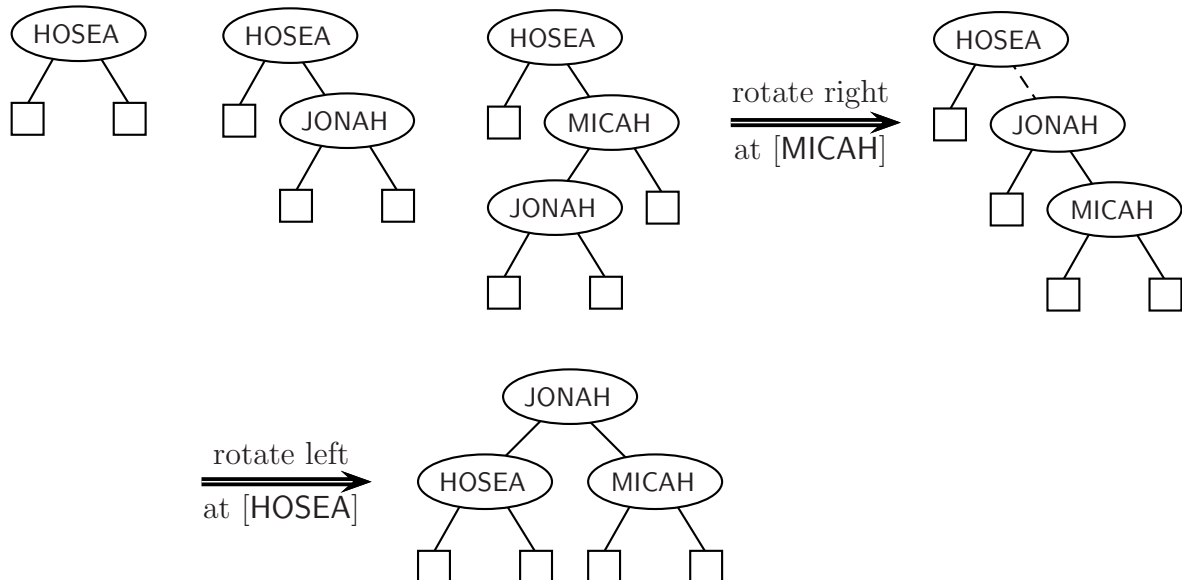
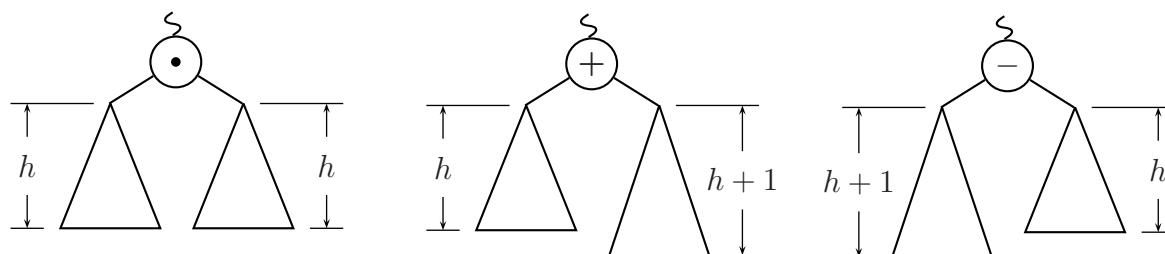


Figure 14.8 The four basic types of rotations on AVL trees (in their simplest form)

The rotation which restores the AVL property in Cases 1 and 2 is called a *single rotation* and the rotation which restores the AVL property in Cases 3 and 4 is called a *double rotation*; both single and double rotations must be viewed as ‘unit’ operations. When an AVL tree ceases to be AVL after an insert operation, *only one* type of rotation, applied on the *smallest* subtree that is non-AVL, is required to convert the entire BST into an AVL tree once again.

Let us now consider how these rotations are implemented in the general case. To keep track of the difference in height of the subtrees of a node, we will associate with each node in the AVL tree a *balance factor*, as shown below:



When the balance factor at some node, say node α , is '+' or '-', insertion into the taller subtree of node α causes the subtree rooted at α to become non-AVL. Consider the subtree shown in Figure 14.9(a). If a new node is inserted into T_3 then the subtree becomes perfectly balanced; however if the new node is inserted into either T_1 or T_2 then the subtree

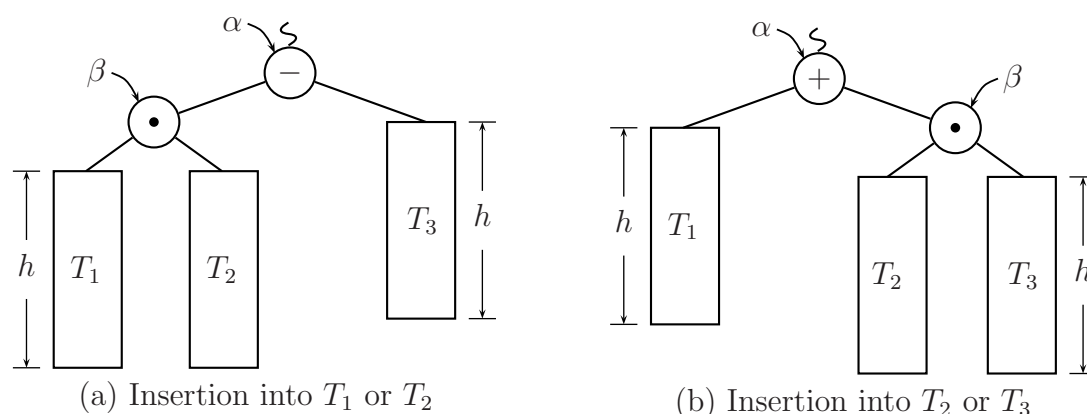


Figure 14.9 Insertion-sensitive AVL trees

rooted at node α becomes non-AVL. Similarly, if a new node is inserted into T_1 of the subtree shown in Figure 14.9(b) then the subtree becomes perfectly balanced; however if the new node is inserted into either T_2 or T_3 then the subtree rooted at α becomes non-AVL. To restore the AVL property the requisite rotation, as summarized in the table below, must be applied on the subtree rooted at node α .

Case	Figure	Insertion into	AVL property restored after a
1	14.9(a)	T_1 (left subtree of left son of α)	right rotation at α
2	14.9(b)	T_3 (right subtree of right son of α)	left rotation at α
3	14.9(a)	T_2 (right subtree of leftson of α)	left-right rotation (at β , at α)
4	14.9(b)	T_2 (left subtree of right son of α)	right-left rotation (at β , at α)

Figures 14.10 through 14.13 show *before* and *after* diagrams indicating in detail how the requisite rotation is performed when each of the four cases listed in the table obtains. Also given is a segment of EASY code which implements the rotation. Study these figures carefully.

Implementing a right rotation

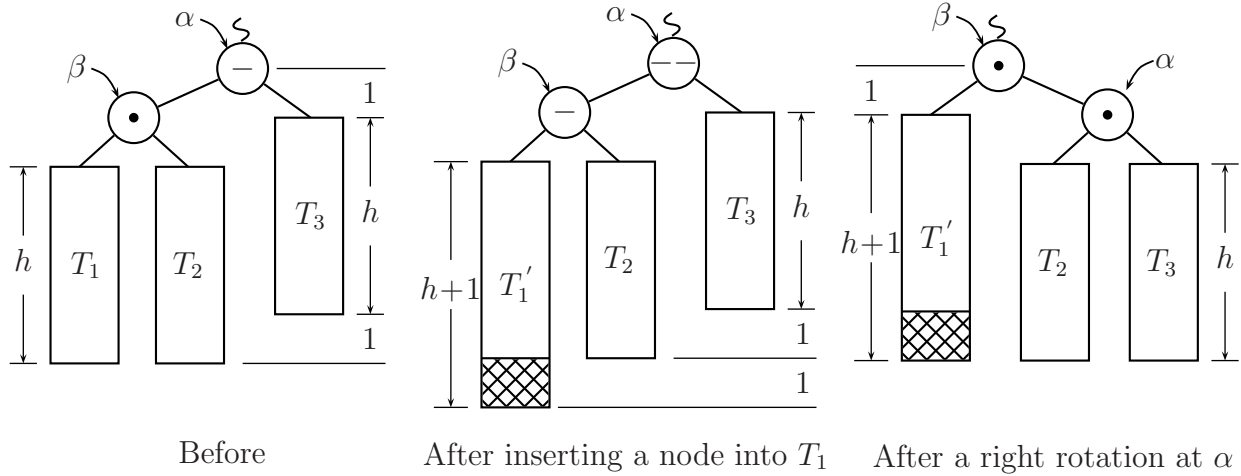


Figure 14.10 Implementing a right rotation

The following segment of EASY code performs the rotation.

```

 $\beta \leftarrow LSON(\alpha)$      $\triangleright$  becomes the root of the rebalanced tree/subtree
 $LSON(\alpha) \leftarrow RSON(\beta)$ 
 $RSON(\beta) \leftarrow \alpha$ 
 $BF(\alpha) \leftarrow BF(\beta) \leftarrow 0$ 

```

Implementing a left rotation

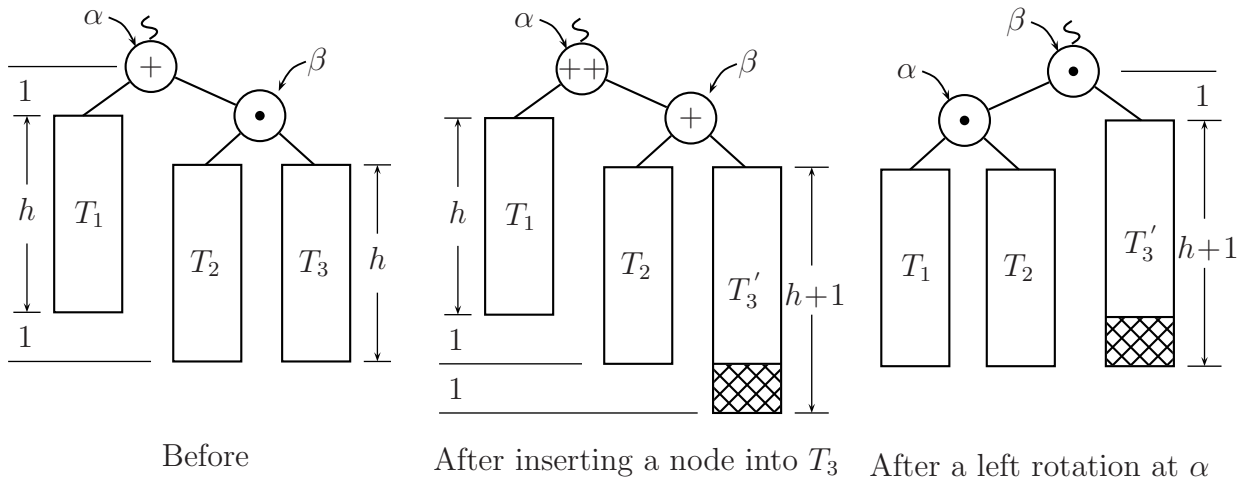


Figure 14.11 Implementing a left rotation

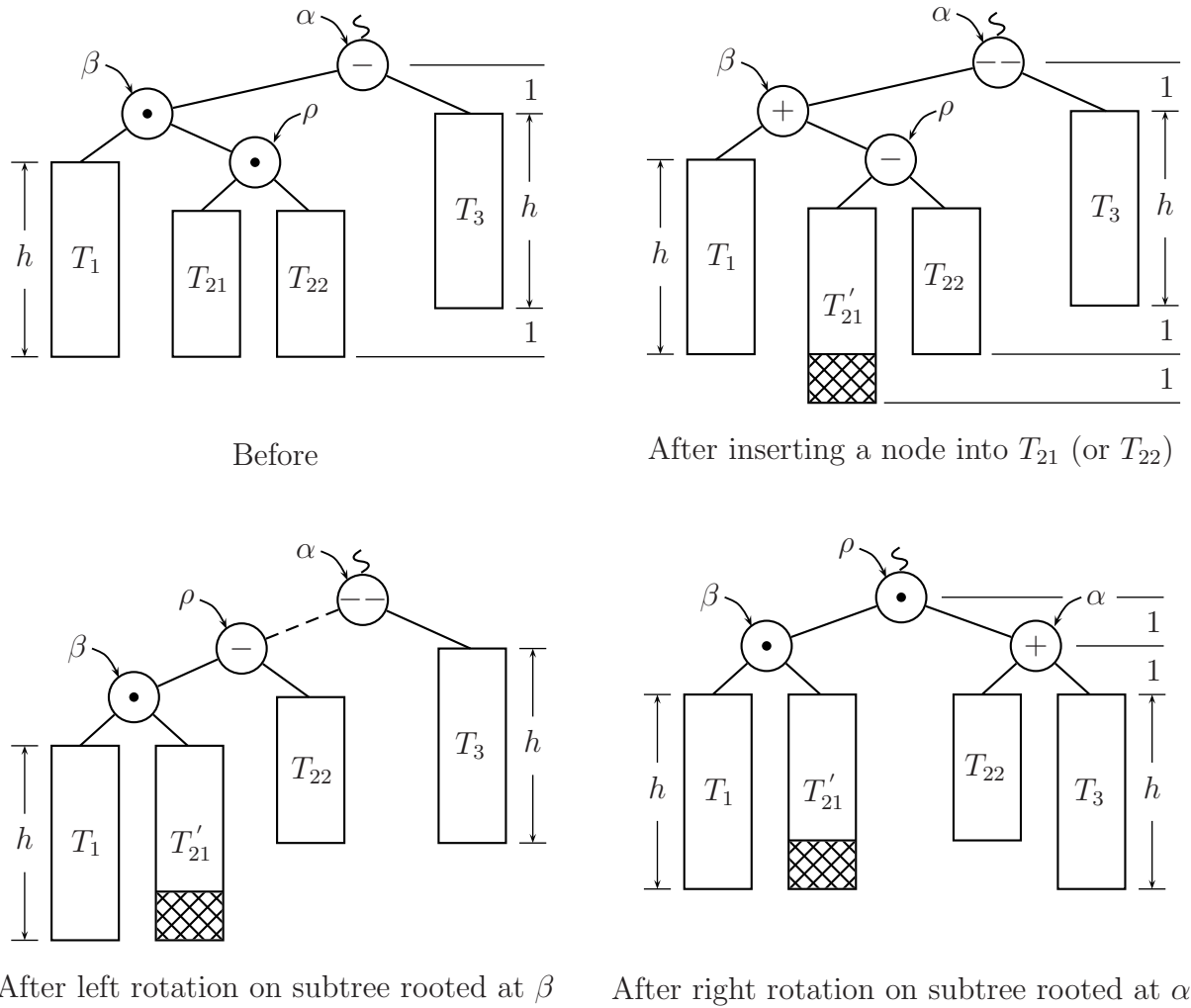
The following segment of EASY code performs the rotation.

```

 $\beta \leftarrow RSON(\alpha)$      $\triangleright$  becomes the root of the rebalanced tree/subtree
 $RSON(\alpha) \leftarrow LSON(\beta)$ 
 $LSON(\beta) \leftarrow \alpha$ 
 $BF(\alpha) \leftarrow BF(\beta) \leftarrow 0$ 

```

Implementing a left-right rotation

**Figure 14.12** Implementing a left-right rotation

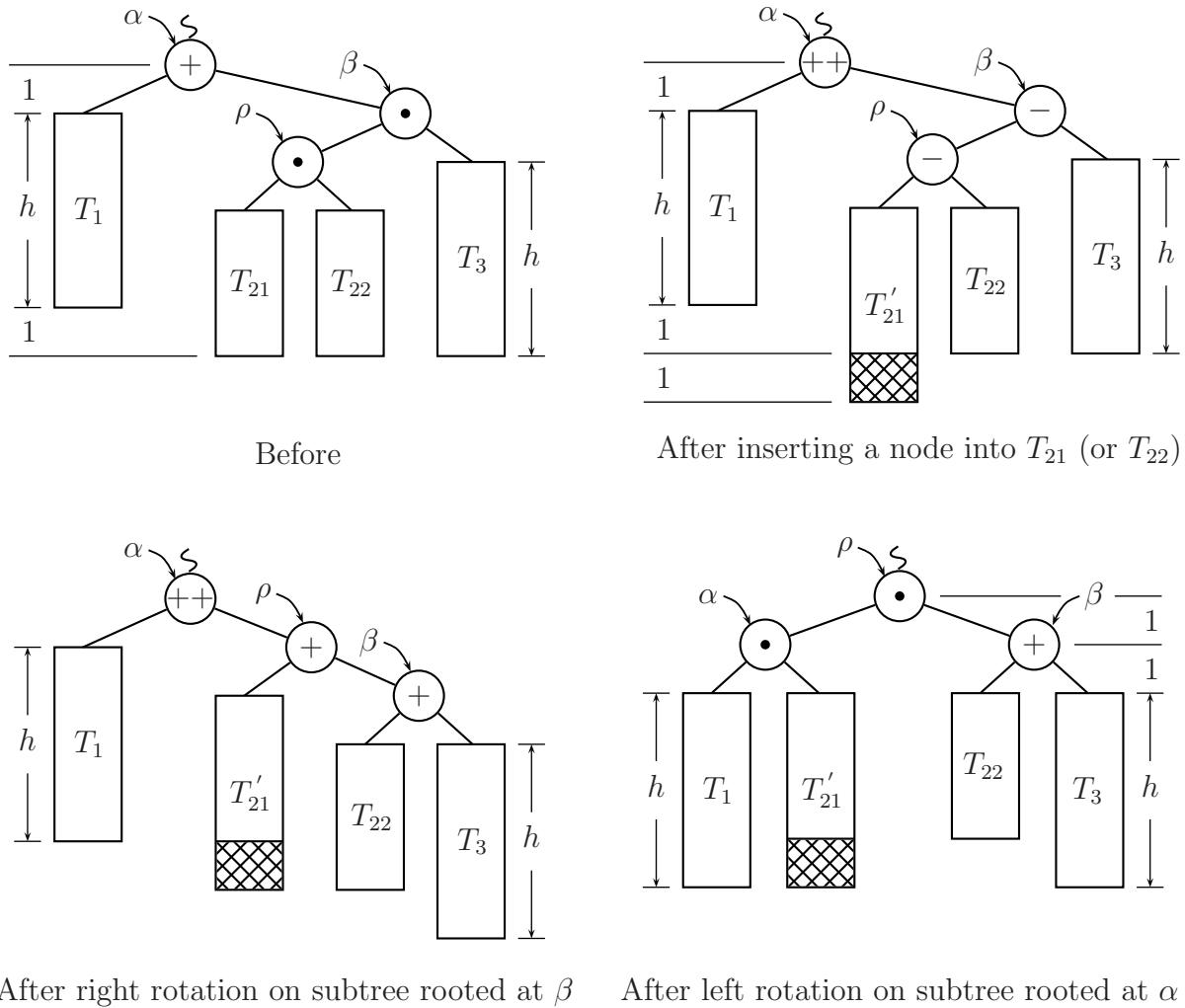
The following segment of EASY code performs the rotation.

```

 $\beta \leftarrow LSON(\alpha)$ 
 $\rho \leftarrow RSON(\beta)$   $\triangleright$  becomes the root of the rebalanced tree/subtree
 $RSON(\beta) \leftarrow LSON(\rho)$ 
 $LSON(\rho) \leftarrow \beta$ 
 $LSON(\alpha) \leftarrow RSON(\rho)$ 
 $RSON(\rho) \leftarrow \alpha$ 
case
  :  $BF(\rho) = 0$  : [  $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow 0$  ]
  :  $BF(\rho) = +1$  : [  $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow -1$  ]
  :  $BF(\rho) = -1$  : [  $BF(\alpha) \leftarrow +1$ ;  $BF(\beta) \leftarrow 0$  ]
endcase
 $BF(\rho) \leftarrow 0$ 

```

Implementing a right-left rotation

**Figure 14.13** Implementing a right-left rotation

The following segment of EASY code performs the rotation.

```

 $\beta \leftarrow RSON(\alpha)$ 
 $\rho \leftarrow LSON(\beta)$   $\triangleright$  becomes the root of the rebalanced tree/subtree
 $LSON(\beta) \leftarrow RSON(\rho)$ 
 $RSON(\rho) \leftarrow \beta$ 
 $RSON(\alpha) \leftarrow LSON(\rho)$ 
 $LSON(\rho) \leftarrow \alpha$ 
case
  :  $BF(\rho) = 0$  : [  $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow 0$  ]
  :  $BF(\rho) = +1$  : [  $BF(\alpha) \leftarrow -1$ ;  $BF(\beta) \leftarrow 0$  ]
  :  $BF(\rho) = -1$  : [  $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow +1$  ]
endcase
 $BF(\rho) \leftarrow 0$ 

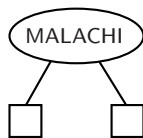
```

Note that in all four types of rotation, the height of the rebalanced subtree is the *same* as the height of the subtree rooted at node α before the insert operation. This is significant, because this means that the rest of the tree above node α remains AVL. Thus, restoring the AVL property after an insert operation takes $O(1)$ time. This contrasts markedly with the $O(n)$ time that it may take to restore complete balance in a BST.

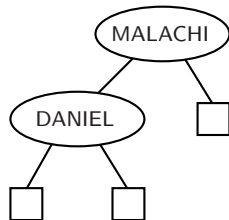
Example 14.1. Building an AVL tree

Let us illustrate with an example how to construct an AVL tree given a set of records to be inserted into an initially empty BST. Specifically, assume that we insert the following records, identified by the indicated keys, in the order given: MALACHI, DANIEL, AMOS, EZEKIEL, ISAIAH, ZECHARIAH, NEHEMIAH, JEREMIAH, ZEPHANIAH, MICAH and JONAH. Here goes.

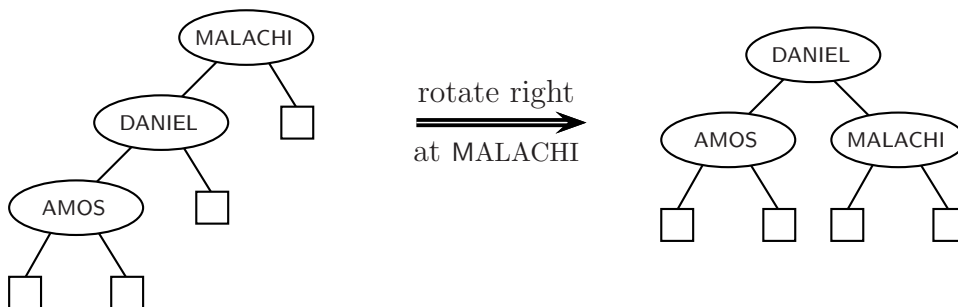
The first node is $\llbracket \text{MALACHI} \rrbracket$.



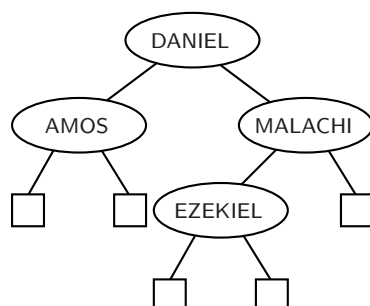
Next we insert $\llbracket \text{DANIEL} \rrbracket$.



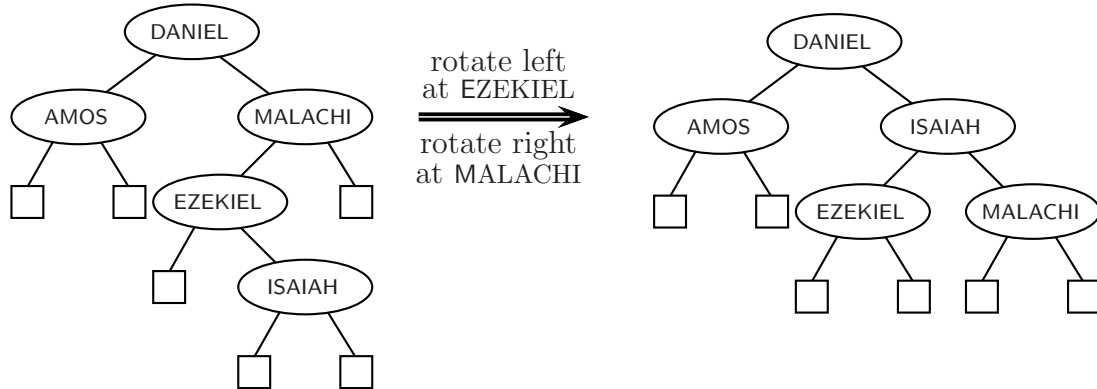
Inserting $\llbracket \text{AMOS} \rrbracket$ destroys the AVL property at $\llbracket \text{MALACHI} \rrbracket$ and is restored by performing a *right rotation*.



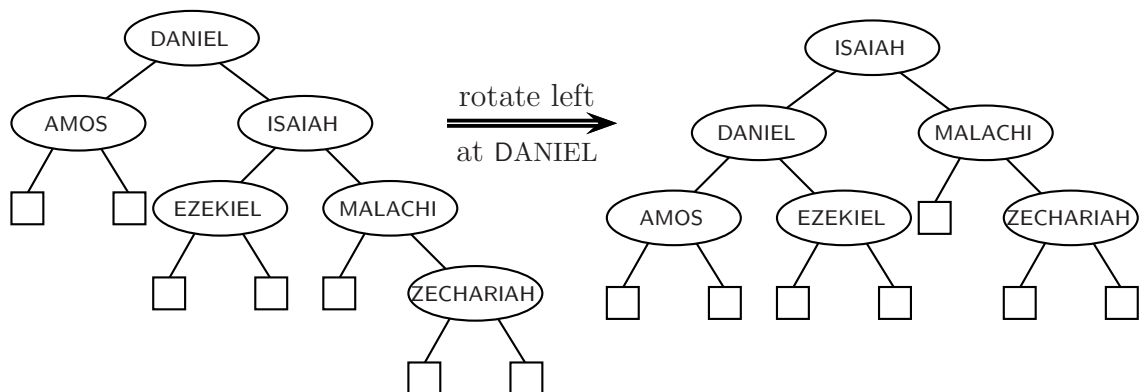
Inserting $\llbracket \text{EZEKIEL} \rrbracket$ maintains the AVL property.



Inserting **ISAIAH** destroys the AVL property at **MALACHI** and is restored by performing a *left-right rotation*.



Inserting **ZECHARIAH** causes the entire BST to become non-AVL; a *left rotation* at the root **DANIEL** restores the AVL property.



Inserting **NEHEMIAH** destroys the AVL property at **MALACHI** and is restored by applying a *right-left rotation*.

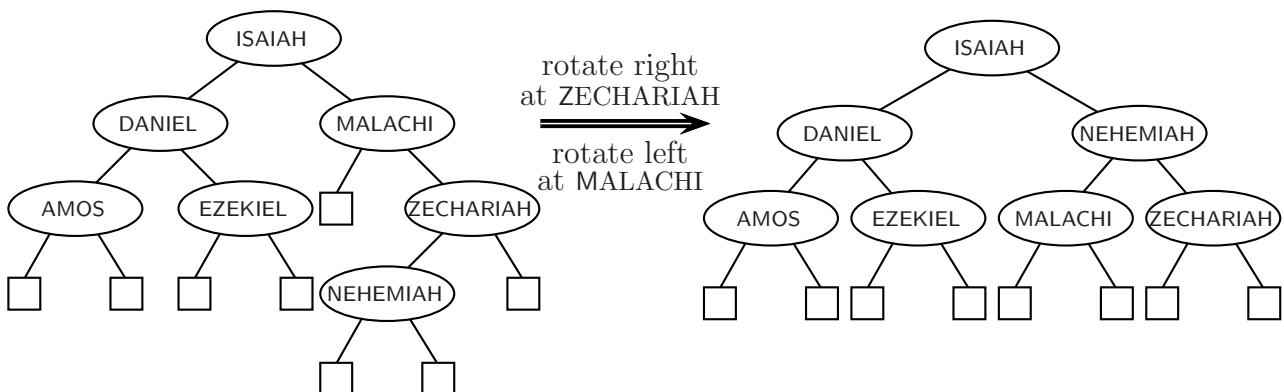
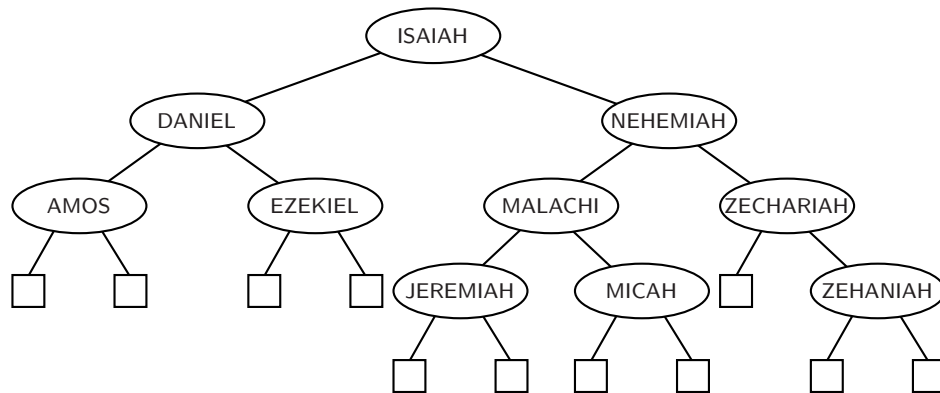
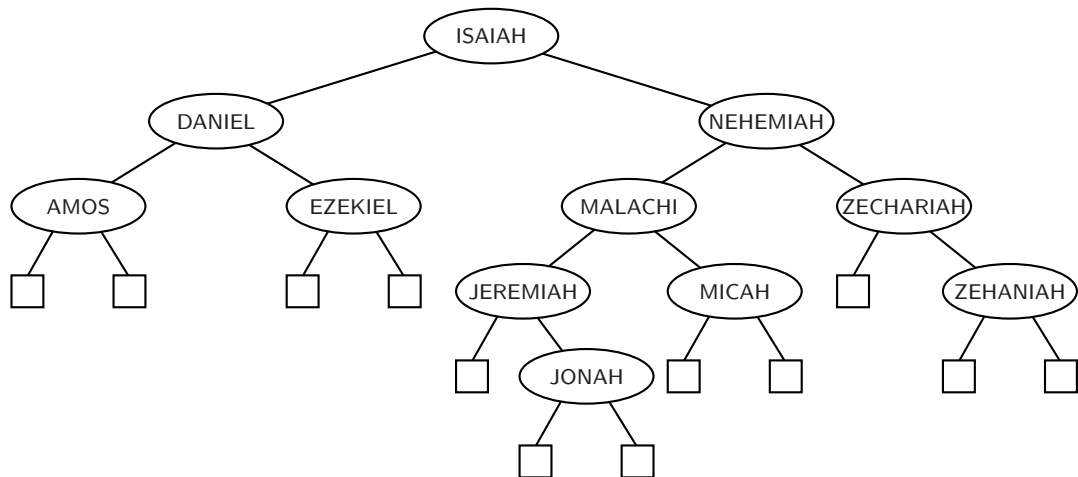


Figure 14.14 Building an AVL tree (continued on next page)

Inserting `JEREMIAH`, `ZEPHANIAH` and `MICAH` maintains the AVL property.



Finally, inserting `JONAH` causes the entire BST to become non-AVL; performing



a *right-left rotation* (right at `NEHEMIAH`, left at `ISAIAH`) restores the AVL property.

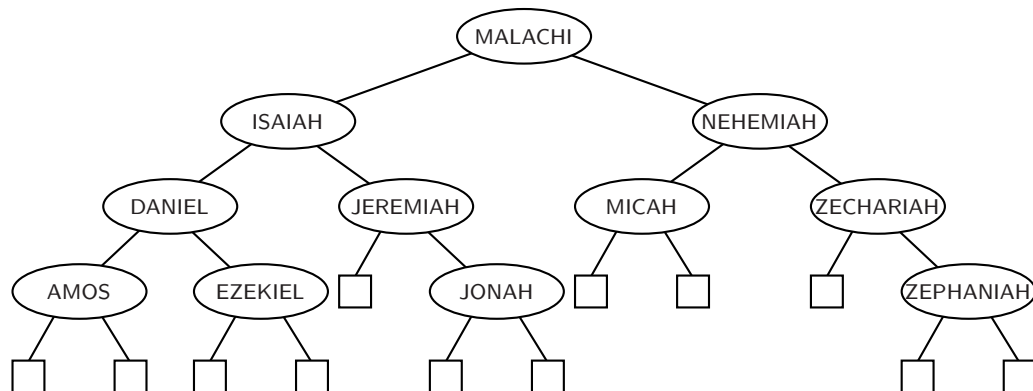


Figure 14.14 Building an AVL tree

This example should have amply demonstrated what we have earlier pointed out, to wit: when the BST loses the AVL property after an insert operation, it takes *only one* type of rotation, applied on the *smallest* subtree that has become non-AVL, to convert the *entire* BST into an AVL tree once again.

EASY implementation of the insert operation on an AVL tree

The EASY procedure AVL_INSERT implements the insert operation on an AVL tree. The input to the procedure is an AVL tree, represented by the data structure $\mathbb{B} = [T, (BF, LSON, KEY, DATA, RSON)]$, and the record (K, d) to be inserted into the BST. The first component of \mathbb{B} contains the address of the root node of the BST; $T = \Lambda$ means the BST is empty. The second component defines the node structure of each node in the BST; in particular, BF is the balance factor and is set to 0, +1 or -1 to denote '.', '+', and '-', respectively.

Several pointer variables are used by the procedure. One, in particular, needs special mention. This is the pointer variable α which points to the node where rebalancing may have to be done. Initially, α points to the root node (whose father is unknown); subsequently, it points to the *last node* in the path from the root to where the new node is inserted *whose balance factor is either +1 or -1, if such a node exists*. In this case, the variable ϕ points to the father of node α .

```

1  procedure AVL_INSERT( $\mathbb{B}, K, d$ )
 $\triangleright$  Check if BST is empty
2  if  $T = \Lambda$  then [ call GETNODE( $\nu$ );  $T \leftarrow \nu$ ;  $KEY(\nu) \leftarrow K$ ;  $DATA(\nu) \leftarrow d$ 
3                       $LSON(\nu) \leftarrow RSON(\nu) \leftarrow \Lambda$ ;  $BF(\nu) \leftarrow 0$ ; return ]
 $\triangleright$  Insert new node into non-null BST
4   $\alpha \leftarrow \gamma \leftarrow T$ 
5  loop
6    case
7      : $K = KEY(\gamma)$ : stop  $\triangleright$  duplicate key
8      : $K < KEY(\gamma)$ : [ $\tau \leftarrow LSON(\gamma)$ 
9                    if  $\tau = \Lambda$  then [ call GETNODE( $\nu$ );  $LSON(\gamma) \leftarrow \nu$ ; goto 1 ] ]
10     : $K > KEY(\gamma)$ : [ $\tau \leftarrow RSON(\gamma)$ 
11                   if  $\tau = \Lambda$  then [ call GETNODE( $\nu$ );  $RSON(\gamma) \leftarrow \nu$ ; goto 1 ] ]
12     endcase
13     if  $BF(\tau) \neq 0$  then [ $\alpha \leftarrow \tau$ ;  $\phi \leftarrow \gamma$ ]  $\triangleright \phi$  is father of new  $\alpha$ 
14      $\gamma \leftarrow \tau$ 
15  forever
 $\triangleright$  Fill in fields of newly inserted node
16 1:  $KEY(\nu) \leftarrow K$ ;  $DATA(\nu) \leftarrow d$ ;  $LSON(\nu) \leftarrow RSON(\nu) \leftarrow \Lambda$ ;  $BF(\nu) \leftarrow 0$ 
 $\triangleright$  Adjust balance factors of nodes in the path from node  $\alpha$  to node  $\nu$ 
17  if  $K < KEY(\alpha)$  then  $\beta \leftarrow \gamma \leftarrow LSON(\alpha)$ 
18      else  $\beta \leftarrow \gamma \leftarrow RSON(\alpha)$ 
19  while  $\gamma \neq \nu$  do
20      if  $K < KEY(\gamma)$  then [ $BF(\gamma) \leftarrow -1$ ;  $\gamma \leftarrow LSON(\gamma)$ ]
21      else [ $BF(\gamma) \leftarrow +1$ ;  $\gamma \leftarrow RSON(\gamma)$ ]
22  endwhile
 $\triangleright$  Rebalance subtree rooted at node  $\alpha$ , if necessary
23  if  $K < KEY(\alpha)$  then  $w = -1$   $\triangleright$  New node inserted into left subtree of node  $\alpha$ 
24      else  $w = +1$   $\triangleright$  New node inserted into right subtree of node  $\alpha$ 

```

Procedure 14.5 Inserting a new node into an AVL tree (continued on next page)

```

25  case
26      :  $BF(\alpha) = 0$  :  $BF(\alpha) = w \triangleright$  node  $\alpha$  is root of entire BST; BST still AVL
27      :  $BF(\alpha) = -w$  :  $BF(\alpha) = 0 \triangleright$  subtree rooted at node  $\alpha$  became perfectly balanced
28      :  $BF(\alpha) = w$  :  $\triangleright$  subtree rooted at node  $\alpha$  became unbalanced
29      [ case
30          :  $K < KEY(\alpha)$  and  $BF(\beta) = -w$  : call ROTATE_RIGHT( $\mathbb{B}, \alpha, \rho$ )
31          :  $K < KEY(\alpha)$  and  $BF(\beta) = w$  : call ROTATE_LEFT_RIGHT( $\mathbb{B}, \alpha, \rho$ )
32          :  $K > KEY(\alpha)$  and  $BF(\beta) = w$  : call ROTATE_LEFT( $\mathbb{B}, \alpha, \rho$ )
33          :  $K > KEY(\alpha)$  and  $BF(\beta) = -w$  : call ROTATE_RIGHT_LEFT( $\mathbb{B}, \alpha, \rho$ )
34      endcase
35      case
36          :  $\alpha = T$  :  $T \leftarrow \rho$ 
37          :  $\alpha = LSON(\phi)$  :  $LSON(\phi) \leftarrow \rho$ 
38          :  $\alpha = RSON(\phi)$  :  $RSON(\phi) \leftarrow \rho$ 
39      endcase ]
40  endcase
41  end AVL_INSERT

1  procedure ROTATE_RIGHT( $\mathbb{B}, \alpha, \beta$ )
2       $\beta \leftarrow LSON(\alpha) \triangleright$  new root of rebalanced tree
3       $LSON(\alpha) \leftarrow RSON(\beta)$ 
4       $RSON(\beta) \leftarrow \alpha$ 
5       $BF(\alpha) \leftarrow BF(\beta) \leftarrow 0$ 
6  end ROTATE_RIGHT

1  procedure ROTATE_LEFT( $\mathbb{B}, \alpha, \beta$ )
2       $\beta \leftarrow RSON(\alpha) \triangleright$  new root of rebalanced tree
3       $RSON(\alpha) \leftarrow LSON(\beta)$ 
4       $LSON(\beta) \leftarrow \alpha$ 
5       $BF(\alpha) \leftarrow BF(\beta) \leftarrow 0$ 
6  end ROTATE_LEFT

1  procedure ROTATE_LEFT_RIGHT( $\mathbb{B}, \alpha, \rho$ )
2       $\beta \leftarrow LSON(\alpha)$ 
3       $\rho \leftarrow RSON(\beta) \triangleright$  new root of rebalanced tree
4       $RSON(\beta) \leftarrow LSON(\rho)$ 
5       $LSON(\rho) \leftarrow \beta$ 
6       $LSON(\alpha) \leftarrow RSON(\rho)$ 
7       $RSON(\rho) \leftarrow \alpha$ 
8  case
9      :  $BF(\rho) = 0$  : [ $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow 0$ ]
10     :  $BF(\rho) = +1$  : [ $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow -1$ ]
11     :  $BF(\rho) = -1$  : [ $BF(\alpha) \leftarrow +1$ ;  $BF(\beta) \leftarrow 0$ ]
12  endcase
13   $BF(\rho) \leftarrow 0$ 
14  end ROTATE_LEFT_RIGHT

```

Procedure 14.5 Inserting a new node into an AVL tree (continued on next page)

```

1  procedure ROTATE_RIGHT_LEFT( $\mathbb{B}, \alpha, \rho$ )
2     $\beta \leftarrow RSON(\alpha)$ 
3     $\rho \leftarrow LSON(\beta)$   $\triangleright$  new root of rebalanced tree
4     $LSON(\beta) \leftarrow RSON(\rho)$ 
5     $RSON(\rho) \leftarrow \beta$ 
6     $RSON(\alpha) \leftarrow LSON(\rho)$ 
7     $LSON(\rho) \leftarrow \alpha$ 
8    case
9      :  $BF(\rho) = 0$  : [ $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow 0$ ]
10     :  $BF(\rho) = +1$  : [ $BF(\alpha) \leftarrow -1$ ;  $BF(\beta) \leftarrow 0$ ]
11     :  $BF(\rho) = -1$  : [ $BF(\alpha) \leftarrow 0$ ;  $BF(\beta) \leftarrow +1$ ]
12  endcase
13     $BF(\rho) \leftarrow 0$ 
14  end ROTATE_RIGHT_LEFT

```

Procedure 14.5 Inserting a new node into an AVL tree

Lines 2–3 handle the trivial case in which insertion is into a null BST; the result is a one node AVL tree which obviously is perfectly balanced, so nothing more needs to be done. Lines 4–15 handle the case in which insertion is into an existing BST; in this case a search is made through the BST with K as the search key. If the search is successful, the insert operation is aborted with the message ‘Duplicate key found.’ Otherwise, a new node pointed to by ν is inserted where the search terminated unsuccessfully, and the loop is exited.

In line 13, the value of α is updated whenever a node is encountered during the search whose balance factor is $+1$ or -1 ; the subtree rooted at such a node may become unbalanced after the insert operation. The address of the father of node α is recorded in ϕ so that later node ϕ can be made to point to whichever node replaces α in the rebalanced subtree.

In lines 17–22, the balance factor of each node in the path between node α and node ν , which is zero, is adjusted depending on whether the new node is inserted into its left or its right subtree. Finally, in lines 23–40 the subtree rooted at α is checked to determine whether it still satisfies the AVL property or not. One of three scenarios may obtain.

- (a) The balance factor of node α is zero, which means node α is the root of the entire BST. Thus the BST is still AVL, with the root having a balance factor of $+1$ if the new node were inserted into its right subtree or -1 if into its left subtree.
- (b) The balance factor of node α is -1 and the new node is inserted into its right subtree, or the balance factor of node α is $+1$ and the new node is inserted into its left subtree; in such a case, the subtree rooted at α becomes perfectly balanced.
- (c) The balance factor of node α is -1 and the new node is inserted into its left subtree, or the balance factor of node α is $+1$ and the new node is inserted into its right subtree. This time one of the four cases depicted in Figures 14.10 through 14.13 arises, and the subtree rooted at α is rebalanced by performing the appropriate rotation (lines 30–33); the pointer variable ρ points to the root of the rebalanced tree

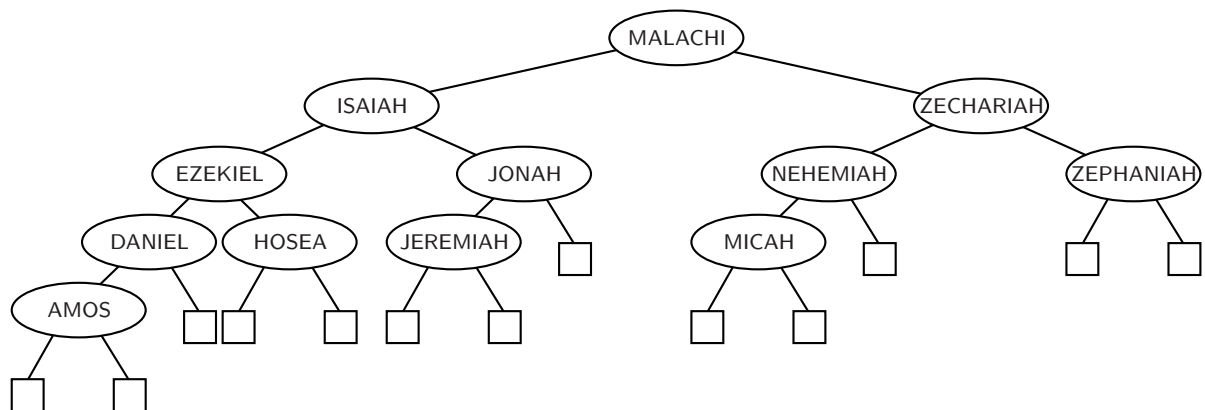
or subtree. In line 36, the condition $\alpha = T$ means the entire BST was rebalanced, and so ρ becomes the root of the BST. Otherwise, ρ becomes the left son or right son of the father of node α , viz., node ϕ , according as to whether node α was originally a left son or a right son of its father (lines 37–38).

It is clear from the procedures which implement the four types of rotations that rebalancing an AVL tree after an insert operation takes $O(1)$ time. Hence the time complexity of `AVL_INSERT` is dominated by the insert operation itself which takes $O(h)$ time. Since the height of an AVL tree is $O(\log n)$, it follows that the time complexity of the procedure is $O(\log n)$.

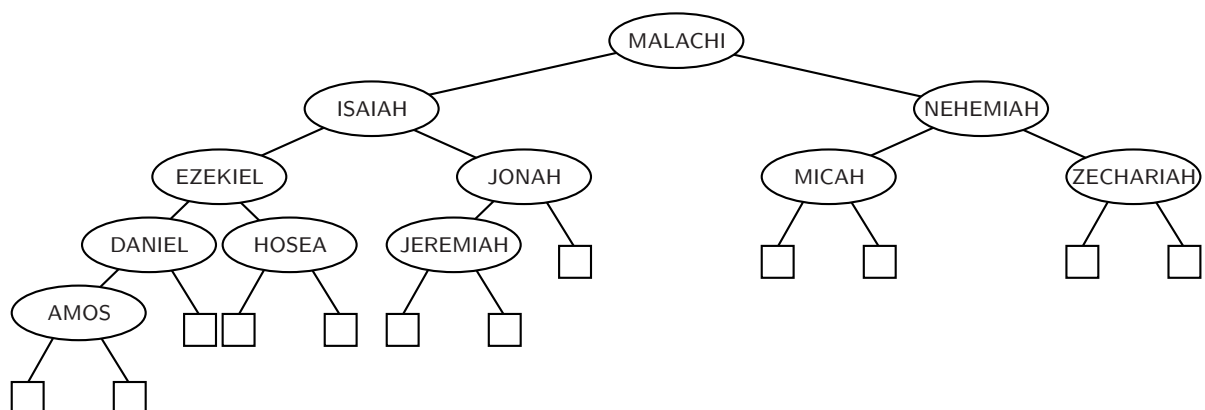
Deletions in AVL trees

We have seen above that when a BST becomes non-AVL after an insertion, it is made AVL once more in $O(1)$ time by applying one of four types of rotations. However, when a BST becomes non-AVL after a deletion, it may take $O(\log n)$ rotations to convert it back to AVL, taking $O(\log n)$ time. For instance, if the rightmost node of a Fibonacci tree (a worst-case AVL tree) is deleted, the tree becomes non-AVL and successive rotations extending all the way to the root of the entire tree must be applied to restore the AVL property.

To illustrate, consider the worst-case AVL tree shown below in which the left subtree of every node is consistently higher by one than the right subtree.



Now, if node `[[ZEPHANIAH]]` is deleted, then the subtree rooted at `[[ZECHARIAH]]` becomes non-AVL. This subtree is made AVL again by performing a right rotation at `[[ZECHARIAH]]` yielding the BST shown below.



This time it is the whole BST which is non-AVL. It is converted to AVL by performing a right rotation at $\llbracket \text{MALACHI} \rrbracket$ to give, finally

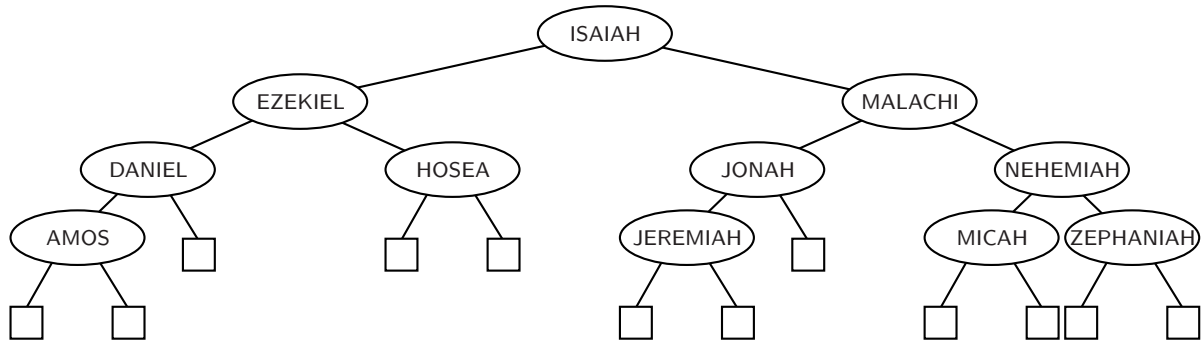


Figure 14.15 Deleting a node from an AVL tree

14.4 Optimum binary search trees

We have seen in the preceding section that if each key in a BST is equally likely as a search key, then a BST with minimum height minimizes the average number of comparisons, or average search time, in successful searches. However, when keys are no longer accessed with equal probability, a BST with minimum height no longer necessarily minimizes the average search time. As a simple example, suppose we have the keys $\{\text{HOSEA}, \text{ISAIAH}, \text{JONAH}\}$. Now let p_1 , p_2 and p_3 be the probability that we are looking for HOSEA, ISAIAH and JONAH, respectively. If $p_1 = p_2 = p_3 = 1/3$, then the BST of Figure 14.16(a) minimizes the average search time. However, if $p_1 = 1/8$, $p_2 = 2/8$ and $p_3 = 5/8$, then the BST of Figure 14.16(b) minimizes the average search time in a successful search, that its height is maximum notwithstanding.

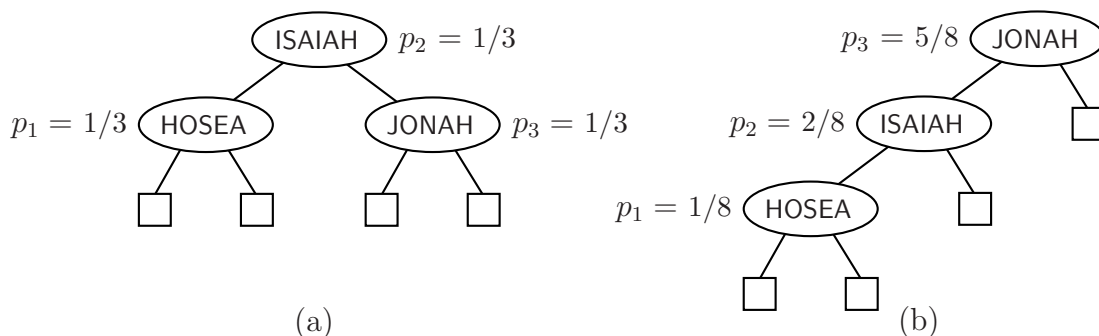


Figure 14.16 Two optimum binary search trees (successful searches only)

In general, we should also try to minimize the average search time in unsuccessful searches, in addition to minimizing the average search time in successful searches. This means that we also have to consider the probability that the search key K is less than the smallest key, or lies between keys in the table or is greater than the largest key. Specifically, let

p_i be the probability that $K = k_i$

q_0 be the probability that $K < k_1$

q_i be the probability that $k_i < K < k_{i+1}$, $i = 1, 2, \dots, n - 1$

q_n be the probability that $K > k_n$

where $p_1 + p_2 + \dots + p_n + q_0 + q_1 + \dots + q_n = 1$. Figure 14.17 depicts these quantities for a BST on the keys $\{k_1, k_2, k_3, k_4\} = \{\text{HOSEA}, \text{ISAIAH}, \text{JONAH}, \text{MICAH}\}$. Note that the probabilities q_i , $0 \leq i \leq 4$, are assigned to the external nodes from left to right to conform to the definition given above. Thus, q_0 is the probability that the search key is less than HOSEA, q_1 is the probability that the search key is greater than HOSEA and less than ISAIAH (e.g., ISAAC), q_2 is the probability that the search key is greater than ISAIAH and less than JONAH (e.g., JOEL), and so on, and q_4 is the probability that the search key is greater than MICAH.

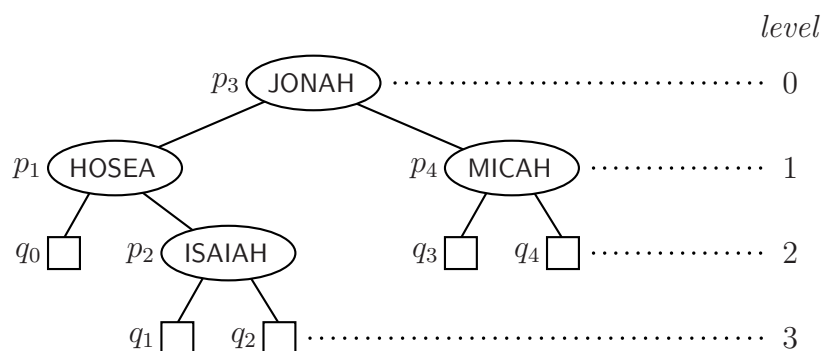


Figure 14.17 A BST with probabilities on the nodes

Referring to Figure 14.17, we note that the number of comparisons performed in a successful search terminating in an internal node at level l is $l + 1$, and the number of comparisons performed in an unsuccessful search terminating in an external node at level l is l . Hence, the *expected number of comparisons* for the BST of Figure 14.17 is $p_3 + 2p_1 + 2p_4 + 3p_2 + 2q_0 + 2q_3 + 2q_4 + 3q_1 + 3q_2$. Another way of looking at this sum is as the *cost* of the BST. In general, then, we have

$$\text{cost} = \sum_{i=1}^n p_i (l_i + 1) + \sum_{i=0}^n q_i l_i \quad (14.16)$$

where l_i is the level of node i in the BST. A BST with minimum cost minimizes the average search time for both successful and unsuccessful searches and is called an **optimum binary search tree**.

Figure 14.18 depicts the five distinct BST's on three distinct keys with probabilities attached to each internal and external node. The cost of each BST, computed according to Eq.(14.16) is as indicated. For the given probabilities the BST in (b) is optimum.

We noted earlier that there are b_n distinct BST's on n distinct keys. Since b_n increases exponentially with n (cf. Eq.(6.4) and Figure 6.3 in Session 6), finding an optimum BST by exhaustive search (i.e., by constructing all possible BST's on a given set of keys,

computing the cost of each and choosing one with minimum cost) as we have done in Figure 14.18, is not a feasible approach. How then do we find an optimum BST?

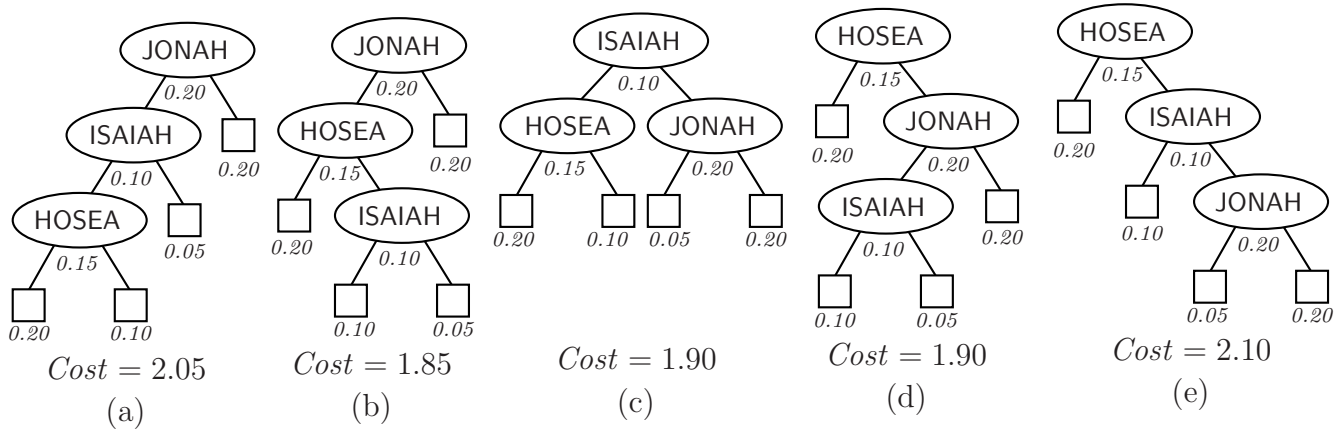


Figure 14.18 Finding an optimum BST by exhaustive search

The key to finding an optimum BST turns out to be quite simple: *all subtrees of an optimum BST are optimum*. Suppose that T is an optimum BST; then all its subtrees must be optimum. Assume, to the contrary, that some subtree in T , say on the keys k_i, k_{i+1}, \dots, k_j , is not optimum. Then, we can replace this non-optimum subtree with an optimum one on the same subset of keys resulting in a BST with a lower cost, contradicting the assumption that T is an optimum BST.

The idea, then, is to construct larger and larger optimum subtrees on larger and larger subsets of keys until all n keys are accounted for, using an algorithmic technique called *dynamic programming*. In Session 10 we saw an application of this technique in Floyd's algorithm to find the shortest paths between every pair of vertices in a weighted digraph.

Finding an optimum BST

Given the keys $k_1 < k_2 < k_3 < \dots < k_n$ with probabilities $p_1, p_2, p_3, \dots, p_n, q_0, q_1, q_2, \dots, q_n$, we want to find a BST whose cost, as given by Eq.(14.16), is minimum. The algorithm to find an optimum BST, based on the dynamic programming technique, consists of n passes in which we construct larger and larger optimum subtrees until, at the n th pass, we find an optimum BST on the given set of n keys. In pass 1, we construct n optimum subtrees of size 1 on the keys $k_1..k_1, k_2..k_2, \dots, k_n..k_n$. In pass 2, we construct $n - 1$ optimum subtrees of size 2 on the keys $k_1..k_2, k_2..k_3, \dots, k_{n-1}..k_n$. In pass 3, we construct $n - 2$ optimum subtrees of size 3 on the keys $k_1..k_3, k_2..k_4, \dots, k_{n-2}..k_n$. And so on. Finally in pass n , we construct an optimum tree on all the keys $k_1..k_n$.

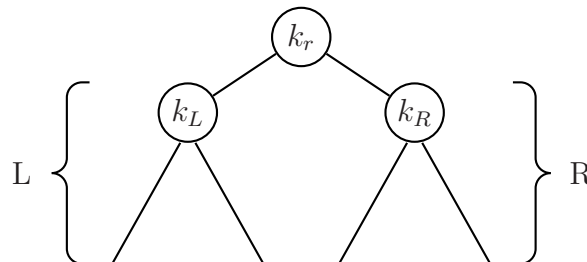


Figure 14.19 A candidate optimum subtree on the keys $k_{i+1}, k_{i+2}, \dots, k_j$

Let c_{ij} be the cost of an optimum BST on the keys $k_{i+1}, k_{i+2}, \dots, k_j$ with probabilities $p_{i+1}, p_{i+2}, \dots, p_j, q_i, q_{i+1}, \dots, q_j$. To obtain an expression for c_{ij} consider the candidate optimum subtree shown in Figure 14.19. There are $j - i$ such candidate trees with roots $\llbracket k_{i+1} \rrbracket, \llbracket k_{i+2} \rrbracket, \dots, \llbracket k_j \rrbracket$. We take note of the following facts about any such tree.

- (a) The root node is $\llbracket k_r \rrbracket$ where $i + 1 \leq r \leq j$.
- (b) L is an *optimum* subtree on the keys k_{i+1}, \dots, k_{r-1} with cost $c_{i,r-1}$. If $r = i + 1$ then L is empty (contains no key) with cost $c_{ii} = 0$.
- (c) R is an *optimum* subtree on the keys k_{r+1}, \dots, k_j with cost c_{rj} . If $r = j$ then R is empty (contains no key) with cost $c_{jj} = 0$.

On the basis of these facts and using Eq.(14.16) we obtain the equation

$$\begin{aligned}
 c_{ij} &= p_r + \min_{i+1 \leq r \leq j} (c_{i,r-1} + c_{rj}) + \underbrace{\sum_{l=i+1}^{r-1} p_l + \sum_{l=i}^{r-1} q_l}_{\substack{\text{since every node in L is one level} \\ \text{deeper from } (k_r) \\ \text{than from } (k_L)}} + \underbrace{\sum_{l=r+1}^j p_l + \sum_{l=r}^j q_l}_{\substack{\text{since every node in R is one level} \\ \text{deeper from } (k_r) \\ \text{than from } (k_R)}} \\
 &\quad \begin{array}{l} \uparrow \\ \text{cost of R relative to its root } (k_R) \\ \uparrow \\ \text{cost of L relative to its root } (k_L) \\ \uparrow \\ \text{cost of root } (k_r) \end{array} \\
 &= \underbrace{\sum_{l=i+1}^j p_l + \sum_{l=i}^j q_l}_{w_{ij}} + \min_{i+1 \leq r \leq j} (c_{i,r-1} + c_{rj}) \\
 &= w_{ij} + \min_{i+1 \leq r \leq j} (c_{i,r-1} + c_{rj}), \quad i < j
 \end{aligned} \tag{14.17}$$

In standard dynamic programming fashion, the costs c_{ij} of larger and larger optimum subtrees found are saved in a table for immediate access when yet larger subtrees which contain these are constructed. Figure 14.20 shows a template for such a table for $n = 6$. The last entry in the table, c_{06} , is the cost of the optimum BST found on the entire set of keys k_1, k_2, \dots, k_6 . To recover this optimum BST we record the value of r which minimizes the quantity $c_{i,r-1} + c_{rj}$ in Eq.(14.17); these are the entries ρ_{ij} in the table. The entries in each cell of Figure 14.20 mean precisely this: ρ_{ij} is the root of the optimum BST on the keys k_{i+1}, \dots, k_j with probabilities $p_{i+1}, \dots, p_j, q_i, \dots, q_j$; the cost of this BST is c_{ij} .

Pass												
1	$k_1..k_1$		$k_2..k_2$		$k_3..k_3$		$k_4..k_4$		$k_5..k_5$		$k_6..k_6$	
	c_{01}	ρ_{01}	c_{12}	ρ_{12}	c_{23}	ρ_{23}	c_{34}	ρ_{34}	c_{45}	ρ_{45}	c_{56}	ρ_{56}
2	$k_1..k_2$		$k_2..k_3$		$k_3..k_4$		$k_4..k_5$		$k_5..k_6$			
	c_{02}	ρ_{02}	c_{13}	ρ_{13}	c_{24}	ρ_{24}	c_{35}	ρ_{35}	c_{46}	ρ_{46}		
3	$k_1..k_3$		$k_2..k_4$		$k_3..k_5$		$k_4..k_6$					
	c_{03}	ρ_{03}	c_{14}	ρ_{14}	c_{25}	ρ_{25}	c_{36}	ρ_{36}				
4	$k_1..k_4$		$k_2..k_5$		$k_3..k_6$							
	c_{04}	ρ_{04}	c_{15}	ρ_{15}	c_{26}	ρ_{26}						
5	$k_1..k_5$		$k_2..k_6$									
	c_{05}	ρ_{05}	c_{16}	ρ_{16}								
6	$k_1..k_6$											
	c_{06}	ρ_{06}										

Figure 14.20 Template for the computations to find an optimum BST

The following algorithm formalizes the process described above.

Algorithm OPTIMUM_BST

1. [Initializations] Set $c_{ii} \leftarrow 0$ and $w_{ii} \leftarrow q_i$ for $0 \leq i \leq n$
2. [Loop on size of subtrees] Do step 3 for $s = 1, 2, \dots, n$; then stop.
3. [Loop on subrange of keys] Do step 4 for $i = 0, 1, \dots, n - s$
4. [Find c_{ij} and ρ_{ij}] Set $j \leftarrow i + s$; then set

$$w_{ij} \leftarrow w_{i,j-1} + p_j + q_j$$

$$c_{ij} \leftarrow w_{ij} + \min_{i+1 \leq r \leq j} (c_{i,r-1} + c_{rj})$$

and set ρ_{ij} to the value of r for which the minimum occurs.

Upon termination of the algorithm the optimum BST found is encoded in the ρ array; its cost is c_{0n} .

EASY implementation of algorithm OPTIMUM_BST

Procedure OPTIMUM_BST(\mathbb{B}) is a straightforward implementation of the above algorithm. The procedure accepts a pointer to the data structure $\mathbb{B} = [n, \text{key}(1:n), p(1:n), q(0:n), c(0:n, 0:n), \text{root}(0:n, 0:n)]$, allowing access to its components. The first four components of \mathbb{B} comprise the input; the last two components comprise the output. The c and root arrays, together, represent the table shown in Figure 14.20.

```

1  procedure OPTIMUM_BST( $\mathbb{B}$ )
2  array  $w(0:n, 0:n)$ 
3  for  $i \leftarrow 0$  to  $n$  do
4       $c(i, i) \leftarrow 0$ 
5       $w(i, i) \leftarrow q(i)$ 
6  endfor
7  for  $s \leftarrow 1$  to  $n$  do
8      for  $i \leftarrow 0$  to  $n - s$  do
9           $j \leftarrow i + s$ 
10          $w(i, j) \leftarrow w(i, j - 1) + p(j) + q(j)$ 
11          $c(i, j) \leftarrow \infty$ 
12         for  $r \leftarrow i + 1$  to  $j$  do
13              $cost \leftarrow w(i, j) + c(i, r - 1) + c(r, j)$ 
14             if  $cost < c(i, j)$  then [ $c(i, j) \leftarrow cost$ ;  $root(i, j) \leftarrow r$ ]
15         endfor
16     endfor
17 endfor
18 end OPTIMUM_BST

```

Procedure 14.6 Finding an optimum BST*Example 14.2.* Finding an optimum BST

To illustrate how the algorithm OPTIMUM_BST works, assume that we are given the following input:

$$\begin{aligned}
 key(1 : 8) &= (\text{DANIEL}, \text{EZEKIEL}, \text{HOSEA}, \text{ISAIAH}, \text{JONAH}, \text{MICAH}, \text{NEHEMIAH}, \text{ZECHARIAH}) \\
 p(1 : 8) &= (0.10, 0.06, 0.08, 0.04, 0.02, 0.09, 0.12, 0.08) \\
 q(0 : 8) &= (0.04, 0.03, 0.05, 0.09, 0.03, 0.06, 0.05, 0.02, 0.04)
 \end{aligned}$$

Figure 14.21 shows the results of the computations performed by procedure OPTIMUM_BST for this input. For readability, the roots of the optimum subtrees found in each pass of the algorithm are identified by the name of the key instead of its index, i.e., what is shown is $key(\rho_{ij})$ instead of ρ_{ij} . The optimum BST found is rooted at **MICAH** and has a cost of 2.82.

To gain a clearer understanding of the way the dynamic programming technique succeeds in finding an optimum BST, consider the computations performed in pass 4 of the algorithm. At this point in the computations, optimum subtrees on subranges of 1, 2 and 3 keys have already been found and saved in the table. Now we consider subranges of 4 keys, for instance, **HOS..MIC** or $k_3..k_6$. We find that $w_{26} = p_3 + p_4 + p_5 + p_6 + q_2 + q_3 + q_4 + q_5 + q_6 = 0.08 + 0.04 + 0.02 + 0.09 + 0.05 + 0.09 + 0.03 + 0.06 + 0.05 = 0.51$. To find an optimum subtree on the keys **HOS..MIC**, we place each of these keys at the root and compute the cost of the resulting BST. For instance, if we place **HOSEA** at the root, then the left subtree is empty with cost $c_{22} = 0$ and the right subtree is the optimum subtree on the keys **ISA..MIC** found in pass 3 with cost $c_{36} = 0.73$. Thus the cost of the subtree rooted at **HOSEA** is $0.51 + 0 + 0.73 = 1.26$.

Pass													
1	DAN..DAN	EZE..EZE	HOS..HOS	ISA..ISA	JON..JON	MIC..MIC	NEH..NEH	ZEC..ZEC					
	0.17 DAN	0.14 EZE	0.22 HOS	0.16 ISA	0.11 JON	0.20 MIC	0.19 NEH	0.14 ZEC					
2	DAN..EZE	EZE..HOS	HOS..ISA	ISA..JON	JON..MIC	MIC..NEH	NEH..ZEC						
	0.42 DAN	0.45 HOS	0.45 HOS	0.35 ISA	0.36 MIC	0.53 MIC	0.45 NEH						
3	DAN..HOS	EZE..ISA	HOS..JON	ISA..MIC	JON..NEH	MIC..ZEC							
	0.84 EZE	0.68 HOS	0.70 ISA	0.73 MIC	0.69 MIC	0.80 NEH							
4	DAN..ISA	EZE..JON	HOS..MIC	ISA..NEH	JON..ZEC								
	1.10 HOS	0.95 HOS	1.09 ISA	1.06 MIC	1.01 NEH								
5	DAN..JON	EZE..MIC	HOS..NEH	ISA..ZEC									
	1.37 HOS	1.41 ISA	1.54 MIC	1.44 MIC									
6	DAN..MIC	EZE..NEH	HOS..ZEC										
	1.89 HOS	1.88 ISA	1.92 MIC										
7	DAN..NEH	EZE..ZEC											
	2.36 HOS	2.26 MIC											
8	DAN..ZEC												
	2.82 MIC												

Figure 14.21 Summary of computations for the given input

Next, we place **ISAIAH** at the root. This time the left subtree is the optimum subtree on the key **HOS..HOS** found in pass 1 with cost $c_{23} = 0.22$ and the right subtree is the optimum subtree on the keys **JON..MIC** found in pass 2 with cost $c_{46} = 0.36$. Thus the cost of the subtree rooted at **ISAIAH** is $0.51 + 0.22 + 0.36 = 1.09$. Similarly, we find that the subtree rooted at **JONAH** and **MICAH** have costs of 1.16 and 1.21, respectively, as shown in Figure 14.22. Hence the optimum subtree on the keys **HOS..MIC** is rooted at **ISAIAH** with a cost of 1.09. These are the entries in row 4, column 3 of Figure 14.21; in procedure **OPTIMUM_BST** these results are recorded by setting $c(2, 6)$ to 1.09 and $root(2, 6)$ to 4.

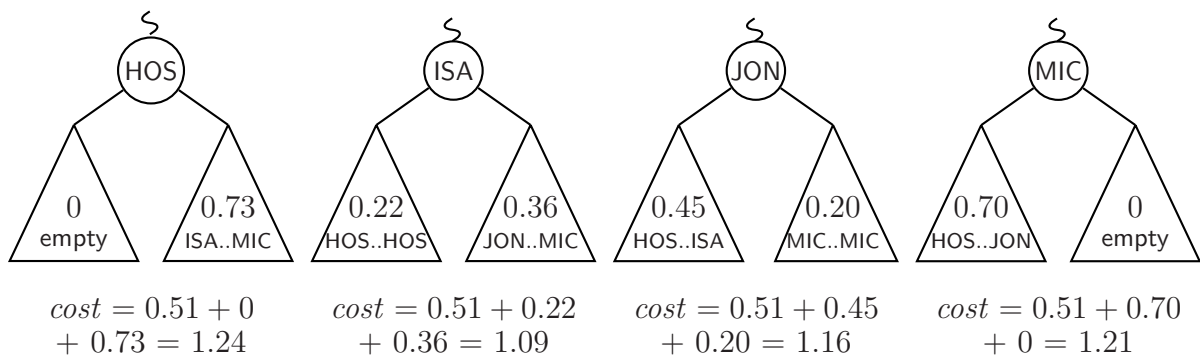


Figure 14.22 Candidate optimum subtrees on the keys **HOS..MIC** (pass 4)

Recovering the optimum BST

The optimum BST found by procedure OPTIMUM_BST is encoded in the *root* array. To recover this BST, let $t[i, j]$ denote the optimum BST on the keys $k_{i+1}, k_{i+2}, \dots, k_j$. Then, $t[i, j]$ is null if $i = j$; otherwise, its left subtree is $t[i, \text{root}(i, j) - 1]$ and its right subtree is $t[\text{root}(i, j), j]$. By this definition, the optimum BST on all the keys k_1, k_2, \dots, k_n is $t[0, n]$.

The recursive procedure DISPLAY_OPTIMUM_BST(\mathbb{B}, i, j) is a straightforward implementation of this recursive definition of $t[i, j]$. On first call, i must be set to zero and j to n . The output of the procedure is a preorder enumeration of the nodes of the BST, both internal and external (denoted NULL).

```

1  procedure DISPLAY_OPTIMUM_BST( $\mathbb{B}, i, j$ )
2    if  $i = j$  then [output 'NULL'; return]
3    else [output( $\mathbb{B}.\text{key}(\mathbb{B}.\text{root}(i, j))$ )]
4          call DISPLAY_OPTIMUM_BST( $\mathbb{B}, i, \mathbb{B}.\text{root}(i, j) - 1$ )
5          call DISPLAY_OPTIMUM_BST( $\mathbb{B}, \mathbb{B}.\text{root}(i, j), j$ )
6  end DISPLAY_OPTIMUM_BST

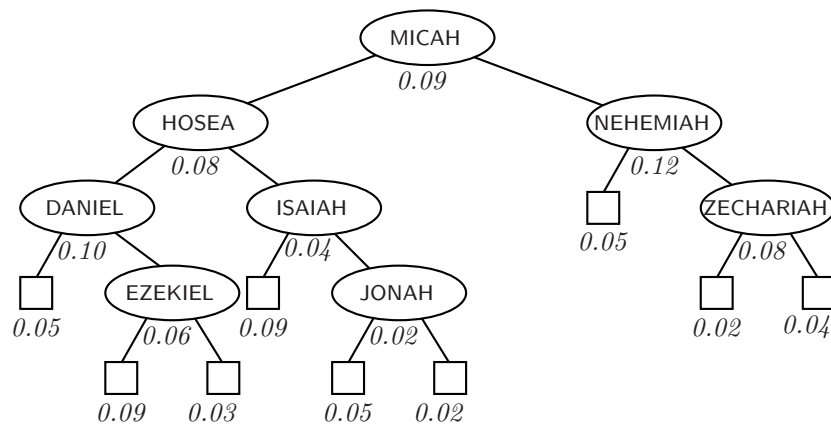
```

Procedure 14.7 Recovering the optimum BST encoded in the *root* array

Figure 14.23(a) shows the output generated by procedure DISPLAY_OPTIMUM_BST for the BST found by procedure OPTIMUM_BST. From this output, the actual BST can be readily constructed ‘by hand’, as shown in Figure 14.23(b). Alternatively, we may write a procedure which generates instead the internal representation of the optimum BST.

MICAH
HOSEA
DANIEL
NULL
EZEKIEL
NULL
NULL
ISAIAH
NULL
JONAH
NULL
NULL
NEHEMIAH
NULL
ZECHARIAH
NULL
NULL

(a)



Cost = 2.82

(b)

Figure 14.23 (a) Output of DISPLAY_OPTIMUM_BST (b) The optimum BST

Analysis of algorithm OPTIMUM_BST

To find the time complexity of algorithm OPTIMUM_BST as implemented by procedure OPTIMUM_BST we only need to count the number of times, say $T(n)$, that line 13 of the procedure is executed in which the cost of a candidate optimum BST is computed.

$$\begin{aligned}
 T(n) &= \sum_{s=1}^n \sum_{i=0}^{n-s} \sum_{r=i+1}^{i+s} 1 \\
 &= \sum_{s=1}^n \sum_{i=0}^{n-s} s \\
 &= \sum_{s=1}^n (n-s+1)s = (n+1) \sum_{s=1}^n s - \sum_{s=1}^n s^2 \\
 &= \frac{n(n+1)^2}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{n(n+1)(n+2)}{6} = O(n^3) \quad (14.18)
 \end{aligned}$$

Thus procedure OPTIMUM_BST executes in $O(n^2)$ space and $O(n^3)$ time. Knuth points out that it is not necessary to consider all $j-i$ candidate subtrees when finding the optimum subtree on the subrange $k_{i+1}, k_{i+2}, \dots, k_j$, as we have done. The resulting algorithm, Algorithm K (*Find optimum binary search trees*) given in KNUTH3[1998], p. 439, takes only $O(n^2)$ time.

Summary

- With the binary search tree as the implementing data structure for a dynamic table, the operations of insertion, deletion and searching for a given key, the minimum key or the maximum key are all realizable in $O(h)$ time. The predecessor (next smaller key) and successor (next larger key) operations can also be implemented in $O(h)$ time if in addition to the *LSON* and *RSN* fields we add a *FATHER* field in each node.
- $O(h)$ time is typically $O(\log n)$ time. However, $O(h)$ becomes $O(n)$ in a degenerate tree. Assuming equal probabilities (i.e., $p_1 = \dots = p_n, q_0 = q_1 = \dots = q_n$), the average search time in a BST is minimum when the external nodes of the BST lie on at most two adjacent levels (i.e., when the BST is completely balanced) and it is maximum when external nodes lie on every level. In the former case, BST search matches binary search; in the latter case, BST search degenerates into linear search.
- Keeping a BST completely balanced in a dynamic environment can be very costly. AVL trees, in which a little imbalance is permitted, provide a compromise solution. It does not take too much effort to maintain the required balance, although searches now take a little more time over the minimum.
- When the assumption of equal probabilities no longer applies a completely balanced BST no longer necessarily yields minimum search time. The optimum BST in this case is one that minimizes the expected number of comparisons in a search on the basis of the specified probabilities, and it can be found by applying the dynamic programming technique.

Exercises

1. Prove that

$$E = I + 2n$$

where I and E are the internal and external path lengths of an extended binary search tree on n internal nodes.

2. The 12 brightest stars in the sky are SIRIUS, CANOPUS, ALPHA CENTAURI, ARCTURUS, VEGA, CAPELLA, RIGEL, PROCYON, BETELGUESE, ACHERNAR, BETA CENTAURI and ALTAIR. Taking these names as alphabetic keys, show the extended binary search tree that results from inserting them, *as enumerated*, into an initially empty BST.
3. Assuming that each of the 12 names is equally likely to be a search key, calculate I , E , C_n and C'_n for the resulting BST in Item 1.
4. Assuming that each of the 12 names in Item 1 is equally likely to be a search key:
 - (a) construct a *minimal* BST on the keys and calculate the corresponding I , E , C_n and C'_n .
 - (b) construct a *maximal* BST on the keys and calculate the corresponding I , E , C_n and C'_n .
5. Show that there are 2^{n-1} distinct maximal BST's on n distinct keys.
6. Draw a worst-case AVL tree on 12 nodes using two-letter words as alphabetic keys.
7. Construct an AVL tree by inserting nodes for the two-letter words listed below, *as enumerated*, into an initially empty BST. Show the progress of the construction and indicate what rotation, if any, is performed at each step.

ON NO DO UP SO WE IN AM ME

8. Construct an AVL tree by inserting nodes for the instruction codes listed below, *as enumerated*, into an initially empty BST. Show the progress of the construction and indicate what rotation, if any, is performed at each step.

LA MR TM BE EX AR XI OI NI

9. Using a language of your choice, transcribe procedure AVL_INSERT (Procedure 14.5) into a running program. Test your program using the example in Figure 14.14.
10. Using your program in Item 9 solve Item 7 anew.
11. Using your program in Item 9 solve Item 8 anew.
12. Find an optimum BST on the keys $K_1 < K_2 < K_3 < K_4$ given the probabilities $(p_1, p_2, p_3, p_4) = (0.15, 0.10, 0.25, 0.05)$ for successful searches and $(q_0, q_1, q_2, q_3, q_4) = (0.15, 0.05, 0.10, 0.05, 0.10)$ for unsuccessful searches by:

- (a) exhaustive search
 - (b) applying Algorithm OPTIMUM_BST ‘by hand’
13. Using a language of your choice, transcribe procedures OPTIMUM_BST (Procedure 14.6) and DISPLAY_OPTIMUM_BST (Procedure 14.7) into a running program. Test your program using the example in Figures 14.21 and 14.23.
 14. Using your program in Item 13, find an optimum BST on the keys $(K_1, K_2, \dots, K_{10}) = (\text{AR}, \text{BC}, \text{CL}, \text{DR}, \text{EX}, \text{LA}, \text{MH}, \text{OI}, \text{ST}, \text{TM})$ for the following frequencies (which can also be used in place of probabilities):
 - (a) $(p_1, p_2, \dots, p_{10}) = (127, 236, 512, 185, 317, 618, 715, 427, 85, 267)$
 $(q_0, q_1, q_2, \dots, q_{10}) = (426, 257, 518, 133, 277, 490, 196, 648, 212, 344, 587)$
 - (b) $(p_1, p_2, \dots, p_{10}) = (127, 236, 512, 185, 317, 618, 715, 427, 85, 267)$
 $(q_0, q_1, q_2, \dots, q_{10}) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
 - (c) $(p_1, p_2, \dots, p_{10}) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
 $(q_0, q_1, q_2, \dots, q_{10}) = (426, 257, 518, 133, 277, 490, 196, 648, 212, 344, 587)$
 - (d) $(p_1, p_2, \dots, p_{10}) = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$
 $(q_0, q_1, q_2, \dots, q_{10}) = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$
 - (e) $(p_1, p_2, \dots, p_{10}) = (1, 2, 4, 8, 16, 32, 64, 128, 256, 512)$
 $(q_0, q_1, q_2, \dots, q_{10}) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

Bibliographic Notes

The algorithms for the basic operations on dynamic BST’s (search, insert, delete, find maximum/minimum key, predecessor/successor) are standard fare and can be found in KNUTH3[1998], CORMEN[2001], STANDISH[1994], among others. In particular, procedure BST_DELETE is an implementation of Algorithm D (*Tree Deletion*) given in KNUTH3[1998], p. 432, where a detailed discussion of other issues pertaining to deletions in BST’s can be found.

The analysis of BST search in the average BST, Eq.(14.13) to Eq.(14.15), which shows that the average BST is just about 38% worse than the best is from KNUTH3[1998], pp. 430–431. Standish (STANDISH[1980], p. 104) calls this result ‘remarkable’ and Knuth’s proof as ‘beautiful’. Eq.(14.9), which we have encountered earlier in the analysis of heapsort, is from KNUTH1[1997], p. 44.

AVL trees are discussed in KNUTH3[1998], STANDISH[1994], WEISS[1997], among others. Procedure AVL_INSERT is an implementation of Algorithm A (*Balanced tree search and insertion*) given in KNUTH3[1998], pp. 462–464, where a detailed analysis of the algorithm is also given. Knuth points out that a theoretical determination of the *average* behavior of the algorithm has proved to be quite difficult. Empirical tests, however, indicate that it takes approximately $1.01 \log_2 n + 0.1$ comparisons on the average to insert the n th item into an AVL tree, for sufficiently large n . For more interesting empirical results on the behavior of AVL trees, see KNUTH3[1998], pp. 466–471.

The idea of an optimum BST as one that minimizes Eq.(14.16), in which probabilities for both successful and unsuccessful searches are considered, was first formulated by

Knuth (KNUTH3[1998], pp. 436–442). Using the dynamic programming technique, he devised Algorithm K (*Find optimum binary search trees*) to find an optimum BST on n nodes, given the probabilities $p_1, p_2, \dots, p_n, q_0, q_1, q_2, \dots, q_n$, in $O(n^2)$ time and $O(n^2)$ space. Algorithm OPTIMUM_BST given above, which runs in $O(n^3)$ time, is a straightforward application of the dynamic programming technique in implementing Eq.(14.17), with hardly any mathematical intricacies. The mathematics which underlies Knuth's more efficient Algorithm K is more intricate. The student who has fully understood Algorithm OPTIMUM_BST may wish to tackle Algorithm K.

For the case in which only the probabilities on the external nodes are relevant, i.e., $p_1 = p_2 = \dots = p_n = 0$, there is available an even more efficient, but much more intricate, algorithm to find an optimum BST called the Garsia-Wachs algorithm. With suitable data structures, this algorithm executes in $O(n \log n)$ time (cf. KNUTH3[1998], pp. 446–453).

SESSION 15

Hash tables

OBJECTIVES At the end of the session, the student is expected to be able to:

1. Explain the basic ideas behind hashing and hash tables.
2. Characterize a good hash function.
3. Describe the division method and multiplicative method for hashing and explain how they must be implemented to yield a good hash function.
4. Explain why collisions are common even in sparsely occupied hash tables and describe how collisions are resolved by separate chaining and by open addressing.
5. Construct a hash table on a given set of keys using chaining, linear probing or double hashing to resolve collisions.
6. Explain how and why clustering occurs in open-address hash tables.
7. Explain important issues pertaining to deletions in open-address hash tables.
8. Construct an ordered open-address hash table on a given set of keys.
9. Explain how and why the search, insert and delete operations can be realized in constant time with hash tables.

READINGS KNUTH3[1998], pp. 513–539; CORMEN[2001], pp. 224–244; STANDISH[1994], pp. 450–496; STANDISH[1980], pp. 141–164.

DISCUSSION

In the implementations of the table ADT that we have considered so far, we always begin a search at a predefined position in the table. In linear search on a sequential table, we begin by comparing the search key with the key of the *first* record in the table. In binary search, we begin by comparing the search key with the key of the *middle* record in the table. In searching a BST we begin by comparing the search key with the key of the record at the *root* of the BST. In all these cases, where we start the search does not depend on what we are searching for. The novel idea behind a **hash table** is that we

start the search at a position in the table that depends on the search key; if indeed the record is in the table, we expect it to be only a few comparisons away.

The vehicle which allows us to implement this smart idea is a **hash function** h which maps any key, say k_j , from some universe of keys U_k to a position, or address, i in a hash table T , as depicted in Figure 15.1. We call the quantity $h(k_j)$ the *hash address* of the key k_j ; equivalently, we say that k_j hashes to the address $h(k_j)$ in T . In general, the set U_k is huge, for instance, the set of *all possible* identifier names in a programming language. In contrast, the table size m is relatively small, for instance, proportionate to the number of *actual* identifier names that are used in a program. A hash function maps a typically huge key space to a much smaller table space, thus:

$$h : U_k \rightarrow \{0, 1, 2, \dots, m-1\} \quad (15.1)$$

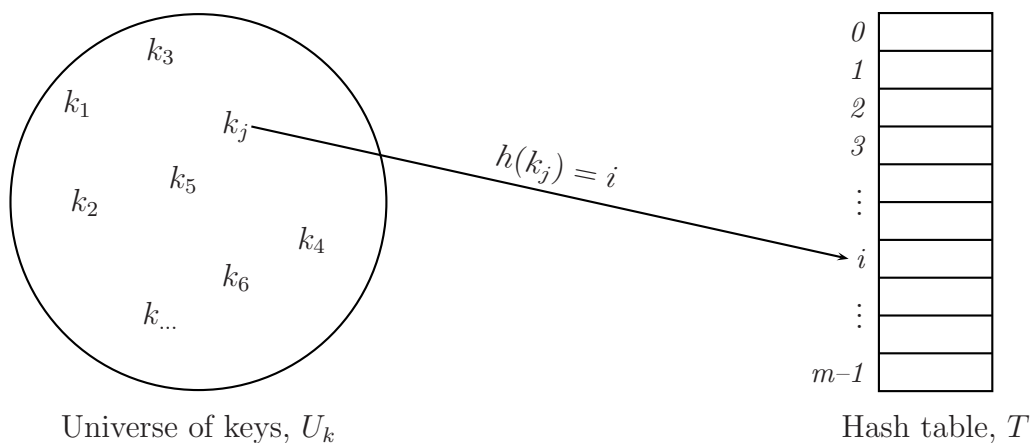


Figure 15.1 Hashing: a key from U_k is mapped to an address in T

Given some record (k_j, d_j) to be inserted into a hash table we compute $i = h(k_j)$, and if we find that cell i is vacant, we then store the record in cell i . Subsequently, when we want to access the same record, we again compute $i = h(k_j)$ and retrieve the record from cell i . There is however one hitch in this otherwise nifty procedure: we may find during insertion that cell i is already occupied. In such a case we have what is called a **collision**: two or more keys hash to the same address in T . Clearly, two records cannot occupy the same cell in the table; one, or both, will have to be stored elsewhere. How we go about doing this depends on the **collision resolution policy** that we adopt.

There are essentially three tasks that we must attend to when we use a hash table to implement the table ADT, namely:

1. Convert non-numeric keys to numbers
2. Choose a good hash function
3. Choose a collision resolution policy

In the rest of this session we will consider each of these tasks in turn.

15.1 Converting keys to numbers

In applying the hash function $h(K)$ we assume that K is a positive integer. In applications in which the keys are character strings, we must first convert each key to an equivalent integer such that *distinct keys are represented by distinct integers*. There are various ways in which we can perform such a conversion. To illustrate, suppose that the keys are strings composed of characters from the set $\{A, B, \dots, Z\}$. Specifically, assume that the key to be converted is **KEYS**; here are three different ways of obtaining a unique integer representation of the key.

1. Replace the letters by their position in the alphabet ($A = 01$, $B = 02$, and so on).

$$\text{KEYS} = 11052519$$

This means that the first, third, fifth, ... digit will be either a 0, 1 or 2, which is *not* a good idea.

2. Express the key as a radix-26 integer

$$\text{KEYS} = 11 \times 26^3 + 05 \times 26^2 + 25 \times 26^1 + 19 \times 26^0 = 197385$$

3. Express the key as a radix-256 integer with the letters represented by their ASCII codes ($A = 01000001_2 = 65_{10}$, $B = 01000010_2 = 66_{10}, \dots$, $Z = 01011010_2 = 90_{10}$)

$$\begin{aligned} \text{KEYS} &= 01001011\ 01000101\ 01011001\ 01010011_2 \\ &= 75 \times 256^3 + 69 \times 256^2 + 89 \times 256^1 + 83 \times 256^0 = 1262836051_{10} \end{aligned}$$

This conversion is particularly easy to implement since we simply interpret the 32-bit character string as a 32-bit binary number. If a key occupies several computer words, we can combine the words into a single word by addition modulo w , where w is the word size (e.g., 2^{32}), or by exclusive-or, denoted \oplus . To illustrate, suppose that the key to be converted is **AVERYLONGKEY**; using the latter method we obtain:

$$\begin{aligned} \text{GKEY} &= 01000111\ 01001011\ 01000101\ 01011001 \\ \text{YLON} &= 01011001\ 01001100\ 01001111\ 01001110 \\ \text{YLON} \oplus \text{GKEY} &= \overline{00011110\ 00000111\ 00001010\ 00010111} \\ \text{AVER} &= 01000001\ 01010110\ 01000101\ 01010010 \\ \text{AVER} \oplus \text{YLON} \oplus \text{GKEY} &= \overline{01011111\ 01010001\ 01001111\ 01000101} = 1599164229_{10} \end{aligned}$$

Hence we have $\text{AVERYLONGKEY} = 1599164229_{10}$. Note that the final result depends on *all* the bits of the original key. Likewise, regardless of the length of the key, no arithmetic overflow occurs when we combine the individual words by exclusive-or.

492 *SESSION 15. Hash tables*

In the rest of this session we will use as our sample ‘records’ those listed in Figure 15.2. Each record consists of a four-letter word as the key and a three-letter word as the satellite data.

1. FOLK	ART	16. FALL	GUY	31. EVIL	EYE	46. FARM	LAD
2. TEST	BIT	17. OPEN	AIR	32. BELL	JAR	47. TAXI	CAB
3. BACK	END	18. BALI	HAI	33. AVIV	TEL	48. HOLY	SEE
4. ROAD	MAP	19. GRAB	BAG	34. RING	OFF	49. BANK	RUN
5. HOME	RUN	20. ZOOM	OUT	35. BOMB	BAY	50. BIRD	FLU
6. CREW	CUT	21. MESS	KIT	36. BABY	FAT	51. JAVA	PGM
7. FAUX	PAS	22. QUIZ	BEE	37. HIGH	HAT	52. MATH	LAB
8. OPUS	DEI	23. WARM	BED	38. ROOF	TOP	53. BODY	FIT
9. SODA	POP	24. HALF	WIT	39. ALSO	RAN	54. DROP	BOX
10. DEEP	FRY	25. ERGO	SUM	40. UNIX	JOB	55. PILI	NUT
11. MENU	BAR	26. THOU	ART	41. IRAQ	WAR	56. OVER	PAR
12. BEAR	HUG	27. FIAT	LUX	42. TOOL	BOX	57. PULP	BIT
13. SORT	OUT	28. TEAR	GAS	43. EPIC	ERA	58. SINE	DIE
14. IRON	BAR	29. DEAD	END	44. JAZZ	MAN	59. ZINC	ORE
15. DATA	SET	30. LONG	WAY	45. SHOW	OFF	60. LAUS	DEO

Figure 15.2 Sample ‘records’

Figure 15.3 shows the numerical equivalent of the keys in Figure 15.2 computed according to the third method described above. In the discussion which follows we will consistently use these particular numerical representations whenever we do numerical computations on the keys.

FOLK	1179601995	FALL	1178684492	EVIL	1163282764	FARM	1178686029
TEST	1413829460	OPEN	1330660686	BELL	1111837772	TAXI	1413568585
BACK	1111573323	BALI	1111575625	AVIV	1096173910	HOLY	1213156441
ROAD	1380925764	GRAB	1196572994	RING	1380535879	BANK	1111576139
HOME	1213156677	ZOOM	1515147085	BOMB	1112493378	BIRD	1112101444
CREW	1129465175	MESS	1296388947	BABY	1111573081	JAVA	1245795905
FAUX	1178686808	QUIZ	1364543834	HIGH	1212761928	MATH	1296127048
OPUS	1330664787	WARM	1463898701	ROOF	1380929350	BODY	1112491097
SODA	1397703745	HALF	1212238918	ALSO	1095521103	DROP	1146244944
DEEP	1145390416	ERGO	1163020111	UNIX	1431193944	PILI	1346980937
MENU	1296387669	THOU	1414025045	IRAQ	1230127441	OVER	1331053906
BEAR	1111834962	FIAT	1179205972	TOOL	1414483788	PULP	1347767376
SORT	1397707348	TEAR	1413824850	EPIC	1162889539	SINE	1397313093
IRON	1230131022	DEAD	1145389380	JAZZ	1245796954	ZINC	1514753603
DATA	1145132097	LONG	1280265799	SHOW	1397247831	LAUS	1279350099

Figure 15.3 Numerical equivalents of the keys in Figure 15.2

15.2 Choosing a hash function

A good hash function $h(K)$ must uniformly and randomly distribute keys over the table addresses $0, 1, 2, \dots, m - 1$. It must satisfy two important requisites, namely:

1. It can be calculated very fast — the principal advantage of a hash table over other implementations of the table ADT is that it typically takes far less key comparisons, starting from address $h(K)$, to find the desired record. However, if it takes too long to compute $h(K)$, an implementation which requires more comparisons but without any such computation, such as a BST, may turn out to be more efficient. We assume that it takes $O(1)$ time to evaluate $h(K)$ for any given key K .
2. It minimizes collisions — it is actually possible to devise a hash function that is ‘perfect’ in that no collisions occur for a fixed set of keys, for example, the set of instruction codes in an assembly language. In most applications such prior knowledge about the keys that are used is not available, making it impossible to devise a perfect hash function. In practice, therefore, we seek to minimize, but not altogether eliminate, collisions since this is usually not possible.

There are various methods that have been devised to transform a numeric key into a hash address in some specified range of addresses. The verb ‘to hash’ means to chop into small pieces. Applying this idea directly, we chop the key into parts and then add up the parts to get the hash address. For instance, if the key is **KEYS** = 1262836051, we may take the hash address to be $12 + 62 + 83 + 60 + 51 = 268$. This method is called **folding**.

Another way is to multiply the key by itself and then take the middle p digits of the product as the hash address; the idea is that these middle digits depend on all the digits of the key. This is called the **midsquare method**. Taking $p = 3$, the range of addresses generated by the method is from 000 to 999; in such a case we should use a table of size $m = 10^3 = 1000$.

In actual applications two simple methods have been found to work quite well; these are the **division method** and the **multiplicative method**. Both methods can be implemented such that the two requisites of a good hash function listed above are satisfied.

15.2.1 The division method

Let m be the table size; then the hash address $h(K)$ is simply the remainder upon dividing K by m . It follows that $0 \leq h(K) \leq m - 1$, which is as it should be from Eq.(15.1).

$$h(K) = K \bmod m \tag{15.2}$$

It is obvious that Eq.(15.2) can be computed very fast; a proper choice of the table size m will also reduce collisions. Here are suggestions on how **not** to choose m .

1. m should not be even. If m is even then even keys will hash to even addresses and odd keys will hash to odd addresses. Thus the function preserves whatever bias in terms of parity there might be in the keys.

2. m should not be a power of the radix of the computer. Consider the keys shown in Figure 15.3 and imagine applying Eq.(15.2) on each one with $m = 2^8 = 64$. Then $K \bmod m$ will simply be the last eight bits, or equivalently the last character, of K . Thus keys ending in the same character will hash to the same address, as we can see in Figure 15.4. We will obtain the same results if m were 32, 128 or 256; in each of these cases the hash addresses are confined to only 26 of the table addresses.

HASH ADDRESS	COLLIDING KEYS				HASH ADDRESS	COLLIDING KEYS			
0					15	ERGO	ALSO		
1	SODA	DATA	JAVA		16	DEEP	DROP	PULP	
2	GRAB	BOMB			17	IRAQ			
3	EPIC	ZINC			18	BEAR	TEAR	OVER	
4	ROAD	DEAD	BIRD		19	OPUS	MESS	LAUS	
5	HOME	SINE			20	TEST	SORT	FIAT	
6	HALF	ROOF			21	MENU	THOU		
7	LONG	RING			22	AVIV			
8	HIGH	MATH			23	CREW	SHOW		
9	BALI	TAXI	PILI		24	FAUX	UNIX		
10					25	BABY	HOLY	BODY	
11	FOLK	BACK	BANK		26	QUIZ	JAZZ		
12	FALL	EVIL	BELL	TOOL	27				
13	ZOOM	WARM	FARM		⋮				
14	IRON	OPEN			63				

Figure 15.4 The division method with $m = 2^6 = 64$

3. m should not be of the form $2^i \pm j$, where j is a small integer, if the keys are character strings interpreted in radix 2^r . Otherwise, keys obtained by a simple transposition of the characters comprising the key are likely to hash to the same address. To illustrate, suppose we have a four-character key $K = C_3C_2C_1C_0$ and we convert it to numeric form as

$$K = c_3 \times \beta^3 + c_2 \times \beta^2 + c_1 \times \beta^1 + c_0 \times \beta^0 \quad (15.3)$$

where c_i is the code for the character C_i and β is the radix of the character set. Using the identities

$$(p + q) \bmod m = (p \bmod m + q \bmod m) \bmod m \quad (15.4)$$

$$(pq) \bmod m = [(p \bmod m)(q \bmod m)] \bmod m \quad (15.5)$$

we obtain

$$\begin{aligned}
 h(K) &= K \bmod m \\
 &= (c_3\beta^3 + c_2\beta^2 + c_1\beta + c_0) \bmod m \\
 &= [(c_3\beta^3) \bmod m + (c_2\beta^2) \bmod m + (c_1\beta) \bmod m + c_0 \bmod m] \bmod m \\
 &= \{[(c_3 \bmod m)(\beta^3 \bmod m)] \bmod m + [(c_2 \bmod m)(\beta^2 \bmod m)] \bmod m \\
 &\quad + [(c_1 \bmod m)(\beta \bmod m)] \bmod m + c_0 \bmod m\} \bmod m
 \end{aligned} \quad (15.6)$$

Now, assume that $\beta = 2^8 = 256$ and $m = 2^4 + 1 = 17$; then $\beta \bmod m = 256 \bmod 17 = 1$, $\beta^2 \bmod m = 256^2 \bmod 17 = 1$ and $\beta^3 \bmod m = 256^3 \bmod 17 = 1$. Substituting these values in Eq.(15.6) we obtain

$$\begin{aligned} h(K) &= (c_3 \bmod m + c_2 \bmod m + c_1 \bmod m + c_0 \bmod m) \bmod m \\ &= (c_3 + c_2 + c_1 + c_0) \bmod m \end{aligned} \quad (15.7)$$

Since addition is commutative, it follows from Eq.(15.7) that any permutation of the characters C_3, C_2, C_1 and C_0 will hash to the same address. For instance, applying Eq.(15.2) with $m = 17$ to each of the keys shown below (the 24 permutations of W, X, Y and Z) will yield the same hash address, namely, 14.

WXYZ	1465407834	XWYZ	1482119514	YWXZ	1498896474	ZWXY	1515673689
WXZY	1465408089	XWZY	1482119769	YWZX	1498896984	ZWYX	1515673944
WYXZ	1465473114	XYWZ	1482250074	YXWZ	1498961754	ZXWY	1515738969
WYZX	1465473624	XYZW	1482250839	YXZW	1498962519	ZXYW	1515739479
WZXY	1465538649	XZWY	1482315609	YZWX	1499092824	ZYWX	1515804504
WZYX	1465538904	XZYW	1482316119	YZXW	1499093079	ZYXW	1515804759

If now we choose $m = 2^8 + 1 = 257$ then $\beta \bmod m = 256 \bmod 257 = 256$, $\beta^2 \bmod m = 256^2 \bmod 257 = 1$ and $\beta^3 \bmod m = 256^3 \bmod 257 = 256$. Substituting these values in Eq.(15.6) we obtain

$$\begin{aligned} h(K) &= (c_3 \times 256 + c_2 + c_1 \times 256 + c_0) \bmod m \\ &= [(c_3 + c_1) \times 256 + (c_2 + c_0)] \bmod m \end{aligned} \quad (15.8)$$

Hence c_1 and c_3 may be interchanged and/or c_0 and c_2 may be interchanged and the same hash address will be generated. Applying Eq.(15.2) with $m = 257$ to each of the keys shown above will yield the following hash addresses.

0	WXZY	WYZX	XWYZ	XZYW	YWXZ	YZXW	ZXWY	ZYWX
2	WXYZ	WZYX	YXWZ	YZWX				
4	WYXZ	WZXY	XYWZ	XZWY				
253	YWZX	YXZW	ZWYX	ZXYW				
255	XWZY	XYZW	ZWXY	ZYXW				

On a binary computer, the above considerations suggest that we should choose the table size m to be a *prime number that is not too close to exact powers of 2*. Figure 15.5 shows the hash addresses generated by Eq.(15.2) with $m = 59$ for the keys given in Figure 15.3. There are collisions as expected and some addresses are unused, but clearly we get a much better distribution of the keys over the table addresses 0 through 58 with this choice of m . Compare these results with those shown in Figure 15.4 and you should appreciate the need to choose the table size judiciously.

HASH ADDRESS	COLLIDING KEYS			HASH ADDRESS	COLLIDING KEYS		
0	BELL			30	FAUX		
1	MESS	EVIL	BIRD	31			
2				32	CREW	BANK	
3	SORT	TEAR	MATH	33			
4	ZOOM			34	HOME	TAXI	HOLY
5				35	DATA	QUIZ	
6				36	JAZZ		
7				37	SHOW		
8				38	IRAQ		
9	FOLK			39			
10				40			
11	TEST	THOU	EPIC	41	OVER		
12	GRAB	RING	ROOF	42	BABY		
13				43			
14				44			
15	FALL			45	SINE		
16	ERGO			46			
17	ALSO			47			
18	FARM	BODY		48	BACK	AVIV	
19	DEAD	LONG		49	BALI	JAVA	
20	IRON	HALF		50	PULP		
21	MENU			51			
22	BEAR	PILI		52	DEEP	ZINC	
23	WARM			53	FIAT		
24	DROP			54	HIGH		
25	ROAD			55			
26				56	OPEN		
27	OPUS			57	BOMB		
28	UNIX			58	SODA	LAUS	
29	TOOL						

Figure 15.5 The division method with $m = 59$

15.2.2 The multiplicative method

The multiplicative method is based on the following theorem (KNUTH3[1998], p. 518):

THEOREM 15.1 (Vera Turán Sós). Let θ be any irrational number. When the points $\theta \bmod 1, 2\theta \bmod 1, \dots, n\theta \bmod 1$ are placed in the line segment $[0..1]$, the $n+1$ line segments formed have at most three different lengths. Moreover, the next point $(n+1)\theta \bmod 1$ will fall in one of the largest existing segments.

Recall from Session 2 that $x \bmod 1$ is simply the fractional part of x ; for instance, $3.14159 \bmod 1 = 0.14159$. The figure below shows a graphical representation of the theorem with θ taken to be the golden ratio $\phi^{-1} = \frac{\sqrt{5}-1}{2} = 0.6180339887$. Shown in the second column is the fractional part of the product $i\theta, 1 \leq i \leq 10$, which is then plotted on the interval $[0..1]$. Note that the plotted points are evenly distributed on the interval. If now we compute the quantities $\lfloor 10(i\theta \bmod 1) \rfloor, 1 \leq i \leq 10$, we find that they constitute a complete permutation of the numbers 0 thru 9. Thus 1 is mapped to 6, 2 is mapped to 2, 3 is mapped to 8, and so on. These results suggest a hash function of the form

$$h(K) = \lfloor m(K\theta \bmod 1) \rfloor \quad (15.9)$$

where we have simply replaced 10 by the table size m and i by the key K . Note that Eq.(15.9) yields a hash address $0 \leq h(K) \leq m - 1$ as required; a judicious choice of θ is expected to distribute the keys uniformly over these table addresses.

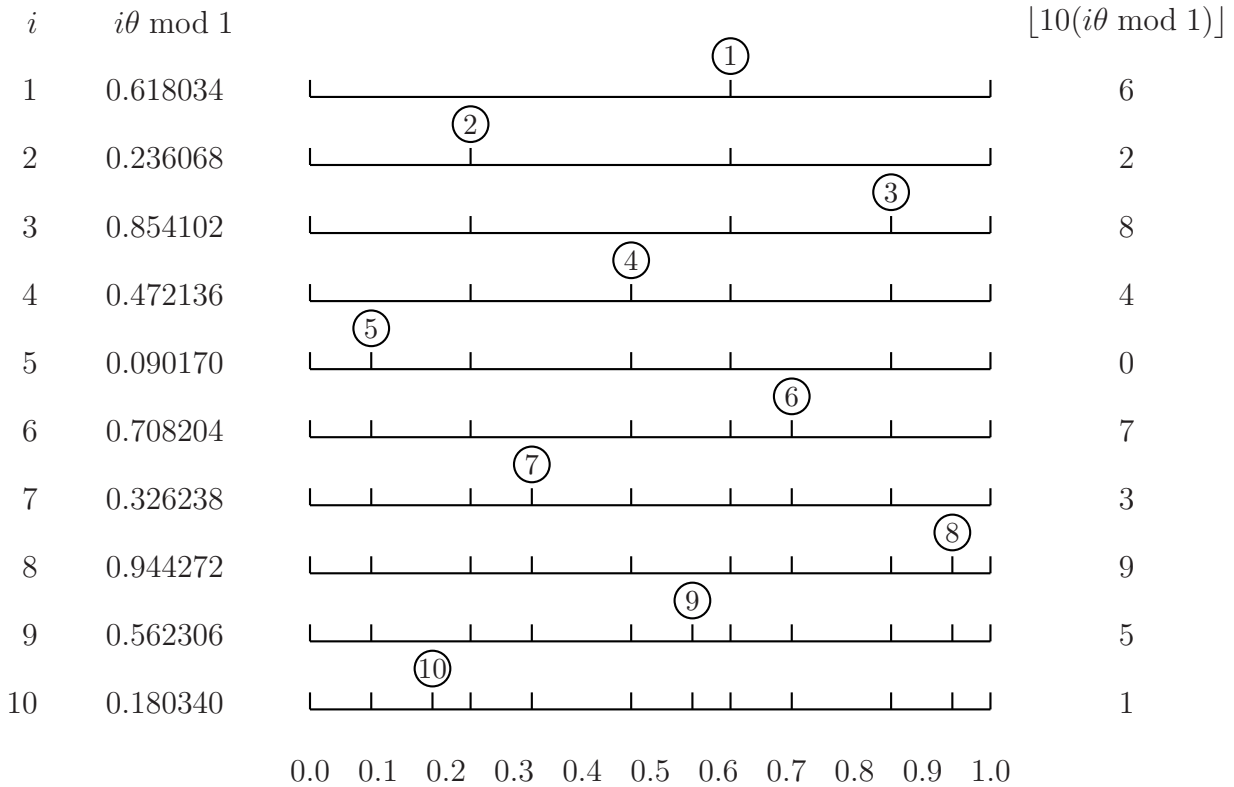


Figure 15.6 An application of Theorem 15.1 with $\theta = \text{golden ratio} = \frac{\sqrt{5}-1}{2}$

Knuth suggests the following strategy to compute Eq.(15.9) very fast.

1. Take θ to be I/w where w is the word size of the computer (for instance, 2^{32}) and I is an integer relatively prime to w . Multiply K by I to obtain a $2w$ -bit product in which the lower w bits contain the fractional part of the product $K\theta$, that is, $K\theta \bmod 1$.

$$\begin{array}{c}
 \boxed{\begin{array}{c} K \\ w \text{ bits} \end{array}} \times \boxed{\begin{array}{c} I \\ w \text{ bits} \end{array}} = \boxed{\begin{array}{c} KI \\ w \text{ bits} \quad \bullet \quad w \text{ bits} \end{array}} \\
 \text{imaginary radix point} \quad \underbrace{\hspace{1.5cm}}_{K\theta \bmod 1}
 \end{array}$$

2. Now take the table size m to be 2^p for some integer p ; then $h(K)$ is simply the leading p high order bits of $K\theta \bmod 1$. To retrieve $h(K)$, set all bits to the left of the imaginary radix point to 0 and then shift left p bits.

$$\begin{array}{c}
 \boxed{\begin{array}{c} KI \\ w \text{ bits} \quad \bullet \quad w \text{ bits} \end{array}} \\
 \underbrace{\hspace{1.5cm}}_{h(K) = \text{leading } p \text{ bits of } K\theta \bmod 1 \text{ for } m = 2^p}
 \end{array}$$

The suggested procedure is best illustrated in assembly language. Assume that K is in register 3, I is in register 5 and that m is 2^{10} . The following segment of IBM 360 Assembly code calculates $h(K)$ from Eq.(15.9).

```

MR      2,5
SR      2,2
SLDA    2,P
      ⋮
P      DC      F'10'
```

(15.10)

The first instruction (Multiply) multiplies the 32-bit integer K in register 3 with the 32-bit integer I in register 5 and stores the 64-bit product in registers 2 and 3 with the imaginary radix point between the two registers. The second instruction (Subtract) sets register 2 to zero. Finally, the third instruction (Shift Left Double, Arithmetic) shifts the 10 high-order bits in register 3 into register 2, which now contains $h(K)$.

The figure below shows the hash addresses generated by Eq.(15.9) with $\theta = \phi^{-1} = 0.6180339887$ and $m = 2^6$ when applied to the keys given in Figure 15.3. Note that the keys are well distributed over the table addresses 0 through 63.

HASH ADDRESS	COLLIDING KEYS			HASH ADDRESS	COLLIDING KEYS		
0	HOME	FALL		32	TEST		
1	ROAD	BELL		33			
2				34	BACK	SODA	
3	FOLK			35			
4	BODY			36			
5				37	JAZZ		
6				38			
7				39	EVIL		
8	ERGO	DEAD		40	LAUS		
9	IRON	HOLY		41			
10	ZINC			42			
11				43	THOU		
12	CREW	SHOW	PILI	44	OVER		
13	EPIC			45	HIGH		
14	WARM	PULP		46			
15				47	HALF		
16	BALI	GRAB		48			
17	BIRD	JAVA		49	DROP		
18	ROOF			50			
19				51	BOMB		
20	SORT			52	MESS	UNIX	
21	BEAR	MATH	SINE	53	RING		
22	LONG			54			
23	OPEN	FIAT	TEAR	55			
24	FAUX			56	TAXI		
25				57	AVIV		
26	DEEP	ZOOM		58			
27				59	OPUS	FARM	BANK
28	TOOL			60			
29				61			
30	QUIZ			62	MENU	BABY	IRAQ
31	DATA			63	ALSO		

Figure 15.7 The multiplicative method with $\theta = \phi^{-1} = 0.6180339887$ and $m = 2^6$

15.3 Collision resolution by chaining

The famous von Mises ‘birthday paradox’ asserts that if there are 23 persons in a room, there is a greater than even chance (the actual value is 0.50729723) that two of them have the same birthday. In the context of hashing, this means that if we insert 23 keys into an initially empty table of size 365 using a good hash function, there is a greater than 50% chance that two will collide. As Standish (STANDISH[1994], p. 465) points out, ‘*Even in sparsely occupied hash tables, collisions are relatively common.*’ Thus we must have a *collision resolution policy* to handle collisions when they inevitably occur. In this section we will consider a method for resolving collisions called **chaining**; in the next section we will look at another strategy called **open addressing**.

In collision resolution by chaining, all records which hash to the same address are constituted into a linked list, called a *chain*. A maximum of m chains, some possibly null, are maintained. The pointers to these m chains are stored in an array of size m .

Figure 15.8 shows the **chained hash table** that is generated when the first 11 records from Figure 15.2 are inserted into an initially empty table of size $m = 11$ using the division method, $h(K) = K \bmod m$, for hashing. For instance, **BACK** (1111573323) and **CREW** (1129465175) both yield a remainder of 3 when divided by 11, and so the records (**BACK** **END**) and (**CREW** **CUT**) are in the chain pointed to by $HEAD(3)$. None of the 11 records hash to address 4, so the chain $HEAD(4)$ is null.

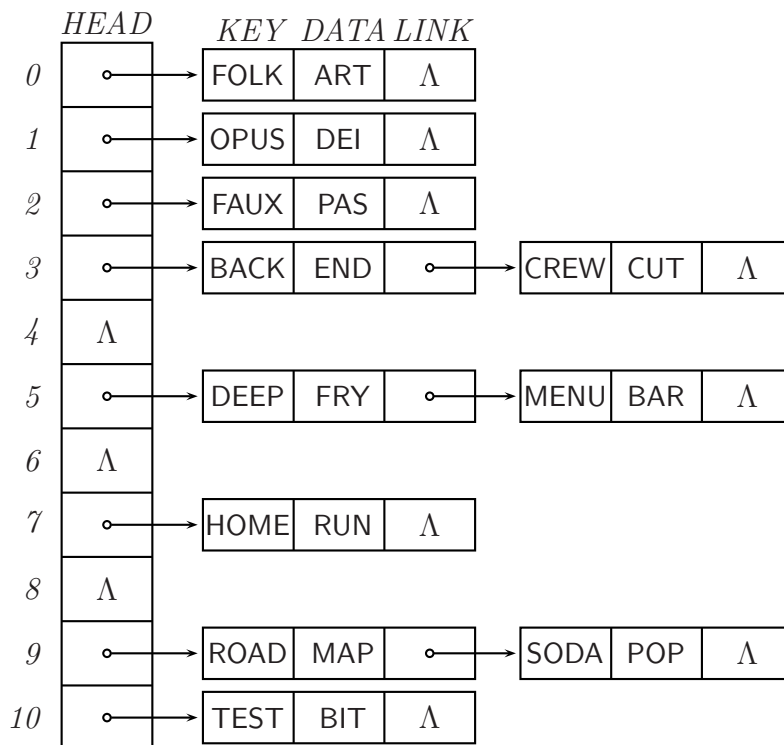


Figure 15.8 A chained hash table

We take note of the following observations pertaining to a chained hash table T of size m .

1. A chain consists of all the records whose keys hashed to the same address in T for the hash function $h(K)$. A chain is maintained as a linked list where each record is a node in the list.
2. Assuming that $h(K)$ is a good hash function, the average length of a chain is n/m where n is the number of records in the table. The quantity n/m is called the **load factor**, denoted α , for T .
3. Any number of records can be stored in a chained hash table, subject only to the availability of nodes from the memory pool. This means that the load factor α can be less than, equal to or greater than 1.
4. Given a search key K , we search the table as follows:
 - (a) Calculate $i = h(K)$.
 - (b) Traverse the list pointed to by $HEAD(i)$ and look for the record whose key matches K , if any.
5. Given a new record (k_j, d_j) , we insert a new node into the table as follows:
 - (a) Perform a table search with search key $K = k_j$.
 - (b) If the search is *unsuccessful* insert a node for the new record into the appropriate chain; otherwise, abort the insertion. The new node may be inserted before the first or after the last node in the chain, or it may be inserted into the chain such that the records are arranged in order by key.
6. Given a record (k_j, d_j) to be deleted from T , we proceed as follows:
 - (a) Perform a table search with search key $K = k_j$.
 - (b) If the search is unsuccessful, then there is nothing to delete and we leave T unchanged (in effect we wasted time doing nothing). Otherwise, if the search is *successful*, then we remove the node which contains the record from its chain and return it to the memory pool.
7. Assuming that each record in the table is an equally likely target of a search, the average search time for *successful* searches is the same for any ordering of the records in a chain. Whatever the ordering, we will be examining half the records in a chain on average.
8. If the records in a chain are arranged in order by key, *unsuccessful* searches are serviced faster. This is because we need not search the entire chain to discover that what we are looking for is not there. Since an insert operation is preceded by an unsuccessful search, it follows that insertions will also be serviced faster.

15.3.1 EASY procedures for chained hash tables

The following EASY procedures implement the three basic operations supported by a hash table, viz., search, insert and delete, and the auxiliary operation to initialize a chained hash table. The common input to all four procedures is a chained hash table T represented by the data structure $\mathbb{T} = [m, \text{HEAD}(1:m), (\text{KEY}, \text{DATA}, \text{LINK})]$. In each case, we assume that a pointer to \mathbb{T} is passed to the procedure, allowing access to its components.

Initializing a chained hash table

```

1  procedure CHAINED_HASH_TABLE_INIT ( $\mathbb{T}$ )
2  for  $i \leftarrow 1$  to  $m$  do
3       $\text{HEAD}(i) \leftarrow \Lambda$ 
4  endfor
5  end CHAINED_HASH_TABLE_INIT

```

Procedure 15.1 Initializing a chained hash table

Inserting a record into a chained hash table

```

1  procedure CHAINED_HASH_TABLE_INSERT( $\mathbb{T}, k, d$ )
▷ Inserts a new record  $(k, d)$  into a hash table. Collisions are resolved by chaining.
▷ Insert operation is aborted if there is already a record with the same key in the table.
▷
▷ Search for possible duplicate key
2       $i \leftarrow h(k)$ 
3       $\alpha \leftarrow \text{HEAD}(i)$ 
4      while  $\alpha \neq \Lambda$  do
5          if  $\text{KEY}(\alpha) = k$  then [output ‘Duplicate key found.’; stop]
6              else  $\alpha \leftarrow \text{LINK}(\alpha)$ 
7      endwhile
▷ Insert new record at head of chain
8      call GETNODE( $\alpha$ )
9       $\text{KEY}(\alpha) \leftarrow k$ ;  $\text{DATA}(\alpha) \leftarrow d$ 
10      $\text{LINK}(\alpha) \leftarrow \text{HEAD}(i)$ ;  $\text{HEAD}(i) \leftarrow \alpha$ 
11 end CHAINED_HASH_TABLE_INSERT

```

Procedure 15.2 Insertion into a chained hash table

As implemented, procedure CHAINED_HASH_TABLE_INSERT aborts the insertion and terminates execution if a record with the same key as the new record to be inserted is found; if this is too drastic an action to take, line 5 should be suitably modified. Line 10 inserts a new record at the head of its list; in effect the records in a chain are maintained in order by time of insertion, from most to least recent. This is a desirable arrangement if the records most recently inserted into the table are most likely the first to be searched for. Alternatively, the lists may be maintained in order by key so that unsuccessful searches and insertions are serviced faster. In such a case line 10 should be modified accordingly. The body of the **while** loop should also be modified to include an explicit test for an unsuccessful search for a possible early exit (as discussed below).

Searching for a record in a chained hash table

```

1  procedure CHAINED_HASH_TABLE_SEARCH( $\mathbb{T}$ ,  $K$ )
  ▷ Searches for a record whose key is  $K$  in a table built using chaining to resolve
  ▷ collisions. Procedure returns address of matching record, if found; or  $\Lambda$ , otherwise.
2     $i \leftarrow h(K)$ 
3     $\alpha \leftarrow \text{HEAD}(i)$ 
4    while  $\alpha \neq \Lambda$  do
5      if  $\text{KEY}(\alpha) = K$  then return( $\alpha$ )    ▷ successful search
6      else  $\alpha \leftarrow \text{LINK}(\alpha)$ 
7    endwhile
8    return( $\Lambda$ )    ▷ unsuccessful search
9  end CHAINED_HASH_TABLE_SEARCH

```

Procedure 15.3 Search in a chained hash table

As implemented, procedure CHAINED_HASH_TABLE_SEARCH assumes that the chains are *not* maintained in order by key. Thus, an entire chain must be traversed (lines 4–7) before the search can be declared unsuccessful (line 8). On the other hand, if the records in a chain are arranged in ascending [descending] order by key, then the condition $K < \text{KEY}(\alpha)$ [$K > \text{KEY}(\alpha)$] already signifies an unsuccessful search, allowing for an early exit from the loop. If the lists are sorted by key, the body of the **while** loop in lines 4–7 should be modified to include such a test.

Deleting a record from a chained hash table

```

1  procedure CHAINED_HASH_TABLE_DELETE( $\mathbb{T}$ ,  $K$ )
  ▷ Deletes record whose key is  $K$  from a hash table built using chaining to resolve
  ▷ collisions. Table is unchanged if there is no such record in the table.
2     $i \leftarrow h(K)$ 
3     $\alpha \leftarrow \text{HEAD}(i)$ 
4    while  $\alpha \neq \Lambda$  do
5      case
6        :  $\text{KEY}(\alpha) \neq K$  : [ $\beta \leftarrow \alpha$ ;  $\alpha \leftarrow \text{LINK}(\alpha)$ ]
7        :  $\text{KEY}(\alpha) = K$  : [if  $\alpha = \text{HEAD}(i)$  then  $\text{HEAD}(i) \leftarrow \text{LINK}(\alpha)$ 
8                           else  $\text{LINK}(\beta) \leftarrow \text{LINK}(\alpha)$ 
9                           call RETNODE( $\alpha$ )
10       return]
11    endcase
12  endwhile
13 end CHAINED_HASH_TABLE_DELETE

```

Procedure 15.4 Deletion from a chained hash table

Line 7 deletes the first node in a chain and line 8 deletes any other node in the chain; in the absence of a list head, the first node in a chain is handled differently from the rest. The deleted node is returned to the memory pool (line 9) so that it can be reused.

15.3.2 Analysis of hashing with collision resolution by chaining

Assume that we have an initially empty hash table T of size m into which we have inserted n records using a hash function that uniformly distributed the records among the m table slots; it follows from this that the average length of a chain is the load factor $\alpha = n/m$. Furthermore, assume that new records were added at the end of a chain during an insertion. Now, let C_n and C'_n denote the average number of key comparisons performed in a successful search and in an unsuccessful search, respectively. Then we have:

$$C'_n = \alpha \quad (15.11)$$

Since the records are *not* maintained in order by key, we must examine *all* the records in a chain, of average length equal to α , before we discover that the record we are looking for is not there; hence, Eq.(15.11) for unsuccessful searches follows.

For successful searches, we have:

$$\begin{aligned} C_n &= \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m} \right) \\ &= \frac{1}{n} \left[n + \frac{1}{m} \sum_{i=1}^n (i-1) \right] \\ &= 1 + \frac{1}{nm} \sum_{i=1}^{n-1} i \\ &= 1 + \frac{1}{nm} \cdot \frac{(n-1)n}{2} \\ &= 1 + \frac{\alpha}{2} - \frac{1}{2m} \\ &\approx 1 + \frac{\alpha}{2} \end{aligned} \quad (15.12) \quad (15.13)$$

The quantity $(i-1)/m$ in Eq.(15.12) is the average length of a chain in T when the i th record is inserted; from Eq.(15.11), this is the average number of comparisons performed in an unsuccessful search prior to inserting the record. Once inserted, it takes an additional comparison to find the same record, i.e., $1 + (i-1)/m$. If now we sum this quantity for all n records in T and then divide by n , we obtain the average number of comparisons to find any record in T , as given by Eq.(15.13).

What Eq.(15.11) and Eq.(15.13) signify is that if we take m to be roughly equal to n such that the load factor is about 1, then the search operation takes essentially constant time on the average. Since the most expensive part of both the insert and delete operations is a search, it follows that these two other operations also take constant time on average.

15.4 Collision resolution by open addressing

In collision resolution by open addressing all the records are stored in a sequential table of size m . When a key hashes to a table cell that is already occupied, the table is *probed* for a vacant cell in which the new record can be stored. The order in which the table cells are successively probed in search of an available cell, if any, is prescribed by a **probe sequence function**.

Figure 15.9 depicts the process of inserting the record (k_j, d_j) into an **open-address hash table** of size $m = 17$. The hash address $h(k_j)$ is 9, but cell 9 is already occupied. The arrows indicate the sequence of probes made until a vacant cell, namely cell 11, is found in which to store the record. The constant difference between successive probe addresses is called the *probe decrement*, s ; in this case, $s = 3$. Starting at the hash address $h(k_j)$, we probe the table addresses in decrements of s until we find an unoccupied address. If, as we probe, we ‘fall off’ the lower (address-wise) end of the table, then we ‘wrap around’ to the other end. The following segment of EASY code, placed within an appropriate loop, captures the process:

```

 $i \leftarrow i - s$ 
if  $i < 0$  then  $i \leftarrow i + m$ 

```

Starting at $i = h(k_j) = 9$ and with $s = 3$, you may easily verify that this segment of code, executed five times, generates the probe sequence 9, 6, 3, 0, 14, 11 as shown in Figure 15.9.

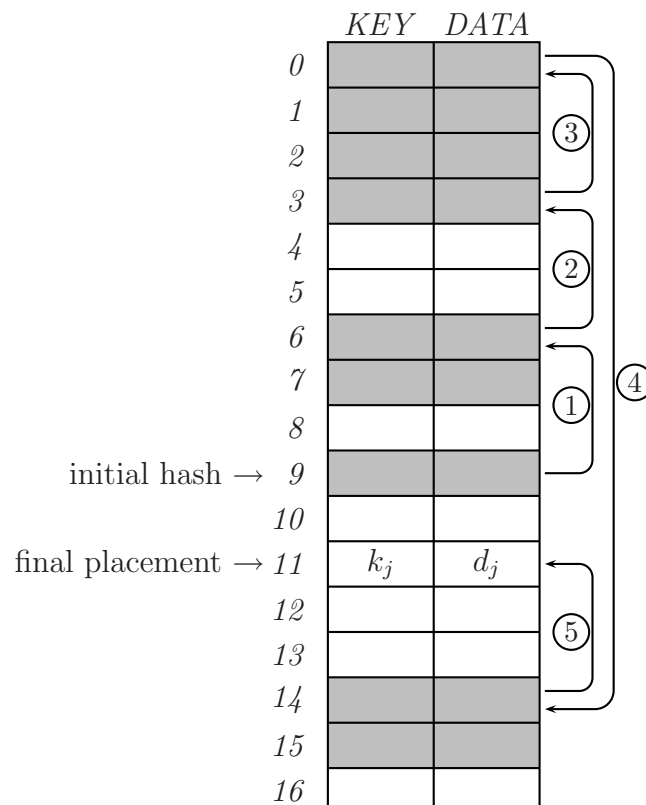


Figure 15.9 Collision resolution by open addressing

We take note of the following observations pertaining to an open-address hash table T of size m .

1. There should be a way to distinguish *vacant* cells from *occupied* cells in T . For instance, a vacant cell can be indicated by an entry of 0 in the *KEY* field if the keys are positive integers, or by a string of blank characters if the keys are character strings. In any case, the *KEY* field must be initialized to indicate that all cells are initially vacant.
2. The table is considered *full* when the number of records n is $m - 1$, i.e., there is exactly one vacant cell in T . It follows from this that the load factor $\alpha = n/m$ is never greater than 1.
3. Given a search key K , we search the table as follows:
 - (a) Calculate $i = h(K)$.
 - (b) If $KEY(i) = K$ then the search is done; we find the record at its hash address. Otherwise we trace the path, as prescribed by a probe sequence function, which K would have followed in its quest for a vacant slot had it been inserted into T . This path should then lead us to a cell in the table which contains the desired record, in which case the search is successful; or it leads us to a vacant cell, in which case the search is unsuccessful. This is the reason why we must leave at least one vacant cell in the table; if we allow all cells to be occupied, then all unsuccessful searches on a full table will require m probes.
4. Given a new record (k_j, d_j) to be inserted into T and assuming that T is not full, we proceed as follows:
 - (a) Perform a table search with search key $K = k_j$.
 - (b) If the search is successful, we abort the insertion; otherwise, if the search leads us to a vacant cell then we store the record in the cell.
5. Given a record (k_j, d_j) to be deleted from T , we proceed as follows:
 - (a) Perform a table search with search key $K = k_j$.
 - (b) If the search is unsuccessful, then there is nothing to delete and we leave T unchanged. Otherwise, if the search is successful, then we mark the cell which contained the deleted record as *vacated*, not as *vacant*. If a vacated cell is treated as if it were vacant, then a record whose probe sequence ‘passed by’ this cell cannot be found anymore. For instance, if the record in cell 3 of Figure 15.9 is deleted and cell 3 is marked *vacant* then a search for the record (k_j, d_j) will terminate unsuccessfully in cell 3. Thus we must distinguish between vacant and vacated cells. A vacated cell can be indicated by an entry of -1 or a special character string in the *KEY* field. If a cell marked *vacated* is encountered during a search, the search should continue until the desired record is found or a cell marked *vacant* is encountered. A vacated cell can be reused to store a new record in an insert operation following an unsuccessful search.

- (c) An alternative approach is to mark the cell containing the deleted record as *vacant* and then rehash all the records which would change positions in the table as a consequence of the fact that this previously occupied cell is now vacant. We will look more closely at this strategy later on.

15.4.1 Linear probing

Suppose we want to insert the record (k_j, d_j) into an open-address hash table T of size m . We compute $h(k_j)$ and we obtain, say, 5. We find cell 5 to be occupied, so we try cell 4. We find cell 4 to be occupied also, so we try cell 3. If cell 3 is occupied we next try cell 2, then cell 1, 0, $m-1$, $m-2$, and so on, until we encounter a vacant cell in which to store the record. This process, in which we examine *consecutive* cells in a hash table in decrements of 1, doing a wraparound if necessary, is called **linear probing**. In general, for some key K , the complete *probe sequence* is

$$h(K), h(K) - 1, h(K) - 2, \dots, 1, 0, m - 1, m - 2, \dots, h(K) + 1 \quad (15.14)$$

which is generated by the *probe sequence function*

$$p(K) = [h(K) - i] \bmod m \quad i = 0, 1, 2, \dots, m - 1 \quad (15.15)$$

Note that the sequence (15.14) is a complete permutation of the table addresses 0 through $m-1$; if needs be, every cell in T will be examined. Note also that the initial hash address $h(K)$ determines the complete probe sequence. Since $h(K)$ can only have values from 0 through $m-1$, only m distinct probe sequences are possible in linear probing out of $m!$ possible sequences.

Figure 15.10 shows the hash table generated when the first 11 records in Figure 15.2, with numeric equivalents as given in Figure 15.3, are inserted into an initially empty hash table of size $m = 19$ using the division method, $h(k) = k \bmod m$, with linear probing to resolve collisions. The figure depicts an undesirable phenomenon which arises in linear probing, namely, the tendency for records to occupy successive cells in the table forming so-called *clusters*. In Figure 15.10, a record will be stored in cell 4 only if it hashes to 4, but it will be stored in cell 17 if hashes to 2 or 1 or 0 or 18 or 17. Thus cell 17 is five times more likely to be occupied than cell 4. The bigger a cluster, the faster it grows. As clusters grow in size they tend to merge, resulting in even bigger clusters. For instance, once cell 17 is occupied, a cluster of size 8 is formed. This phenomenon, brought about by linear probing, in which clusters form, grow and merge is called **primary clustering**.

As a hash table gets full, primary clustering causes the performance of hashing by open addressing to degrade rapidly. Insertions take longer and longer to execute since a record is more likely to hash to a cell in a cluster, and a long sequence of occupied cells must be probed before a vacant one is found.

			KEY	DATA
	(k, d)	$h(k)$	0	SODA POP
			1	DEEP FRY
			2	BACK END
			3	
			4	
			5	
			6	
			7	ROAD MAP
			8	HOME RUN
			9	CREW CUT
			10	FOLK ART
			11	
			12	
			13	
			14	OPUS DEI
			15	FAUX PAS
			16	TEST BIT
			17	
			18	MENU BAR

Figure 15.10 Open-address hash table (with linear probing to resolve collisions)

15.4.2 Quadratic probing

Quadratic probing attempts to solve the problem of primary clustering by making the probe sequence depend on the probe number i not linearly, as in Eq.(15.15), but rather quadratically, as in the function

$$p(K) = [h(K) - c_1 i - c_2 i^2] \bmod m \quad i = 0, 1, 2, \dots, m-1 \quad (15.16)$$

Suitable values of the parameters c_1 and c_2 and the table size m are chosen so that Eq.(15.16) produces a probe sequence that is a complete permutation of the table addresses. An example of quadratic probing is *quadratic residue search* which uses the probe sequence

$$h(K), [h(K) + i^2] \bmod m, [h(K) - i^2] \bmod m \quad i = 1, 2, \dots, (m-1)/2 \quad (15.17)$$

The table size m should be a prime number of the form $4p+3$ for some integer p so that the probe sequence (15.17) is a complete permutation of $(0, 1, 2, \dots, m-1)$. To illustrate, suppose we are inserting a record (K, d) into a table of size $m = 19$ (which is $4 * 4 + 3$) and the cell at address $h(K) = 11$ happens to be occupied; the third column in Figure (15.11) below prescribes the sequence in which we are to examine the other cells until we find a vacant one. The idea is that by jumping from one cell to another as we probe $(11, 12, 10, 15, 7, 1, \dots)$, clustering will be eliminated. Indeed quadratic probing eliminates primary clustering, but a milder form called **secondary clustering** still occurs. This

is because the hash address $h(K)$ completely determines the resulting probe sequence; thus keys which hash to the same address follow the same probe sequence. As in linear probing, only m distinct probe sequences are possible in quadratic probing.

i	$h(K) + i^2$ $h(K) - i^2$	$[h(K) + i^2] \bmod 19$ $[h(K) - i^2] \bmod 19$
1	12 10	12 10
2	15 7	15 7
3	20 2	1 2
4	27 -5	8 14
5	36 -14	17 5
6	47 -25	9 13
7	60 -38	3 0
8	75 -53	18 4
9	92 -70	16 6

Figure 15.11 Probe sequence for quadratic residue search for $h(K) = 11$

15.4.3 Double hashing

As in linear and quadratic probing, the probe sequence in double hashing starts with the hash address $h(K)$, but unlike in the the first two, the rest of the sequence now depends on a **secondary hash function** $h'(K)$. It is extremely unlikely that two keys k_i and k_j which hash to the same address $h(k_i) = h(k_j)$, will also have the same value for $h'(k_i)$ and $h'(k_j)$. Thus although colliding keys start out from the same hash address they subsequently follow different probe sequences, eliminating both primary and secondary clustering. In double hashing m^2 distinct probe sequences, one for every pair $(h(K), h'(K))$, are possible; this makes double hashing superior to both linear and quadratic probing.

The probe sequence function for double hashing has the form

$$p(K) = [h(K) - i \cdot h'(K)] \bmod m \quad i = 0, 1, 2, \dots, m-1 \quad (15.18)$$

For the entire probe sequence to be a complete permutation of the table addresses $(0, 1, 2, \dots, m-1)$, the secondary hash function $h'(K)$ must yield an integer between 1 and $m-1$ that is *relatively prime to* m . If m is prime (as it should be if we use the division method for our primary hash function) then $h'(K)$ can be *any* integer from 1 through $m-1$. If $m = 2^p$ for some integer p (as recommended if use multiplicative hashing) then $h'(K)$ can be *any odd* integer between 1 and $m-1$ inclusive.

As an example, suppose that we have a table of size $m = 16$ and $h(k_i) = h(k_j) = 7$, $h'(k_i) = 3$ and $h'(k_j) = 9$. Then the probe sequences for k_i and k_j , computed using Eq.(15.18), are:

k_i : 7 4 1 14 11 8 5 2 15 12 9 6 3 0 13 10
 k_j : 7 14 5 12 3 10 1 8 15 6 13 4 11 2 9 0

Thus k_i and k_j follow different probe sequences. Likewise, since 3 and 9 are relatively prime to 16, the probe sequences are complete permutations of the table addresses 0 through 15. Now suppose that for some key K we have $h(K) = 7$ and $h'(K) = 2$; then the probe sequence will be:

K : 7 5 3 1 15 13 11 9

After 9 comes 7 and the sequence repeats; half of the table addresses are missing because 2 is not relatively prime to 16. A table search during an insert operation which examines the addresses 7, 5, 3, ..., 9 and finds them occupied will conclude that there is no more available cell although there may in fact be unprobed vacant cells.

Knuth (KNUTH3[1998], p.529) suggests the following schemes for double hashing. They are guaranteed to produce a probe sequence that does not miss out on any of the table addresses.

1. Division method: $h(K) = K \bmod m$

Use $h'(K) = 1 + K \bmod (m - 2)$ or $h'(K) = 1 + \lfloor K/m \rfloor \bmod (m - 2)$. Choose m such that m and $m - 2$ are primes, e.g., 61 and 59, 109 and 107, 523 and 521, 1021 and 1019, and so on.

2. Multiplicative method: $h(K) = \lfloor m(K\theta \bmod 1) \rfloor$, $m = 2^p$

If $h(K)$ is computed as suggested in section 15.2.2, then after retrieving $h(K)$ compute $h'(K)$ as follows: (a) set all bits to the left of the imaginary radix point to zero, (b) shift left p bits, and (c) set the rightmost of the p bits to 1 (to make sure that $h'(K)$ is odd). The following assembly code computes $h'(K)$ after $h(K)$ is retrieved from register 2 in (15.10):

	SR	2,2	
	SLDA	2,P	
	O	2,ONE	
	:		
ONE	DC	F'1'	(15.19)

The last instruction (Or) sets the rightmost bit of register 2 to 1 yielding an odd integer. Now $h'(K)$ can be retrieved from register 2.

Figure 15.12 shows the hash table generated when the first 11 records in Figure 15.2, with numeric equivalents as given in Figure 15.3, are inserted into an initially empty table of size $m = 19$ using the division method, $h(k) = k \bmod m$, with double hashing to resolve

collisions. The secondary hash function is $h'(k) = 1 + k \bmod (m - 2)$. Compare this table with that in Figure 15.10; note that the records are more spread out this time.

(k, d)	$h(k)$	$h'(k)$		KEY	DATA
			0	SODA	POP
			1		
FOLK ART	10	12	2	BACK	END
TEST BIT	16	15	3		
BACK END	2	2	4		
ROAD MAP	7	6	5	OPUS	DEI
HOME RUN	8	9	6	DEEP	FRY
CREW CUT	10	17	7	ROAD	MAP
FAUX PAS	15	3	8	HOME	RUN
OPUS DEI	15	5	9		
SODA POP	0	7	10	FOLK	ART
DEEP FRY	2	15	11		
MENU BAR	18	4	12	CREW	CUT
			13		
			14		
			15	FAUX	PAS
			16	TEST	BIT
			17		
			18	MENU	BAR

Figure 15.12 Open-address hash table (with double hashing to resolve collisions)

15.4.4 EASY procedures for open-address hash tables

The following EASY procedures implement the three basic operations supported by a hash table, viz., search, insert and delete, and the auxiliary operation to initialize an open-address hash table. The common input to all four procedures is an open-address hash table T represented by the data structure $\mathbb{T} = [m, n, KEY(1:m), DATA(1:m)]$. In each case, we assume that a pointer to \mathbb{T} is passed to the procedure, allowing access to its components. All keys inserted into the table are assumed to be positive integers.

Initializing an open-address hash table

```

1  procedure OPEN_ADDRESS_HASH_TABLE_INIT ( $\mathbb{T}$ )
2  for  $i \leftarrow 1$  to  $m$  do
3     $KEY(i) \leftarrow 0$ 
4  endfor
5  end OPEN_ADDRESS_HASH_TABLE_INIT

```

Procedure 15.5 Initializing an open-address hash table

Inserting a record into an open-address hash table

```

1  procedure OPEN_ADDRESS_HASH_TABLE_INSERT( $\mathbb{T}, k, d$ )
  ▷ Inserts a new record  $(k, d)$  into a hash table. Collisions are resolved by open
  ▷ addressing. Insert operation is aborted if the table is full or if there is already
  ▷ a record in the table with the same key.
2  if  $n = m - 1$  then [output 'Table is full.'; stop]
3   $i \leftarrow h(k)$       ▷ hash address
4   $s \leftarrow p(k)$     ▷ probe decrement
5  loop
6    case
7      : $KEY(i) = k$ : [output 'Duplicate key found.'; stop]
8      : $KEY(i) > 0$ : [ $i \leftarrow i - s$ ; if  $i < 0$  then  $i \leftarrow i + m$ ]
9      : $KEY(i) = 0$ : [ $KEY(i) \leftarrow k$ ;  $DATA(i) \leftarrow d$ ;  $n \leftarrow n + 1$ ; return]
10   endcase
11  forever
12  end OPEN_ADDRESS_HASH_TABLE_INSERT

```

Procedure 15.6 Insertion into an open-address hash table

The table is considered full (line 2) once it has only one remaining vacant position. This is where all unsuccessful searches on a full table will terminate. The number of probes that each such unsuccessful search entails ranges from a minimum of zero to a maximum of $m - 1$.

As coded, procedure OPEN_ADDRESS_HASH_TABLE_INSERT can implement either linear probing or double hashing. If we take $p(k)$ in line 4 to be 1, we get linear probing; if we take $p(k)$ to be $h'(k)$, we get double hashing. Quadratic probing does not fit nicely into the structure of the procedure, but this should not really matter. We may have a good reason to use linear probing, for instance if we want to be able to rearrange the hash table after a deletion (to be discussed below), but we hardly have any reason to use quadratic probing since double hashing is a much better alternative.

The procedure assumes that a table cell is either vacant ($KEY(i) = 0$) or occupied ($KEY(i) > 0$). If deletions are allowed such that there are cells marked *vacated* (as in $KEY(i) = -1$), then the procedure must be modified to allow for vacated cells to be reused.

The code in line 8 implements Eq.(15.15) for linear probing, in which case $p(k) = 1$, or Eq.(15.18) for double hashing, in which case $p(k) = h'(k)$. Note that there is no need to make any explicit call to the **mod** function in applying these equations; the **if** statement handles the requisite wraparound when the probe address falls off the lower end (addresswise) of the table.

The condition $KEY(i) = 0$ in line 9 is a sure guarantee that there is no record in the entire hash table whose key is k , and therefore insertion of the new record (k, d) can proceed. If such a record were previously inserted then it must be in cell i and we should not find cell i vacant.

Searching for a record in an open-address hash table

```

1  procedure OPEN_ADDRESS_HASH_TABLE_SEARCH( $T, K$ )
  ▷ Searches for a record in a hash table built using open addressing to resolve collisions.
  ▷ Procedure returns index of matching record, if found; or  $-1$ , otherwise.
2     $i \leftarrow h(K)$       ▷ hash address
3     $s \leftarrow p(K)$     ▷ probe decrement
4    loop
5      case
6        :  $KEY(i) = K$  : return( $i$ )
7        :  $KEY(i) = 0$  : return ( $-1$ )
8        :  $KEY(i) > 0$  : [ $i \leftarrow i - s$ ; if  $i < 0$  then  $i \leftarrow i + m$ ]
9      endcase
10   forever
11  end OPEN_ADDRESS_HASH_TABLE_SEARCH

```

Procedure 15.7 Search in an open-address hash table

As we have previously explained, the condition $KEY(i) = 0$ means that there is no record in the table whose key matches the search key, or equivalently, the search is unsuccessful. Every unsuccessful search terminates in a vacant cell. We have noted early on that this is the reason why we must leave at least one vacant cell in the table and consider the table to be full when $n = m - 1$.

Deleting a record from an open-address hash table

Deletion from a chained hash table is easy, almost trivial; deletion from an open-address hash table is anything but. Earlier we noted two different ways of handling deletions of the latter kind. One way is to mark the cell previously occupied by the deleted record as *vacated*; this is sometimes referred to as *lazy deletion*. Thus we have three categories of cells: vacant, occupied and vacated. During a search, we treat a vacated cell as if it were occupied in that we continue searching past a vacated cell. In an insert operation *following an unsuccessful search*, we consider a vacated cell as vacant; we store the record in the first cell that we encounter that is either vacant or vacated. The problem with this approach is that if deletions are common, then there will be a proliferation of vacated cells resulting in longer search times for unsuccessful searches and insert operations.

An alternative approach to deletions is to mark the cell containing the deleted record as *vacant* and then rehash all the records which might change positions in the table as a consequence of the fact that this previously occupied cell is now vacant. There is no need to invent the special category ‘vacated’, resulting in neater code and faster unsuccessful searches and insertions. However, this method clearly entails additional work during deletions. Furthermore, it is feasible only if collisions are resolved by linear probing.

The main idea behind this approach is to *rearrange the records in the table such that the table becomes what it should have been had we not inserted the record that was deleted*. To illustrate, consider the figures shown below. Figure 15.13(b) shows the hash table that results after inserting the last ten records in Figure 15.2 using the division method for hashing and linear probing to resolve collisions. Figure 15.13(c) depicts the table after

record (BODY,FIT) is deleted from cell 2 leaving it vacant. The rearranged table, as a result of the actions enumerated below, is shown in Figure 15.13(d).

$h(\text{PILI}) = 3$; record (PILI NUT) is moved to cell 2.

$h(\text{DROP}) = 0$; record (DROP BOX) stays in cell 0.

$h(\text{OVER}) = 0$; record (OVER PAR) stays in cell 12.

$h(\text{ZINC}) = 12$; record (ZINC ORE) stays in cell 11.

$h(\text{LAUS}) = 1$; record (LAUS DEO) is moved to cell 1.

(k, d)	$h(k)$		KEY	DATA		KEY	DATA		KEY	DATA
JAVA PGM	3	0	DROP	BOX	0	DROP	BOX	0	DROP	BOX
MATH LAB	8	1	PILI	NUT	1	PILI	NUT	1	LAUS	DEO
BODY FIT	3	2	BODY	FIT	2			2	PILI	NUT
DROP BOX	0	3	JAVA	PGM	3	JAVA	PGM	3	JAVA	PGM
PILI NUT	3	4			4			4		
OVER PAR	0	5			5			5		
PULP BIT	7	6	SINE	DIE	6	SINE	DIE	6	SINE	DIE
SINE DIE	7	7	PULP	BIT	7	PULP	BIT	7	PULP	BIT
ZINC ORE	12	8	MATH	LAB	8	MATH	LAB	8	MATH	LAB
LAUS DEO	1	9			9			9		
		10	LAUS	DEO	10	LAUS	DEO	10		
		11	ZINC	ORE	11	ZINC	ORE	11	ZINC	ORE
		12	OVER	PAR	12	OVER	PAR	12	OVER	PAR

Figure 15.13 Deletion from an open-address hash table (linear probing)

We take note of the following observations pertaining to the procedure illustrated in Figure 15.13.

1. Let i be the address of the cell originally occupied by the deleted record, and let l be the address of the nearest vacant cell; then the records in cells $(i - 1) \bmod m$ through $(l + 1) \bmod m$ must be checked for possible re-positioning in the table.
2. Let j be the address of the cell containing the record, say (k, d) , to be checked, and let r be its hash address, i.e., $h(k) = r$. Then, (k, d) stays in cell j if r lies cyclically between i and j , i.e., $j \leq r < i$ (Case 1) or $r < i < j$ (Case 2) or $i < j \leq r$ (Case 3), as depicted in Figure 15.14. Otherwise, (k, d) is moved to cell i .
3. In the example above, the records (DROP,BOX), (OVER,PAR) and (ZINC,ORE) are illustrative of Cases 1, 2 and 3, respectively.

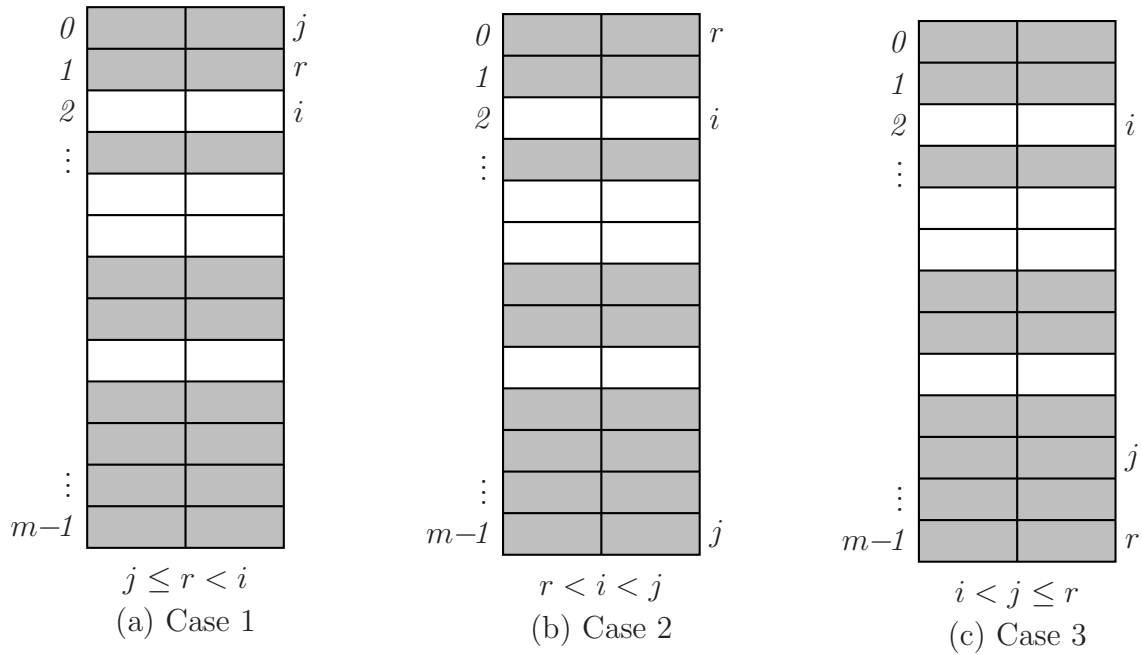


Figure 15.14 Record in cell j whose hash address is r is **not** affected by deletion of record in cell i

The following EASY procedure formalizes the process described above.

```

1  procedure OPEN_ADDRESS_HASH_TABLE_DELETE( $T, K$ )
▷ Deletes record in cell  $i$  of a hash table built using linear probing to resolve collisions.
▷ The procedure rearranges the records in the table such that it becomes what it should
▷ be if the deleted record were not inserted in the first place.
2   $i \leftarrow \text{OPEN\_ADDRESS\_HASH\_TABLE\_SEARCH}(T, K)$ 
3  if  $i < 0$  then return
4  loop
5     $\text{KEY}(i) \leftarrow 0$ 
6     $j \leftarrow i$ 
7  A:  $j \leftarrow j - 1$ ; if  $j < 0$  then  $j \leftarrow j + m$ 
8    if  $\text{KEY}(j) = 0$  then return
9     $r \leftarrow h(\text{KEY}(j))$ 
10   if  $(j \leq r < i)$  or  $(r < i < j)$  or  $(i < j \leq r)$  then goto A
11   else [ $\text{KEY}(i) \leftarrow \text{KEY}(j)$ 
12          $\text{DATA}(i) \leftarrow \text{DATA}(j)$ 
13          $i \leftarrow j$ ]
14 forever
15 end OPEN_ADDRESS_HASH_TABLE_DELETE

```

Procedure 15.8 Deletion from an open-address hash table (linear probing)

15.4.5 Ordered open-address hash tables

Earlier we noted that if we maintain the chains in a chained hash table in order by key, then unsuccessful searches and insertions (which are always preceded by an unsuccessful search) are serviced faster. This is because we do not have to get to the end of a chain to discover that what we are looking for is not in the table. In an open-address hash table, say T , an unsuccessful search is indicated when while following the probe sequence of some search key K we encounter a vacant cell (line 7 of Procedure 15.7). Is it possible to know *sooner*, even before we encounter a vacant cell, that there is no record in T whose key is K ? The answer is ‘Yes’, if we build T as an **ordered hash table**.

Inserting a record into an ordered open-address hash table

Suppose we want to insert the record (k, d) into a hash table. In Procedure 15.6, we start from $h(k)$ and follow k ’s probe sequence, passing by occupied cells, until we find a vacant cell in which to store the record. Suppose that, instead of just ‘passing by’ an occupied cell, we compare k with the key of the record, say (\hat{k}, \hat{d}) , in the occupied cell. If we find that $k < \hat{k}$, we move on; but if we find that $k > \hat{k}$, then we ‘kick out’ (\hat{k}, \hat{d}) from its slot and store (k, d) instead. Now we proceed to re-insert (\hat{k}, \hat{d}) , following \hat{k} ’s probe sequence, back into the table *using the same strategy*. If we start with an empty table and apply this method each time we insert a record, we obtain an ordered hash table. Procedure ORDERED_HASH_TABLE_INSERT given below formalizes the process.

```

1  procedure ORDERED_HASH_TABLE_INSERT( $T, k, d$ )
  ▷ Inserts a new record  $(k, d)$  into an ordered hash table. Collisions are resolved
  ▷ by open addressing. Insert operation is aborted if the table is full or if there
  ▷ is already a record in the table with the same key.
2  if  $n = m - 1$  then [output ‘Table is full.’; stop]
3   $i \leftarrow h(k)$     ▷ initial hash address
4  loop
5    case
6      : $KEY(i) = k$ : [output ‘Duplicate key found.’; stop]
7      : $KEY(i) > 0$ : [if  $KEY(i) < k$  then [ $KEY(i) \leftrightarrow k$ ;  $DATA(i) \leftrightarrow d$ ]    ▷ swap records
8                     $i \leftarrow i - p(k)$ ; if  $i < 0$  then  $i \leftarrow i + m$ ]
9      : $KEY(i) = 0$ : [ $KEY(i) \leftarrow k$ ;  $DATA(i) \leftarrow d$ ;  $n \leftarrow n + 1$ ; return]
10   endcase
11 forever
12 end ORDERED_HASH_TABLE_INSERT

```

Procedure 15.9 Insertion into an ordered open-address hash table

Figures 15.15(a) through (c) shows three snapshots of the ordered hash table that is generated when the first 11 records in Figure 15.2, with numeric equivalents as given in Figure 15.3, are inserted into an initially empty hash table of size $m = 19$ using the division method, $h(k) = k \bmod m$, with double hashing to resolve collisions. The secondary hash function is $h'(k) = 1 + k \bmod (m - 2)$. The actions which produced each of the snapshots

depicted in the figure are described in detail below.

- (a) FOLK, TEST, BACK, ROAD and HOME hash to 10, 16, 2, 7 and 8, respectively; these are vacant cells, so the records are stored where they hash to. CREW collides with FOLK in cell 10; since $\text{CREW} < \text{FOLK}$, FOLK stays. Next cell in probe sequence of CREW is $(10 - 17) \bmod 19 = 12$ which is vacant, so CREW is stored in cell 12. FAUX hashes to 15 and is stored there.
- (b) OPUS hashes to 15 and collides with FAUX. Since $\text{OPUS} > \text{FAUX}$, FAUX is kicked out and OPUS takes the slot. Next cell in probe sequence of FAUX is $(15 - 3) \bmod 19 = 12$ which is occupied by CREW. Since $\text{FAUX} > \text{CREW}$, CREW is kicked out and FAUX takes the slot. Next cell in probe sequence of CREW is $(12 - 17) \bmod 19 = 14$ which is vacant, so CREW is stored there.
- (c) SODA hashes to 0 and is stored there. DEEP hashes to 2 which is occupied by BACK. Since $\text{DEEP} > \text{BACK}$, BACK is kicked out and DEEP takes the slot. Next cell in probe sequence of BACK is $(2 - 2) \bmod 19 = 0$ which is occupied by SODA which stays put since $\text{BACK} < \text{SODA}$. Next cell in probe sequence of BACK is $(0 - 2) \bmod 19 = 17$ which is vacant, so BACK is stored there. Finally, MENU hashes to 18, which is vacant, so it is stored there.

(k, d)	$h(k)$	$h'(k)$	KEY	DATA	KEY	DATA	KEY	DATA
FOLK ART	10	12	0		0		0	SODA POP
TEST BIT	16	15	1		1		1	
BACK END	2	2	2	BACK	2	BACK	2	DEEP FRY
ROAD MAP	7	6	3		3		3	
HOME RUN	8	9	4		4		4	
CREW CUT	10	17	5		5		5	
FAUX PAS	15	3	6		6		6	
OPUS DEI	15	5	7	ROAD	7	ROAD	7	ROAD
SODA POP	0	7	8	HOME	8	HOME	8	HOME
DEEP FRY	2	15	9		9		9	
MENU BAR	18	4	10	FOLK	10	FOLK	10	FOLK
			11		11		11	
			12	CREW	12	FAUX	12	FAUX
			13		13		13	
			14		14	CREW	14	CREW
			15	FAUX	15	OPUS	15	OPUS
			16	TEST	16	TEST	16	TEST
			17		17		17	BACK
			18		18		18	MENU

Figure 15.15 Ordered hash table (with double hashing to resolve collisions)

Searching for a record in an ordered open-address hash table

```

1  procedure ORDERED_HASH_TABLE_SEARCH( $T, K$ )
  ▷ Searches for a record in an ordered hash table built using open addressing to resolve
  ▷ collisions. Procedure returns index of matching record, if found; or  $-1$ , otherwise.
2     $i \leftarrow h(k)$       ▷ initial hash address
3     $s \leftarrow p(k)$     ▷ probe decrement
4    loop
5      case
6        : $KEY(i) = K$  : return( $i$ )
7        : $KEY(i) < K$  : return ( $-1$ )
8        : $KEY(i) > K$  : [ $i \leftarrow i - s$ ; if  $i < 0$  then  $i \leftarrow i + m$ ]
9      endcase
10   forever
11 end ORDERED_HASH_TABLE_SEARCH

```

Procedure 15.10 Search in an ordered open-address hash table

The condition $KEY(i) < K$ in line 7 indicates that there is no record in the table whose key is K . If such a record were previously inserted it must have kicked out the record in cell i whose key is less than K . Since we find cell i occupied by a record whose key is less than K , we conclude that K is not in the table. In an ordered hash table, we need not encounter a vacant cell with $KEY(i) = 0$ (see line 7 of Procedure 15.6), to know that the search is unsuccessful; thus unsuccessful searches are serviced faster than they would be in an otherwise unordered hash table.

What about successful searches? Are they serviced faster if the hash table were ordered than if it were not? The answer to this question hinges on a remarkable property of an ordered hash table, namely: *Given a set of n records, the **same** ordered hash table is generated regardless of the sequence in which the records are inserted into the table.* For instance, if we insert the keys in Figure 15.15 in ascending order (BACK, CREW, DEEP, FAUX, FOLK, HOME, MENU, OPUS, ROAD, SODA, TEST) or in descending order (TEST, SODA, ROAD, OPUS, MENU, HOME, FOLK, FAUX, DEEP, CREW, BACK) or in any order at all, we will obtain the same ordered hash table shown in Figure 15.15(c) [verify]. It follows from this that each record has a fixed position in an ordered hash table. Thus the number of probes that it takes a key, say K , to get to its final position in the table from its initial hash address $h(K)$ is the *same* regardless of when the key is inserted and regardless of whether in the process it gets repeatedly kicked out, and must resume its probe, or not. Now imagine that the records are inserted into an ordered hash table in *descending* order; each record will follow its own probe sequence to its final resting place in the table, and none will ever be kicked out. This is exactly what will happen if the records were inserted into an unordered hash table, taking exactly the same number of probes. Since the number of probes it takes to insert a record is the same number of probes it takes to locate the record, it follows that the average number of probes in a successful search is the same for both an ordered and an unordered hash table.

15.4.6 Analysis of hashing with collision resolution by open addressing

The analysis of open addressing requires mathematics which is beyond the scope of this book. We therefore simply give a summary of the results; the interested student will find the detailed derivation and other relevant information in the references cited in the Bibliographic Notes for this session.

In the formulas listed below, C_n denotes the average number of probes in a successful search, C'_n denotes the average number of probes in an unsuccessful search, and $\alpha = \frac{n}{m} < 1$ is the load factor.

1. Linear probing (open addressing with primary clustering)

$$C_n \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \quad (15.20)$$

$$C'_n \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right) \quad (15.21)$$

2. Quadratic probing (open addressing with secondary clustering)

$$C_n \approx 1 - \ln(1 - \alpha) - \frac{\alpha}{2} \quad (15.22)$$

$$C'_n \approx \frac{1}{1 - \alpha} - \alpha - \ln(1 - \alpha) \quad (15.23)$$

3. Double hashing (open addressing with no primary and secondary clustering)

$$C_n \approx \frac{1}{\alpha} \ln \left(\frac{1}{1 - \alpha} \right) \quad (15.24)$$

$$C'_n \approx \frac{1}{1 - \alpha} \quad (15.25)$$

What do these formulas signify? If we impose an upper limit on the load factor, if necessary by expanding the table such that n/m does not exceed a predetermined value, then hash table search takes *constant time* on average. For instance, if we use double hashing and if we do not allow the table to become more than half full, i.e., $\alpha \leq 0.50$, then the expected number of probes in a successful search is at most 1.39 and the expected number of probes in an unsuccessful search is at most 2. These values apply if we have 10 records in a table of size 20, or 1000 records in a table of size 2000, or 1 million records in a table of size 2 million. In all these cases it takes no more than 2 comparisons on average to find what we are looking for or to discover that what we are looking for is not in the table. Clearly, a hash table provides an extremely efficient implementation of the table ADT insofar as the three basic operations, namely, search, insert and delete, are concerned. Note, however, that other table operations, such as finding the record with the next smaller or next larger key or enumerating all the records in the table in ascending order by key, are not supported in any efficient way by a hash table.

Summary

- What sets a hash table apart from other implementations of the table ADT is the novel idea of storing a record at a position in the table, called its *hash address*, that is a function of its key. The only hitch in this otherwise nifty idea is that two or more keys may hash to the same position in the table, a phenomenon referred to as a *collision*.
- In practice, no matter how good the hash function that is used may be, collisions inevitably occur and must be resolved. The von Mises birthday paradox provides a mathematical basis to this empirically observed behavior. Two well-studied collision resolution policies are *chaining* and *open addressing*.
- In collision resolution by chaining all the records which hash to the same table address are constituted into a linked list called a chain. There is a chain, some possible null, for each table address; thus the hash table is essentially an array of linked lists.
- Assuming that the hash function is a good one in that it uniformly distributes the records over the table addresses, the average length of a chain is n/m , where m is the table size and n is the number of records in the table. The quantity n/m is called the *load factor*; in a chained hash table it can be less than, equal to or greater than 1.
- In collision resolution by open addressing all the records are stored in a sequential table, obviating the need for pointers. When a record hashes to a position in the table that is already occupied, the table is probed for a vacant slot, if any, in which to store the new record. Positions in the table are inspected in *decrements* of s , doing a wraparound when the probe falls off the lower end (addresswise) of the table.
- In *linear probing* the probe decrement s is simply taken to be 1; thus, keys which hash to the same address in the table follow the same probe sequence. This leads to a phenomenon called *primary clustering* in which long consecutive positions in the table are occupied by keys, leading to longer and longer probes as the table becomes full.
- In *double hashing* a secondary hash function is applied on the key to yield the probe decrement s ; thus, keys which initially hash to the same address in the table now follow their own individual probe sequences, preventing the formation of clusters.
- As with a chained hash table, the load factor α for an open-address hash table is defined to be n/m ; unlike in a chained hash table, the value of α cannot exceed 1 in an open-address hash table.
- A rigorous analysis of these collision resolution strategies indicate that the average search time on a hash table is not dependent, *per se*, on the number of records in the table, but on the load factor α . If the value of α is maintained, say by extending the table if needs be, then the search and insert operations take constant time on the average, which is as good as one could expect.

Exercises

- Express the key ANALPHABETICKEY as a radix-256 integer by replacing the letters by their ASCII codes and combining 32-bit words using the exclusive-or operator.
- Redo the table in Figure 15.4 for $m = 2^7 = 256$. Explain the dismal results.
- Redo the table in Figure 15.7 for $m = 2^7 = 256$.
- Using the multiplicative method for hashing with $\theta = \phi^{-1} = 0.6180339887$, show the chained hash table that results after inserting the first 20 records in Figure 15.2 into an initially empty table of size $2^4 = 16$.
- Rewrite procedure CHAINED_HASH_TABLE_INSERT (Procedure 15.2) such that the chains are sorted in increasing order by key.
- Rewrite procedure CHAINED_HASH_TABLE_SEARCH (Procedure 15.3) given that the chains are sorted in increasing order by key.
- Using the multiplicative method for hashing with $\theta = \phi^{-1} = 0.6180339887$, show the open-address hash table that results after inserting the first 20 records in Figure 15.2 into an initially empty table of size $2^5 = 32$ using linear probing to resolve collisions.
- Find the complete probe sequence generated by *quadratic residue search* for $m = 23$ beginning with $h(K) = 17$. Describe the resulting sequence.
- Find the complete probe sequence generated by *double hashing* using the function

$$[h(K) - j * h'(K)] \bmod m, \quad 0 \leq j \leq m - 1$$

where $h(K) = K \bmod m$ and $h'(K) = 1 + K \bmod (m - 2)$ for $K = 23786$ and $m = 19$. Characterize the resulting sequence.

- Show the open-address hash table that results after inserting records 21 through 35 in Figure 15.2 into an initially empty table of size $m = 19$ using double hashing to resolve collisions. Use $h(K) = K \bmod m$ as the primary hash function and $h'(K) = 1 + \left\lfloor \frac{K}{m} \right\rfloor \bmod (m - 2)$ as the secondary hash function.
- Assume that deletions in an open-address hash table are implemented the ‘lazy’ way, i.e., by marking a cell previously occupied by a deleted record as *vacated*. Modify procedure OPEN_ADDRESS_HASH_TABLE_INSERT (Procedure 15.6) and procedure OPEN_ADDRESS_HASH_TABLE_SEARCH (Procedure 15.7) to take into account the presence of both vacant and vacated cells.
- Show the *ordered* hash table that results after inserting records 21 through 35 in Figure 15.2 into an initially empty table of size $m = 19$ using double hashing to resolve collisions. Use $h(K) = K \bmod m$ as the primary hash function and $h'(K) = 1 + \left\lfloor \frac{K}{m} \right\rfloor \bmod (m - 2)$ as the secondary hash function.

13. Redo Item 12, but insert the records in alphabetical order from AVIV TEL to QUIZ BEE.
14. Redo Item 12, but insert the records in reverse alphabetical order from QUIZ BEE to AVIV TEL.

15. The table shown was constructed using the hash function $h(K) = K \bmod m$, where K is as given in Figure 15.3 and $m = 13$. Collisions were resolved using linear probing. Show the resulting table after record TOOL BOX is deleted and the table is rearranged to make the vacated cell reusable.

	KEY	DATA
0	HOLY	SEE
1	ALSO	RAN
2	OPEN	AIR
3	TOOL	BOX
4	QUIZ	BEE
5		
6	EVIL	EYE
7	HALF	WIT
8		
9	IRAQ	WAR
10	TAXI	CAB
11	UNIX	JOB
12	BALI	HAI

16. Using a language of your choice, transcribe procedure CHAINED_HASH_TABLE_INSERT (Procedure 15.2) into a running program. Using your program, solve Item 4 anew.
17. Using a language of your choice, transcribe procedure OPEN_ADDRESS_HASH_TABLE_INSERT (Procedure 15.6) into a running program. Using your program, solve Items 7 and 10 anew.
18. Using a language of your choice, transcribe procedure ORDERED_HASH_TABLE_INSERT (Procedure 15.9) into a running program. Using your program, solve Items 12, 13 and 14 anew.
19. Using a language of your choice, transcribe procedure OPEN_ADDRESS_HASH_TABLE_DELETE (Procedure 15.8) into a running program. Using your program, solve Item 15 anew.

Bibliographic Notes

Hashing and hash tables are discussed in most textbooks on Data Structures and on Algorithms. KNUTH3[1998], CORMEN[2001] and STANDISH[1994] are excellent sources. The division and multiplicative methods for hashing are covered in all three; the graphical interpretation of Theorem S, on which the multiplicative method is based, is from STANDISH[1980], p. 162. The algorithms for searching, insertion and deletion in chained hash tables, implemented as procedures CHAINED_HASH_TABLE_SEARCH, CHAINED_HASH_TABLE_INSERT and CHAINED_HASH_TABLE_DELETE, are based on familiar operations on linked lists. A rigorous proof of Eq.(15.11) and Eq.(15.13) for the average

number of key comparisons in an unsuccessful and successful search, respectively, on a chained hash table is found in CORMEN[2001], pp. 226–228.

The algorithms for searching and insertion in open-address hash tables which utilize either linear probing or double hashing, implemented as procedures OPEN_ADDRESS_HASH_TABLE_SEARCH and OPEN_ADDRESS_HASH_TABLE_INSERT, are pretty straightforward. Deletion, with subsequent rearrangement of the table entries, requires more ingenuity. Procedure OPEN_ADDRESS_HASH_TABLE_DELETE is an implementation of Algorithm R (*Deletion with linear probing*) given in KNUTH3[1998], pp. 533–534; Knuth remarks that *no* analogous algorithm for deletion with double hashing is possible.

Ordered hash tables were put forth by Amble and Knuth; ordered hashing is discussed in STANDISH[1980], pp. 149–153. Procedures ORDERED_HASH_TABLE_SEARCH and ORDERED_HASH_TABLE_INSERT are implementations of Algorithm 4.6 (*Ordered hash table search by open addressing*) and Algorithm 4.7 (*Key insertion in an ordered hash table*) given in this reference.

A detailed analysis of the algorithms for open addressing is given in KNUTH3[1998], pp. 534–539 and in CORMEN[2001], pp. 241–244. Equations 15.20 through 15.25 are from the former.

An interesting exposition on the von Mises birthday paradox, with the underlying mathematics, is found in STANDISH[1994], pp. 462–465. The value 0.50729723, which is the probability that two persons in a group of 23 will have the same birthday, is from this reference.

Bibliography

- [AHO[1983]] Aho, A.V., Hopcroft, J.E. and Ullman, J.D. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [AHO[1974]] Aho, A.V., Hopcroft, J.E. and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [BAASE[1988]] Baase, Sara. *Computer Algorithms: Introduction to Design and Analysis*, 2nd ed. Addison-Wesley, Reading, MA, 1988.
- [COHEN[1978]] Cohen, Daniel I A. *Basic Techniques of Combinatorial Theory*. John Wiley and Sons, Inc., New York, 1978.
- [COLLINS[1992]] Collins, William. *Data Structures: An Object-Oriented Approach*. Addison-Wesley, Reading, MA, 1992.
- [CORMEN[2001]] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA, 2001.
- [EVEN[1979]] Even, Shimon. *Graph Algorithms*. Computer Science Press, Rockville, Maryland, 1979.
- [GOODRICH[1998]] Goodrich, M.T. and Tamassia, R. *Data Structures and Algorithms in JAVA*. John Wiley and Sons, Inc., New York, 1998.
- [GRIMALDI[1994]] Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 3rd ed. Addison-Wesley, Reading, MA, 1994.
- [HOROWITZ[1976]] Horowitz, E. and Sahni, S. *Fundamentals of Data Structures*. Computer Science Press, Rockville, Maryland, 1976.
- [KNUTH1[1997]] Knuth, Donald E. *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, 3rd ed. Addison-Wesley, Reading, MA, 1997.
- [KNUTH2[1998]] Knuth, Donald E. *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*, 3rd ed. Addison-Wesley, Reading, MA, 1998.

524 BIBLIOGRAPHY

- [KNUTH3[1998]] Knuth, Donald E. *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, 2nd ed. Addison-Wesley, Reading, MA, 1998.
- [KORFHAGE[1987]] Korfhage, R.R. and Gibbs, N.E. *Principles of Data Structures and Algorithms with Pascal*. Brown Publishers, Dubuque, Iowa, 1987.
- [LEWIS[1976]] Lewis, T.G. and Smith, M.Z. *Applying Data Structures*. Houghton Mifflin Company, Boston, 1976.
- [MORET[1991]] Moret, B.M.E. and Shapiro, H.D. *Algorithms from P to NP*, Vol. I: *Design and Efficiency*. Benjamin Cummings, Redwood City, CA, 1991.
- [NIEVERGELT[1993]] Nievergelt, J. and Hinrichs, K. *Algorithms and Data Structures*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
- [SEEDGEWICK[1990]] Sedgewick, Robert. *Algorithms in C*. Addison-Wesley, Reading, MA, 1990.
- [STANDISH[1994]] Standish, Thomas A. *Data Structures, Algorithms, and Software Principles*. Addison-Wesley, Reading, MA, 1994.
- [STANDISH[1980]] Standish, Thomas A. *Data Structure Techniques*. Addison-Wesley, Reading, MA, 1980.
- [TENENBAUM[1986]] Tenenbaum, A.M. and Augenstein, M.J. *Data Structures Using Pascal*, 2nd. ed. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [TREMBLAY[1976]] Tremblay, J.P. and Sorenson, P.G. *An Introduction to Data Structures with Applications*. McGraw-Hill, New York, 1976.
- [WEISS[1997]] Weiss, Mark A. *Data Structures and Algorithm Analysis in C*, 2nd ed. Addison-Wesley, Menlo Park, CA, 1997.

Index

- Ω -notation, 66
- Θ -notation, 67
- ϕ (golden ratio), 25
- abstract data type, 4
 - binary tree, 144–146
 - deque, 132
 - generalized list, 400, 401
 - hash table, 489, 490
 - linear list, 338, 339
 - queue, 111
 - stack, 4, 75, 76
 - table, 424, 425
 - tree, 211–213
- abstract data type, implementing an
 - linked design, 9
 - sequential design, 8
- address range discrimination, 406
- Adel'son-Vel'skii, G.M., 462
- algorithm
 - analyzing an, 59
 - communicating an, 50
 - requisites, 5
- antisymmetry, 30
- APSP problem, 312, 325
- arithmetic series, 35
- arithmetic tree representations, 224
- articulation point, 277
- asymptotic notations, 62
 - Ω -notation, 66
 - Θ -notation, 67
 - O -notation, 63
- atomic node, 402
- automatic storage reclamation
 - garbage-collection technique, 413
 - reference-counter technique, 414
- AVL tree, 462
 - AVL_INSERT, 474
 - deletions in, 476
 - ROTATE_LEFT, 474
 - ROTATE_LEFT_RIGHT, 474
 - ROTATE_RIGHT, 474
 - ROTATE_RIGHT_LEFT, 475
 - rotations on, 464
- best-fit method, 368, 372
- biconnected components, 277
- binary relations, 29
- binary search, 431
 - analysis of, 433–435
 - multiplicative version, 435
- binary search tree, 448
 - average BST, 458
 - AVL tree, 462
 - BST_DELETE, 454
 - BST_INSERT, 451
 - BST_MINKEY, 451
 - BST_SEARCH, 450
 - DISPLAY_OPTIMUM_BST, 484
 - height of, 449
 - height-balanced BST, 461
 - maximal BST, 457
 - minimal BST, 456
 - optimum BST, 477
 - OPTIMUM_BST, 482
- binary trees
 - b_n , 146
 - a nomenclature for, 147
 - COPY, 161
 - degree, 144
 - EQUIVALENT, 162, 172
 - expression tree, 152
 - height, 145
 - INORDER_THREADED, 167

- internal and external nodes, 148
- level, 145
- path, 145
- POSTORDER, 154, 158, 160
- PREORDER, 154, 156
- PREORDER_BY_LINK_INVERSION, 174
- PREORDER_SIKLÓSSY, 177
- representation in memory, 153
- Siklóssy traversal, 175
- similarity and equivalence, 171
- some operations on, 145
- some properties of, 146
- threaded, 162
- traversal by link inversion, 172
- traversals, 149
- binary trees, applications of, 184
 - calculating conveyance losses in an irrigation network, 184
 - CALCULATE, 191
 - COMPUTE_IDR, 191
 - CONSTRUCT, 190
 - PRINT_IDR, 191
- heapsort, 192
- priority queues, 202
- birthday paradox, 499
- bit map, 406
- boundary-tag technique, 375, 377
- breadth first search (BFS), 265
- bubble sort, 39
- buddy, 378, 379
- canonical form, polynomial in, 348
- Cayley's theorem, 288
- ceiling function, 22
- chaining, 499
 - analysis of, 503
- circular deque, 135
- circular list, 343, 345
- circular queue, 115
- class 1 vertex, 312
- class 2 vertex, 312
- collapsing problem, 373
- collapsing rule for the find operation, 235
- collision resolution policy, 490
 - chaining, 499
 - open addressing, 504
- combinations, 34
- comparative growth of functions, 69
- complete binary tree, 148
 - sequential representation, 196
- complexity classes, 69
- converting keys to numbers, 491
- data abstraction, 13
- data structure, 4
- data type, 3
- depth first search (DFS), 253
- deque
 - as an ADT, 132
 - DELETE_DEQUE, 135, 137, 139
 - INSERT_DEQUE, 134, 137, 138
 - linked representation, 138
 - MOVE_DEQUE, 134
 - sequential implementation
 - circular deque, 135
 - straight deque, 133
- Dijkstra's algorithm, 312
 - analysis of, 325
 - correctness of, 317
 - DIJKSTRA, 320
 - Dijkstra's rule, 316
 - DISPLAY_PATH, 324
 - example, 314, 321
 - InitPQ, 320
 - priority queue for, 319
 - shortest-paths tree, 316, 317
- direct predecessors, count of, 126, 127
- direct successors, list of, 126, 127
- direct-address table, 425
- directed acyclic graph (dag), 271
- disjoint set ADT, 239
- division method, 493
- double hashing, 508
- doubly-linked linear list, 345
- dynamic programming, 327
- dynamic storage management, 367
 - BEST_FIT, 372
 - BOUNDARY_TAG_TECHNIQUE, 377
 - buddy-system methods, 378
 - BUDDY_SYSTEM_LIBERATION, 387
 - BUDDY_SYSTEM_RESERVATION, 386
 - FIRST_FIT_1, 370
 - FIRST_FIT_2, 371

- sequential-fit methods, 368, 373
 - SORTED_LIST_TECHNIQUE, 374
- equivalence classes, 229, 246, 247
- equivalence problem, 305
 - collapsing rule for find, 235
 - disjoint set ADT, 239
 - EQUIVALENCE, 232, 237
 - FIND, 236
 - path compression, 235
 - TEST, 232, 237
 - UNION, 234
 - weighting rule for union, 233
- equivalence relation, 31, 228
- Euclid's algorithm, 5
- exam-scheduling problem, 2, 12
 - an instance of, 12
 - an optimal solution, 15
- exclusive or, \oplus , 175
- exponentials, 23
- extended binary tree, 148, 433
- external fragmentation, 370
- external nodes, 148, 433
- external path length, 434
- EASY, 51
 - parameter passing in, 55
 - program structure, 54
- factorials, 25
 - Stirling's approximation, 25
- family-order sequential representation, 221
- farm irrigation requirement, *FIR*, 184
- Fibonacci numbers, 25
- Fibonacci tree, 442
- Fibonacci search, 441
- FIND, 236, 306
- first-fit method, 368, 370, 371
- Fischer, M.J., 229
- floor function, 22
- Floyd's algorithm, 325, 326
 - analysis of, 330
 - DISPLAY_PATH, 330
 - example, 326
 - FLOYD, 328
 - shortest-paths tree, 329
- Floyd, R.W., 198
- folding, 493
- free tree, 213, 282
- fringe vertex, 265, 292
- full binary tree, 148
- Galler, B.A., 229
- garbage-collection technique, 413
- generalized lists
 - CONSTRUCT, 409
 - garbage-collection technique, 413
 - GATHER, 420
 - HEAD, 407
 - head and tail, 400
 - length and depth, 400
 - MARK_LIST_1, 417
 - MARK_LIST_2, 417
 - MARK_LIST_3, 419
 - marking and gathering, 416
 - reference-counter technique, 414
 - representation in computer
 - memory, 402–406
 - TAIL, 408
 - TRAVERSE_PURE_LIST, 411
 - TRAVERSE_THREADED_PURE_LIST, 412
- geometric series, 36
- GETNODE, 11
- golden ratio, 25
- graph coloring, 14
 - approximate algorithm for, 15
- graph traversals, 252
 - analysis of BFS, 270
 - analysis of DFS, 264
 - back edge, 253, 258, 270
 - BFS, 269
 - BFS_DRIVER, 270
 - breadth first search, 265
 - breadth-first forest, 268
 - breadth-first tree, 267
 - cross edge, 253, 258, 270
 - depth first search, 253
 - depth-first forest, 257
 - depth-first tree, 257
 - descendants in a depth-first
 - tree, 259, 260
 - DFS, 263
 - DFS_DRIVER, 264
 - discovery edge, 253
 - discovery time, 256

- finishing time, 256
- forward edge, 253, 258, 270
- fringe vertex, 265
- non-discovery edge, 253
- predecessor subgraph, 256
- tree edge, 253, 258, 270
- graphs
 - acyclic graph, 246
 - acyclic undirected graph, 283
 - articulation point, 247, 277, 280
 - ARTICULATION_POINTS, 280
 - biconnected components, 277, 280
 - biconnected undirected graph, 247
 - complete graph, 245
 - component graph, 276
 - connected components, 246
 - connected undirected graph, 246
 - cycle, 246
 - DAG, algorithm, 272
 - degree, 245
 - dense graph, 245
 - directed acyclic graph (DAG), 271
 - directed graph, 245
 - FIND_ARTICULATION_POINTS, 281
 - in-degree, 245, 250
 - minimum cost spanning tree, 248
 - out-degree, 245, 250
 - path, 246
 - simple cycle, 246
 - simple path, 246
 - spanning tree, 248
 - sparse graph, 245
 - strongly connected components, 247, 274
 - strongly connected directed graph, 247
 - STRONGLY_CONNECTED_COMPONENTS, 275
 - topological sorting, 272
 - TOPOLOGICAL_SORT, algorithm, 273
 - TRANSPOSE, 274
 - undirected graph, 244
 - weighted graph, 248
- graphs, representation of
 - adjacency lists, 250
 - adjacency matrix, 249
- greedy algorithm, 18, 289
- halting problem, 1
- harmonic numbers, 37
- hash function, 490
 - division method, 493
 - folding, 493
 - midsquare method, 493
 - multiplicative method, 496
 - requisites of a good, 493
- hash tables
 - CHAINED_HASH_TABLE_DELETE, 502
 - CHAINED_HASH_TABLE_INSERT, 501
 - CHAINED_HASH_TABLE_SEARCH, 502
 - CHAINED_HASH_TABLE_INIT, 501
 - OPEN_ADDRESS_TABLE_DELETE, 514
 - OPEN_ADDRESS_TABLE_INSERT, 511
 - OPEN_ADDRESS_TABLE_SEARCH, 512
 - OPEN_ADDRESS_TABLE_INIT, 510
 - ORDERED_HASH_TABLE_INSERT, 515
 - ORDERED_HASH_TABLE_SEARCH, 517
- heap, 192
 - heap-order property, 192
 - sift-up, 193
- heapsort
 - algorithm, 198
 - analysis of, 201
 - HEAPIFY, 195, 197
 - HEAPSORT, 199
- inorder predecessor, 166
- inorder successor, 165
- inorder traversal, 149
- insertion sort, 59
 - analysis of, 61
 - implementation in EASY, 60
- internal fragmentation, 371
- internal nodes, 148, 433
- internal path length, 434
- irreflexivity, 30
- irrigation diversion requirement, *IDR*, 184
- Kruskal's algorithm, 301

- analysis of, 311
- example, 301, 307
- InitPQ, 306
- KRUSKAL, 306
- priority queue for, 305
- Landis, E.M., 462
- level, 145, 212
- level order with degrees/weights, 225
- level-order, 149
- level-order sequential representation, 222
- liberation in DSM, 367, 373, 378
- linear lists
 - as an ADT, 338, 339
 - circular, 343, 345
 - doubly-linked, 345
 - dynamic storage management, 367
 - linked representation, 339
 - multiple-precision integer arithmetic, 353
 - polynomial arithmetic, 347
 - sequential representation, 339
 - UPCAT processing, 388
 - with list head, 342, 345
- linear probing, 506
- linear search, 428
 - analysis of, 429–431
- linked design
 - memory pool, 11
 - node structure, 9
 - nodes, a C implementation, 11
- LISP, 400, 405
- list head, 341
- list inversion in-place, 58
- list node, 402
- load factor, 500, 518
- logarithms, 24
- long integers, 353
 - ADD_LONGINTEGERS, 362
 - addition of, 356
 - DELETE_LEADING_ZEROS, 363
 - INIT_PRODUCT, 365
 - INSERT_AT_TAIL, 361
 - INSERTD_AT_HEAD, 363
 - IsGT, 363
 - multiplication of, 358
 - MULTIPLY_LONGINTEGERS, 365
 - READ_LONGINTEGER, 361
 - representation in computer memory, 359
 - ZERO_LONGINTEGER, 361
- marking and gathering, 413
- memory pool, 367
 - GETNODE, 11
 - RETNODE, 11
- midsquare method, 493
- mod function, 22
- MST theorem, 289
- multiple-precision integer arithmetic
 - see *long integers*, 353
- multiplicative method, 496
- natural correspondence, 214
 - CONVERT, 223
- O-notation, 63
 - preponderance of, 68
- open addressing, 504
 - analysis of, 518
 - double hashing, 508
 - linear probing, 506
 - quadratic probing, 507
- optimal-fit method, 368
- ordered hash table, 515
- ordered tree, 212, 213
- oriented tree, 212, 213, 229
- partial order, 31, 124
- partition of a set, 28
- path, 145, 246
- path compression, 235
 - analysis of, 236
- permutations, 33
- polynomial arithmetic, 347
 - POLYADD, 350
 - POLYMULT, 352
 - POLYREAD, 349
 - POLYSUB, 351
 - ZEROPOLY, 349
- polynomials, 23
- postorder traversal, 149
- postorder with degrees/weights, 224
- powerset, 46, 72
- preorder sequential representation, 219
- preorder successor, 166, 167
- preorder traversal, 149

- preorder with degrees/weights, 224
- Prim's algorithm, 289
 - analysis of, 300
 - as an application of BFS, 292
 - DECREASE_KEY, 296
 - DISPLAY_MST, 299
 - example, 290, 297
 - EXTRACT_MIN, 296
 - HEAPIFY, 296
 - InitPQ, 295
 - IsEmptyPQ, 295
 - PRIM, 295
 - Prim's rule, 292
 - priority queue for, 294
- primary clustering, 506
- priority queue
 - as an ADT, 202
 - heap-ordered array implementation, 206
 - in Dijkstra's algorithm, 319
 - in Kruskal's algorithm, 305
 - in Prim's algorithm, 294
 - PQ_EXTRACT, 204, 205, 207
 - PQ_INSERT, 204, 205, 207
 - sorted array implementation, 205
 - unsorted array implementation, 204
- proof by contradiction, 45
- proof by counterexample, 45
- proof by mathematical induction, 42
- pseudocode, 50
- pure list, 401
- quadratic probing, 507
- quadratic residue search, 507
- queue
 - a C implementation, 118, 121
 - as an ADT, 111
 - DEQUEUE, 114, 117, 120, 121, 123
 - ENQUEUE, 114, 117, 119, 120, 123
 - InitQueue, 119, 122
 - IsEmptyQueue, 119, 123
 - linked implementation, 120
 - MOVE_QUEUE, 114
 - sequential implementation
 - circular queue, 115
 - straight queue, 112
 - topological sorting, 124
- recurrences, 38
- recursive list, 401
- reentrant list, 401
 - example, 403
- reference-counter technique, 414
- reflexivity, 29
- reservation in DSM, 367, 368, 378
- RETNODE, 11
- reverse level order, 149
- roving pointer, 371
- rule of product, 32
- rule of sum, 32
- Scheme, 400
- Schorr-Waite-Deutsch algorithm, 417
- secondary clustering, 507
- secondary hash function, 508
- sequential tables, 427
 - binary search, 431
 - BINARY_SEARCH, 432
 - Fibonacci search, 441
 - FIBONACCIAN_SEARCH, 442
 - linear search, 428
 - LINEAR_SEARCH_1, 428
 - LINEAR_SEARCH_2, 429
 - MULTIPLICATIVE_BINARY_SEARCH, 436
 - NEW_INDEX, 440
 - REARRANGE_FOR_MBS, 440
- sequential-fit methods, 368, 373
- sets, 26
 - binary relations, 29
 - Cartesian product, 28
 - partition, 28
- shortest-paths tree, 316, 317, 329
- sift-up, 193
- sorted-list technique, 373, 374
- space complexity, 59
- SPARKS, 51
- SPSP problem, 313
- SSSP problem, 312
- stack
 - a C implementation, 81, 87
 - a Pascal implementation, 79, 86
 - as an ADT, 76
 - InitStack, 78, 84
 - IsEmptyStack, 78, 84
 - linked implementation, 83

- POP, 79, 85
- PUSH, 78, 85
 - sequential implementation, 77
- stackless traversal, 168, 172, 174, 175, 177
- stacks, application of
 - a simple pattern recognition problem, 89
 - converting arithmetic expressions from
 - infix to postfix, 90
- strictly binary tree, 147
- strongly connected components, 274
- summations, 35
 - arithmetic series, 35
 - geometric series, 36
 - harmonic series, 37
 - miscellaneous sums, 37
- symmetry, 29
- threaded binary trees, 162
 - EQUIVALENT, 172
 - inorder predecessor, 166
 - inorder successor, 165
 - INORDER_THREADED, 167
 - INPRED, 166
 - INSERT_RIGHT, 169
 - INSUC, 165
 - preorder successor, 166
 - PRESUC, 167
 - REPLACE_RIGHT, 170
 - representation in memory, 163, 164
 - traversal, 167
- time complexity, 59
- topological sorting, 124, 272
- TOPOLOGICAL_SORT, 130
- total order, 31
- Towers of Hanoi problem, 40
 - algorithm for, 40
 - implementation in EASY, 57
- transitive closure, 331
- transitivity, 29
- trees
 - ANCESTORS, 226
 - arithmetic tree representations, 224
 - DEGREE, 227
 - degree, 212
 - equivalence problem, 228
 - expression tree, 217
 - family-order sequential representation, 221
 - forest, 214
 - forest traversals, 216
 - free tree, 213
 - height, 212
 - level, 212
 - level order with degrees/weights, 225
 - level-order sequential representation, 222
 - linked representation, 218
 - natural correspondence, 214
 - CONVERT, 223
 - ordered tree, 212, 213
 - oriented tree, 212, 213
 - postorder sequence with weights, 224
 - postorder with degrees/weights, 224
 - preorder sequential representation, 219
 - preorder with degrees/weights, 224
- UNION, 234, 306
- UPCAT processing
 - ASSIGN_PROGRAM_QUALIFIERS, 394
 - INSERT_PQLIST, 394
- Warshall's algorithm, 331
 - analysis of, 332
 - example, 332
 - WARSHALL, 332
- Wegbreit's algorithm, 419
- weighting rule for the union operation, 233
- Williams, J.W.J., 198
- worst-fit method, 369