Chapter 7

Trees

7.1 INTRODUCTION

So far, we have been studying mainly linear types of data structures: strings, arrays, lists, stacks and queues. This chapter defines a nonlinear data structure called a *tree*. This structure is mainly used to represent data containing a hierarchical relationship between elements, e.g., records, family trees and tables of contents.

First we investigate a special kind of tree, called a *binary tree*, which can be easily maintained in the computer. Although such a tree may seem to be very restrictive, we will see later in the chapter that more general trees may be viewed as binary trees.

7.2 BINARY TREES

A *binary tree* T is defined as a finite set of elements, called *nodes*, such that:

(*a*) T is empty (called the *null tree* or *empty tree*), or

(*b*) T contains a distinguished node R, called the *root* of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

If T does contain a root R, then the two trees T_1 and T_2 are called, respectively, the *left* and *right subtrees* of R. If T_1 is nonempty, then its root is called the *left successor* of R; similarly, if T_2 is nonempty, then its root is called the *right successor* of R.

A binary tree T is frequently presented by means of a diagram. Specifically, the diagram in Fig. 7-1 represents a binary tree T as follows. (i) T consists of 11 nodes, represented by the letters A through L, excluding I. (ii) The root of T is the node A at the top of the diagram. (iii) A left-downward slanted line from a node N indicates a left successor of N, and a right-downward slanted line from N indicates a right successor of N. Observe that:

(*a*) B is a left successor and C is a right successor of the node A.

(*b*) The left subtree of the root A consists of the nodes B, D, E and F, and the right subtree of A consists of the nodes C, G, H, J, K and L.

Any node N in a binary tree T has either 0, 1 or 2 successors. The nodes A, B, C and H have two successors, the nodes E and J have only one successor, and the nodes D, F, G, L and K have no successors. The nodes with no successors are called *terminal nodes*.

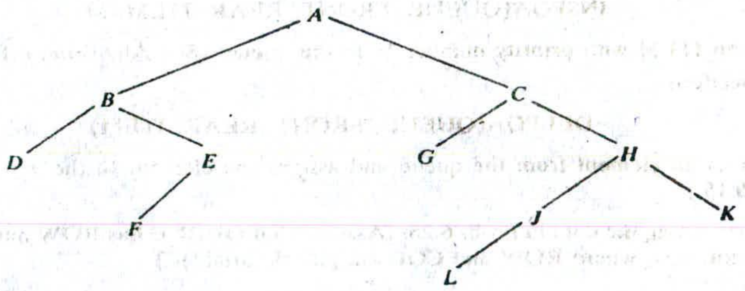


Fig. 7-1

The above definition of the binary tree T is recursive since T is defined in terms of the binary subtrees T_1 and T_2. This means, in particular, that every node N of T contains a left and a right subtree. Moreover, if N is a terminal node, then both its left and right subtrees are empty.

Binary trees T and T' are said to be *similar* if they have the same structure or, in other words, if they have the same shape. The trees are said to be *copies* if they are similar and if they have the same contents at corresponding nodes.

EXAMPLE 7.1

Consider the four binary trees in Fig. 7-2. The three trees (*a*), (*c*) and (*d*) are similar. In particular, the trees (*a*) and (*c*) are copies since they also have the same data at corresponding nodes. The tree (*b*) is neither similar nor a copy of the tree (*d*) because, in a binary tree, we distinguish between a left successor and a right successor even when there is only one successor.

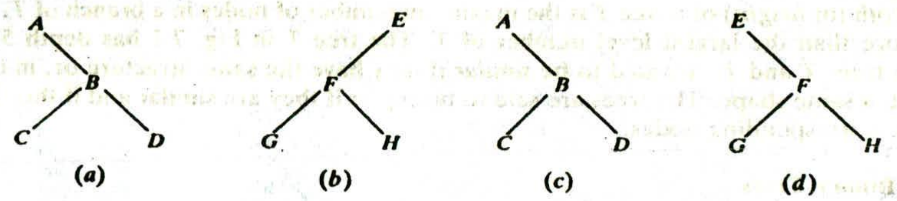


Fig. 7-2

EXAMPLE 7.2 Algebraic Expressions

Consider any algebraic expression E involving only binary operations, such as

$$E = (a - b)/((c * d) + e)$$

E can be represented by means of the binary tree T pictured in Fig. 7-3. That is, each variable or constant in E appears as an "internal" node in T whose left and right subtrees correspond to the operands of the operation. For example:

(*a*) In the expression E, the operands of + are $c * d$ and e.

(*b*) In the tree T, the subtrees of the node + correspond to the subexpressions $c * d$ and e.

Clearly every algebraic expression will correspond to a unique tree, and vice versa.

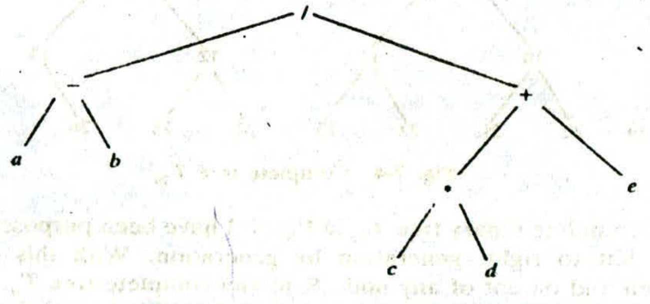


Fig. 7-3 $E = (a - b)/((c * d) + e)$.

Terminology

Terminology describing family relationships is frequently used to describe relationships between the nodes of a tree T. Specifically, suppose N is a node in T with left successor S_1 and right successor S_2. Then N is called the *parent* (or *father*) of S_1 and S_2. Analogously, S_1 is called the *left child* (or *son*) of N, and S_2 is called the *right child* (or *son*) of N. Furthermore, S_1 and S_2 are said to be *siblings* (or *brothers*). Every node N in a binary tree T, except the root, has a unique parent, called the *predecessor* of N.

The terms descendant and ancestor have their usual meaning. That is, a node L is called a *descendant* of a node N (and N is called an *ancestor* of L) if there is a succession of children from N to L. In particular, L is called a *left* or *right descendant* of N according to whether L belongs to the left or right subtree of N.

Terminology from graph theory and horticulture is also used with a binary tree T. Specifically, the line drawn from a node N of T to a successor is called an *edge*, and a sequence of consecutive edges is called a *path*. A terminal node is called a *leaf*, and a path ending in a leaf is called a *branch*.

Each node in a binary tree T is assigned a *level number*, as follows. The root R of the tree T is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent. Furthermore, those nodes with the same level number are said to belong to the same *generation*.

The *depth* (or *height*) of a tree T is the maximum number of nodes in a branch of T. This turns out to be 1 more than the largest level number of T. The tree T in Fig. 7-1 has depth 5.

Binary trees T and T' are said to be *similar* if they have the same structure or, in other words, if they have the same shape. The trees are said to be *copies* if they are similar and if they have the same contents at corresponding nodes.

Complete Binary Trees

Consider any binary tree T. Each node of T can have at most two children. Accordingly, one can show that level r of T can have at most 2^r nodes. The tree T is said to be *complete* if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible. Thus there is a unique complete tree T_n with exactly n nodes (we are, of course, ignoring the contents of the nodes). The complete tree T_{26} with 26 nodes appears in Fig. 7-4.

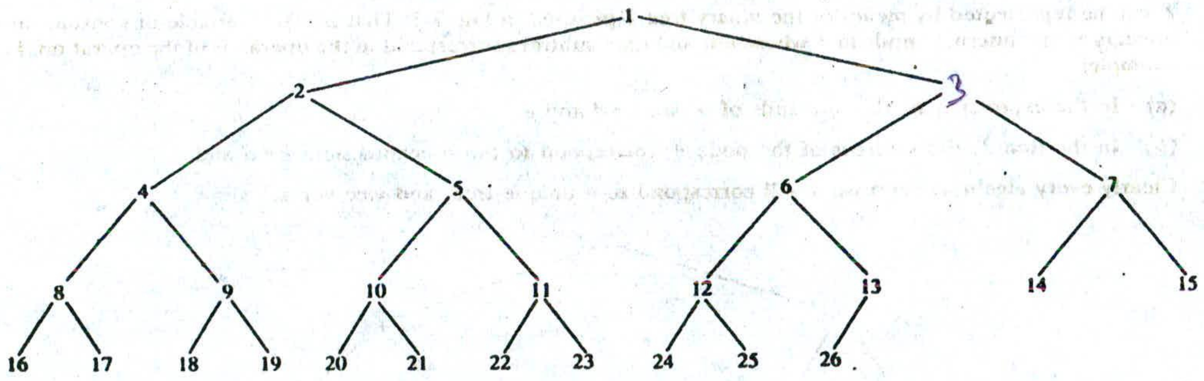


Fig. 7-4 Complete tree T_{26}.

The nodes of the complete binary tree T_{26} in Fig. 7-4 have been purposely labeled by the integers 1, 2, . . . , 26, from left to right, generation by generation. With this labeling, one can easily determine the children and parent of any node K in any complete tree T_n. Specifically, the left and right children of the node K are, respectively, $2 * K$ and $2 * K + 1$, and the parent of K is the node $\lfloor K/2 \rfloor$. For example, the children of node 9 are the nodes 18 and 19, and its parent is the node $\lfloor 9/2 \rfloor = 4$. The depth d_n of the complete tree T_n with n nodes is given by

$$D_n = \lfloor \log_2 n + 1 \rfloor$$

This is a relatively small number. For example, if the complete tree T_n has $n = 1\,000\,000$ nodes, then its depth $D_n = 21$.

Extended Binary Trees: 2-Trees

A binary tree tree *T* is said to be a *2-tree* or an *extended binary tree* if each node *N* has either 0 or 2 children. In such a case, the nodes with 2 children are called *internal nodes*, and the nodes with 0 children are called *external nodes*. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.

The term "extended binary tree" comes from the following operation. Consider any binary tree *T*, such as the tree in Fig. 7-5(*a*). Then *T* may be "converted" into a 2-tree by replacing each empty subtree by a new node, as pictured in Fig. 7-5(*b*). Observe that the new tree is, indeed, a 2-tree. Furthermore, the nodes in the original tree *T* are now the internal nodes in the extended tree, and the new nodes are the external nodes in the extended tree.

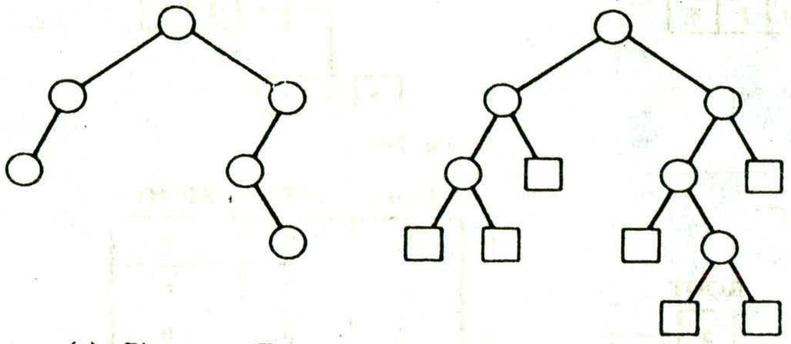

(*a*) Binary tree *T*. (*b*) Extended 2-tree.

Fig. 7-5 Converting a binary tree *T* into a 2-tree.

An important example of a 2-tree is the tree *T* corresponding to any algebraic expression *E* which uses only binary operations. As illustrated in Fig. 7-3, the variables in *E* will appear as the external nodes, and the operations in *E* will appear as internal nodes.

7.3 REPRESENTING BINARY TREES IN MEMORY

Let T be a binary tree. This section discusses two ways of representing T in memory. The first and usual way is called the link representation of T and is analogous to the way linked lists are represented in memory. The second way, which uses a single array, called the sequential representation of T. The main requirement of any representation of T is that one should have direct access to the root R of T and, given any node N of T, one should have direct access to the children of N.

Linked Representation of Binary Trees

Consider a binary tree T. Unless otherwise stated or implied, T will be maintained in memory by means of a *linked representation* which uses three parallel arrays, INFO, LEFT and RIGHT, and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that:

(1) INFO[K] contains the data at the node N.

(2) LEFT[K] contains the location of the left child of node N.

(3) RIGHT[K] contains the location of the right child of node N.

Furthermore, ROOT will contain the location of the root R of T. If any subtree is empty, then the corresponding pointer will contain the null value; if the tree T itself is empty, then ROOT will contain the null value.

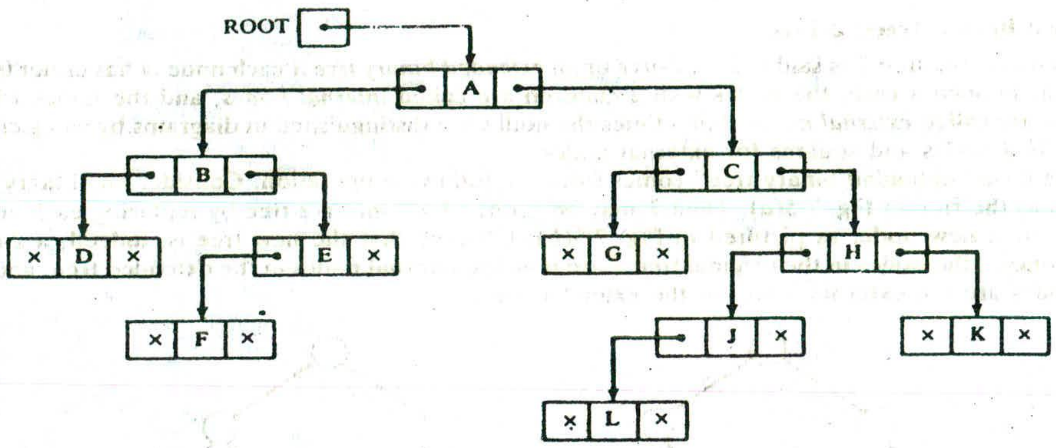
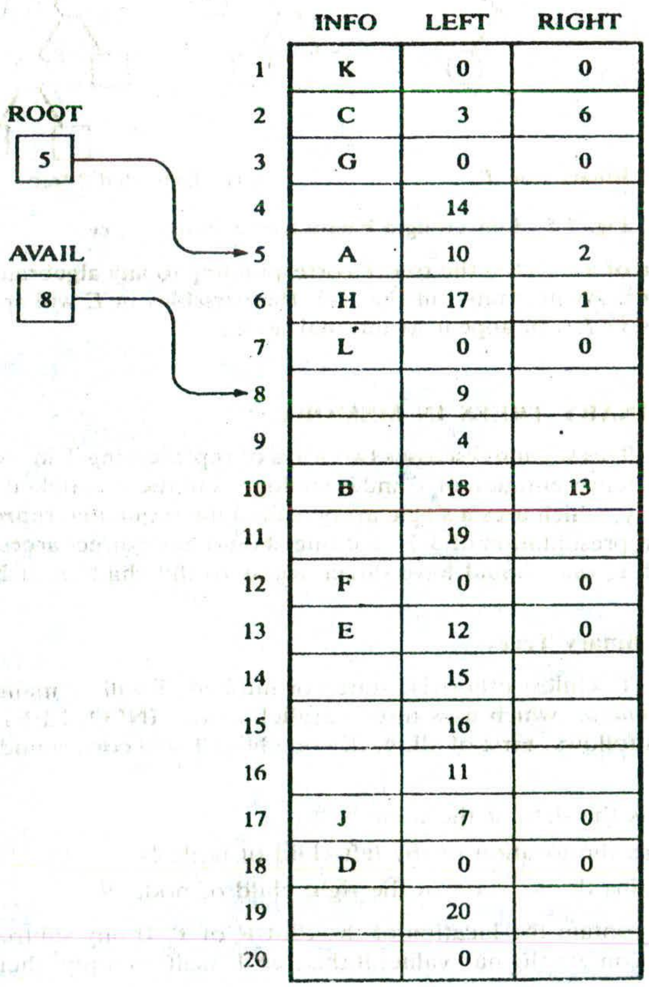


Fig. 7-6



	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

Fig. 7-7

Remark 1: Most of our examples will show a single item of information at each node N of a binary tree T. In actual practice, an entire record may be stored at the node N. In other words, INFO may actually be a linear array of records or a collection of parallel arrays.

Remark 2: Since nodes may be inserted into and deleted from our binary trees, we also implicitly assume that the empty locations in the arrays INFO, LEFT and RIGHT form a linked list with pointer AVAIL, as discussed in relation to linked lists in Chap. 5. We will usually let the LEFT array contain the pointers for the AVAIL list.

Remark 3: Any invalid address may be chosen for the null pointer denoted by NULL. In actual practice, 0 or a negative number is used for NULL. (See Sec. 5.2.)

EXAMPLE 7.3

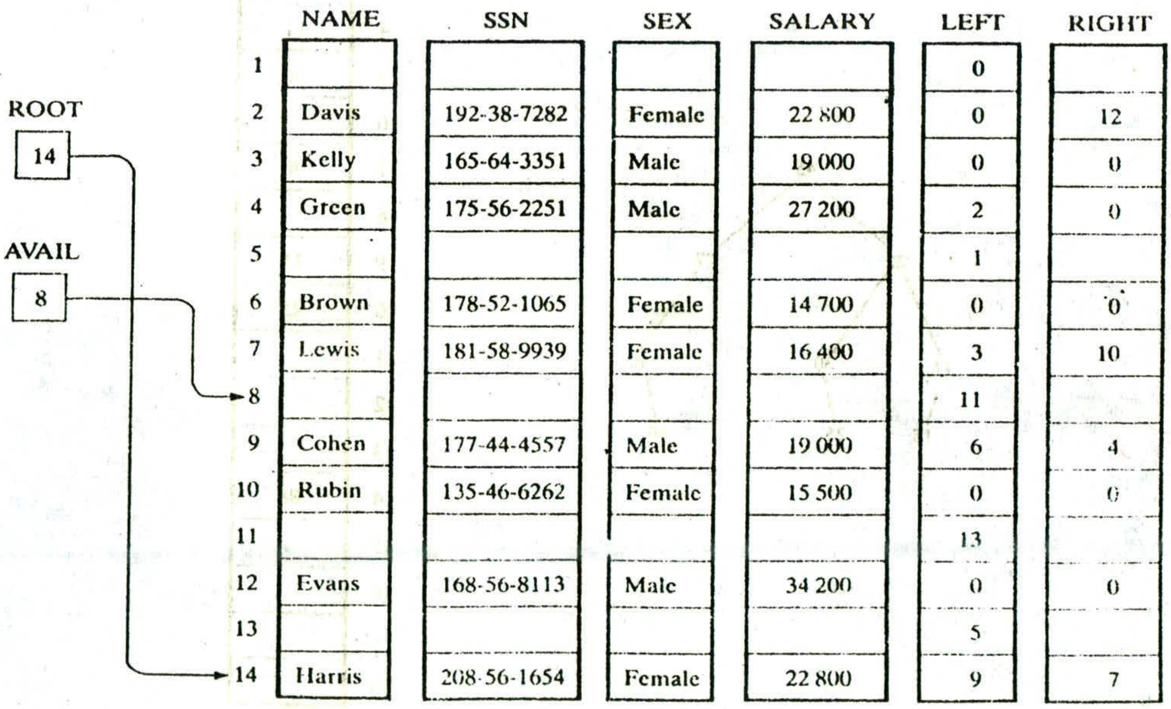
Consider the binary tree *T* in Fig. 7-1. A schematic diagram of the linked representation of *T* appears in Fig. 7-6. Observe that each node is pictured with its three fields, and that the empty subtrees are pictured by using × for the null entries. Figure 7-7 shows how this linked representation may appear in memory. The choice of 20 elements for the arrays is arbitrary. Observe that the AVAIL list is maintained as a one-way list using the array LEFT.

EXAMPLE 7.4

Suppose the personnel file of a small company contains the following data on its nine employees:

Name, Social Security Number, Sex, Monthly Salary

Figure 7-8 shows how the file may be maintained in memory as a binary tree. Compare this data structure with Fig. 5-12, where the exact same data are organized as a one-way list.



	NAME	SSN	SEX	SALARY	LEFT	RIGHT
1					0	
2	Davis	192-38-7282	Female	22 800	0	12
3	Kelly	165-64-3351	Male	19 000	0	0
4	Green	175-56-2251	Male	27 200	2	0
5					1	
6	Brown	178-52-1065	Female	14 700	0	0
7	Lewis	181-58-9939	Female	16 400	3	10
8					11	
9	Cohen	177-44-4557	Male	19 000	6	4
10	Rubin	135-46-6262	Female	15 500	0	0
11					13	
12	Evans	168-56-8113	Male	34 200	0	0
13					5	
14	Harris	208-56-1654	Female	22 800	9	7

Fig. 7-8

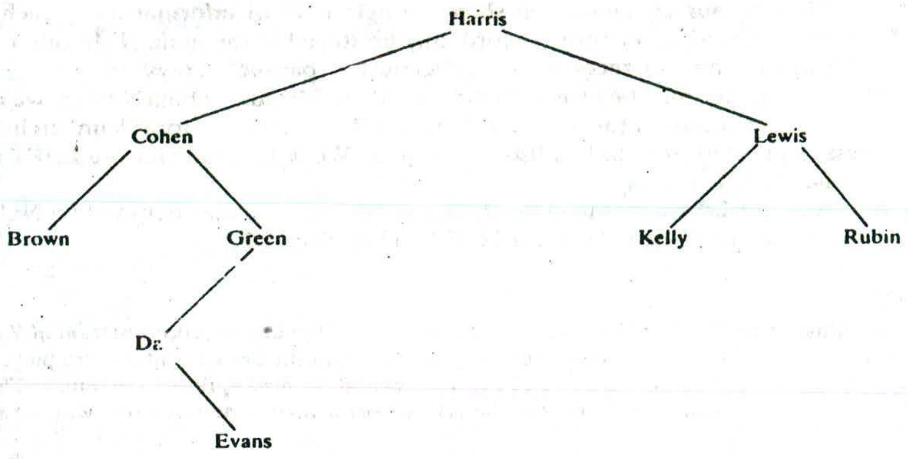
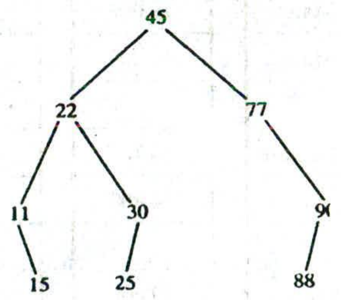


Fig. 7-9



(*a*)

TREE

1	45
2	22
3	77
4	11
5	30
6	
7	90
8	
9	15
10	25
11	
12	
13	
14	88
15	
16	
⋮	
29	

(*b*)

Fig. 7-10

Suppose we want to draw the tree diagram which corresponds to the binary tree in Fig. 7-8. For notational convenience, we label the nodes in the tree diagram only by the key values NAME. We construct the tree as follows:

(*a*) The value ROOT = 14 indicates that Harris is the root of the tree.

(*b*) LEFT[14] = 9 indicates that Cohen is the left child of Harris, and RIGHT[14] = 7 indicates that Lewis is the right child of Harris.

Repeating Step (*b*) for each new node in the diagram, we obtain Fig. 7-9.

Sequential Representation of Binary Trees

Suppose T is a binary tree that is complete or nearly complete. Then there is an efficient way of maintaining T in memory called the *sequential representation* of T. This representation uses only a single linear array TREE as follows:

(*a*) The root R of T is stored in TREE[1].

(*b*) If a node N occupies TREE[K], then its left child is stored in TREE[2 * K] and its right child is stored in TREE[2 * K + 1].

Again, NULL is used to indicate an empty subtree. In particular, TREE[1] = NULL indicates that the tree is empty.

The sequential representation of the binary tree T in Fig. 7-10(*a*) appears in Fig. 7-10(*b*). Observe that we require 14 locations in the array TREE even though T has only 9 nodes. In fact, if we included null entries for the successors of the terminal nodes, then we would actually require TREE[29] for the right successor of TREE[14]. Generally speaking, the sequential representation of a tree with depth *d* will require an array with approximately 2^{d+1} elements. Accordingly, this sequential representation is usually inefficient unless, as stated above, the binary tree T is complete or nearly complete. For example, the tree *T* in Fig. 7-1 has 11 nodes and depth 5, which means it would require an array with approximately $2^6 = 64$ elements.

7.4 TRAVERSING BINARY TREES

There are three standard ways of traversing a binary tree T with root R. These three algorithms, called preorder, inorder and postorder, are as follows:

Preorder: (1) Process the root R.
(2) Traverse the left subtree of R in preorder.
(3) Traverse the right subtree of R in preorder.

Inorder: (1) Traverse the left subtree of R in inorder.
(2) Process the root R.
(3) Traverse the right subtree of R in inorder.

Postorder: (1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

Observe that each algorithm contains the same three steps, and that the left subtree of R is always traversed before the right subtree. The difference between the algorithms is the time at which the root R is processed. Specifically, in the "pre" algorithm, the root R is processed before the subtrees are traversed; in the "in" algorithm, the root R is processed between the traversals of the subtrees; and in the "post" algorithm, the root R is processed after the subtrees are traversed.

The three algorithms are sometimes called, respectively, the node-left-right (NLR) traversal, the left-node-right (LNR) traversal and the left-right-node (LRN) traversal.

Observe that each of the above traversal algorithms is recursively defined, since the algorithm involves traversing subtrees in the given order. Accordingly, we will expect that a stack will be used when the algorithms are implemented on the computer.

EXAMPLE 7.5

Consider the binary tree T in Fig. 7-11. Observe that A is the root, that its left subtree L_T consists of nodes B, D and E and that its right subtree R_T consists of nodes C and F.

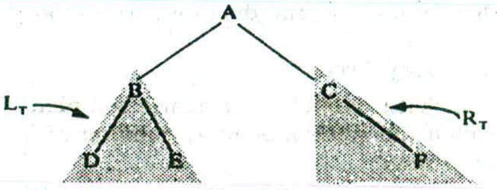


Fig. 7-11

(*a*) The preorder traversal of T processes A, traverses L_T and traverses R_T. However, the preorder traversal of L_T processes the root B and then D and E, and the preorder traversal of R_T processes the root C and then F. Hence ABDECF is the preorder traversal of T.

(*b*) The inorder traversal of T traverses L_T, processes A and traverses R_T. However, the inorder traversal of L_T processes D, B and then E, and the inorder traversal of R_T processes C and then F. Hence DBEACF is the inorder traversal of T.

(*c*) The postorder traversal of T traverses L_T, traverses R_T, and processes A. However, the postorder traversal of L_T processes D, E and then B, and the postorder traversal of R_T processes F and then C. Accordingly, DEBFCA is the postorder traversal of T.

EXAMPLE 7.6

Consider the tree T in Fig. 7-12. The preorder traversal of T is ABDEFCGHJLK. This order is the same as the one obtained by scanning the tree from the left as indicated by the path in Fig. 7-12. That is, one "travels" down the left-most branch until meeting a terminal node, then one backtracks to the next branch, and so on. In the preorder traversal, the right-most terminal node, node K, is the last node scanned. Observe that the left subtree of the root A is traversed before the right subtree, and both are traversed after A. The same is true for any other node having subtrees, which is the underlying property of a preorder traversal.

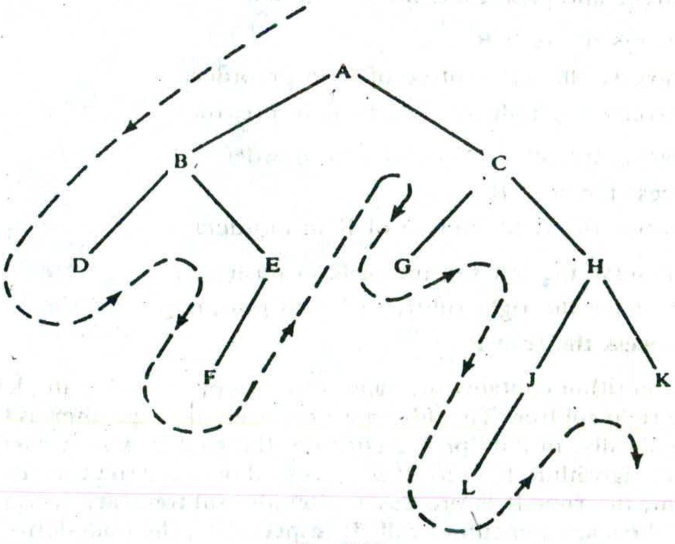


Fig. 7-12

The reader can verify by inspection that the other two ways of traversing the binary tree in Fig. 7-12 are as follows:

(Inorder) D B F E A G C L J H K
(Postorder) D F E B G L J K H C A

Observe that the terminal nodes, D, F, G, L and K, are traversed in the same order, from left to right, in all three traversals. We emphasize that this is true for any binary tree T.

EXAMPLE 7.7

Let E denote the following algebraic expression:

$$[a + (b - c)] * [(d - e)/(f + g - h)]$$

The corresponding binary tree T appears in Fig. 7-13. The reader can verify by inspection that the preorder and postorder traversals of T are as follows:

(Preorder) $* \quad + \quad a \quad - \quad b \quad c \quad / \quad - \quad d \quad e \quad - \quad + \quad f \quad g \quad h$
(Postorder) $a \quad b \quad c \quad - \quad + \quad d \quad e \quad - \quad f \quad g \quad + \quad h \quad - \quad / \quad *$

The reader can also verify that these orders correspond precisely to the prefix and postfix Polish notation of E as discussed in Sec. 6.4. We emphasize that this is true for any algebraic expression E.

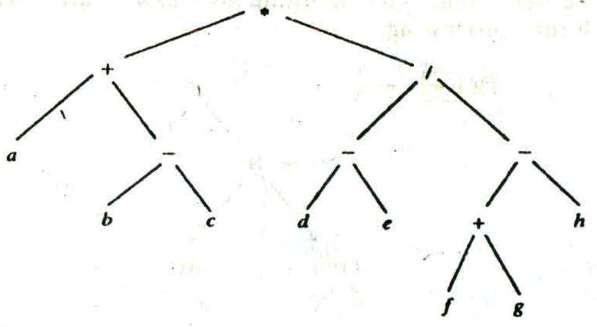


Fig. 7-13

EXAMPLE 7.8

Consider the binary tree T in Fig. 7-14. The reader can verify that the postorder traversal of T is as follows:

S_3, S_6, S_4, S_1, S_7, S_8, S_5, S_2, M

One main property of this traversal algorithm is that every descendant of any node N is processed before the node N. For example, S_6 comes before S_4, S_6 and S_4 come before S_1. Similarly, S_7 and S_8 come before S_5, and S_7, S_8 and S_5 come before S_2. Moreover, all the nodes S_1, S_2, . . . , S_8 come before the root M.

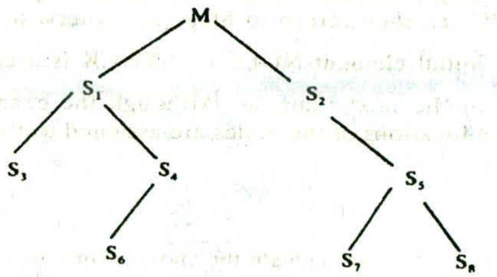


Fig. 7-14

Remark: The reader may be able to implement by inspection the three different traversals of a binary tree T if the tree has a relatively small number of nodes, as in the above two examples. Implementation by inspection may not be possible when T contains hundreds or thousands of nodes. That is, we need some systematic way of implementing the recursively defined traversals. The stack is the natural structure for such an implementation. The discussion of stack-oriented algorithms for this purpose is covered in the next section.

7.5 TRAVERSAL ALGORITHMS USING STACKS

Suppose a binary tree T is maintained in memory by some linked representation

TREE(INFO, LEFT, RIGHT, ROOT)

This section discusses the implementation of the three standard traversals of T, which were defined recursively in the last section, by means of nonrecursive procedures using stacks. We discuss the three traversals separately.

Preorder Traversal

The preorder traversal algorithm uses a variable PTR (pointer) which will contain the location of the node N currently being scanned. This is pictured in Fig. 7-15, where L(N) denotes the left child of node N and R(N) denotes the right child. The algorithm also uses an array STACK, which will hold the addresses of nodes for future processing.

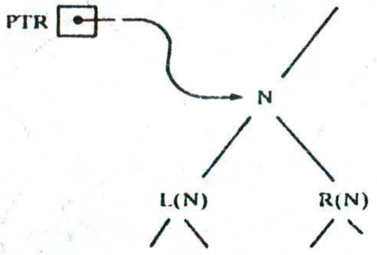


Fig. 7-15

Algorithm: Initially push NULL onto STACK and then set PTR := ROOT. Then repeat the following steps until PTR = NULL or, equivalently, while PTR ≠ NULL.

(*a*) Proceed down the left-most path rooted at PTR, processing each node N on the path and pushing each right child R(N), if any, onto STACK. The traversing ends after a node N with no left child L(N) is processed. (Thus PTR is updated using the assignment PTR := LEFT[PTR], and the traversing stops when LEFT[PTR] = NULL.)

(*b*) [Backtracking.] Pop and assign to PTR the top element on STACK. If PTR ≠ NULL, then return to Step (*a*); otherwise Exit.

(We note that the initial element NULL on STACK is used as a sentinel.)

We simulate the algorithm in the next example. Although the example works with the nodes themselves, in actual practice the locations of the nodes are assigned to PTR and are pushed onto the STACK.

EXAMPLE 7.9

Consider the binary tree T in Fig. 7-16. We simulate the above algorithm with T, showing the contents of STACK at each step.

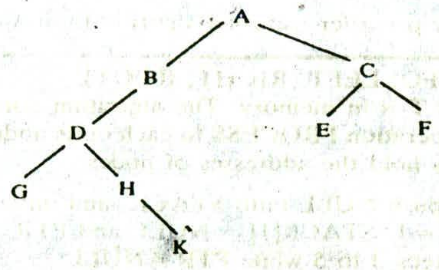


Fig. 7-16

1. Initially push NULL onto STACK:
   STACK: Ø.
   Then set PTR := A, the root of T.
2. Proceed down the left-most path rooted at PTR = A as follows:
   (i) Process A and push its right child C onto STACK:
   STACK: Ø, C.
   (ii) Process B. (There is no right child.)
   (iii) Process D and push its right child H onto STACK:
   STACK: Ø, C, H.
   (iv) Process G. (There is no right child.)
   No other node is processed, since G has no left child.
3. [Backtracking.] Pop the top element H from STACK, and set PTR := H. This leaves:
   STACK: Ø, C.
   Since PTR ≠ NULL, return to Step (*a*) of the algorithm.
4. Proceed down the left-most path rooted at PTR = H as follows:
   (v) Process H and push its right child K onto STACK:
   STACK: Ø, C, K.
   No other node is processed, since H has no left child.
5. [Backtracking.] Pop K from STACK, and set PTR := K. This leaves:
   STACK: Ø, C.
   Since PTR ≠ NULL, return to Step (*a*) of the algorithm.
6. Proceed down the left-most path rooted at PTR = K as follows:
   (vi) Process K. (There is no right child.)
   No other node is processed, since K has no left child.
7. [Backtracking.] Pop C from STACK, and set PTR := C. This leaves:
   STACK: Ø.
   Since PTR ≠ NULL, return to Step (*a*) of the algorithm.
8. Proceed down the leftmost path rooted at PTR = C as follows:
   (vii) Process C and push its right child F onto STACK:
   STACK: Ø, F.
   (viii) Process E. (There is no right-child.)
9. [Backtracking.] Pop F from STACK, and set PTR := F. This leaves:
   STACK: Ø.
   Since PTR ≠ NULL, return to Step (*a*) of the algorithm.
10. Proceed down the left-most path rooted at PTR = F as follows:
    (ix) Process F. (There is no right child.)
    No other node is processed, since F has no left child.
11. [Backtracking.] Pop the top element NULL from STACK, and set PTR := NULL. Since PTR = NULL, the algorithm is completed.

As seen from Steps 2, 4, 6, 8 and 10, the nodes are processed in the order A, B, D, G, H, K, C, E, F. This is the required preorder traversal of T.

A formal presentation of our preorder traversal algorithm follows:

```
Algorithm 7.1: PREORD(INFO, LEFT, RIGHT, ROOT)
    A binary tree T is in memory. The algorithm does a preorder traversal of T,
    applying an operation PROCESS to each of its nodes. An array STACK is used
    to temporarily hold the addresses of nodes.

    1.  [Initially push NULL onto STACK, and initialize PTR.]
        Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
    2.  Repeat Steps 3 to 5 while PTR ≠ NULL:
    3.      Apply PROCESS to INFO[PTR].
    4.      [Right child?]
            If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
                Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
            [End of If structure.]
    5.      [Left child?]
            If LEFT[PTR] ≠ NULL, then:
                Set PTR := LEFT[PTR].
            Else: [Pop from STACK.]
                Set PTR := STACK[TOP] and TOP := TOP − 1.
            [End of If structure.]
        [End of Step 2 loop.]
    6.  Exit.
```

Inorder Traversal

The inorder traversal algorithm also uses a variable pointer PTR, which will contain the location of the node N currently being scanned, and an array STACK, which will hold the addresses of nodes for future processing. In fact, with this algorithm, a node is processed only when it is popped from STACK.

Algorithm: Initially push NULL onto STACK (for a sentinel) and then set PTR := ROOT. Then repeat the following steps until NULL is popped from STACK.

(*a*) Proceed down the left-most path rooted at PTR, pushing each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.

(*b*) [Backtracking.] Pop and process the nodes on STACK. If NULL is popped, then Exit. If a node N with a right child R(N) is processed, set PTR = R(N) (by assigning PTR := RIGHT[PTR]) and return to Step (*a*).

We emphasize that a node N is processed only when it is popped from STACK.

EXAMPLE 7.10

Consider the binary tree T in Fig. 7-17. We simulate the above algorithm with T, showing the contents of STACK.

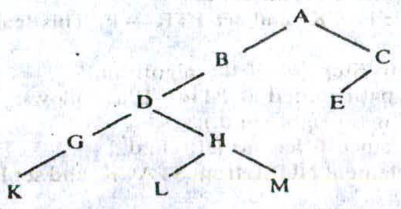


Fig. 7-17

1. Initially push NULL onto STACK:
   STACK: ∅.
   Then set PTR := A, the root of T.
2. Proceed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G and K onto STACK:
   STACK: ∅, A, B, D, G, K.
   (No other node is pushed onto STACK, since K has no left child.)
3. [Backtracking.] The nodes K, G and D are popped and processed, leaving:
   STACK: ∅, A, B.
   (We stop the processing at D, since D has a right child.) Then set PTR := H, the right child of D.
4. Proceed down the left-most path rooted at PTR = H, pushing the nodes H and L onto STACK:
   STACK: ∅, A, B, H, L.
   (No other node is pushed onto STACK, since L has no left child.)
5. [Backtracking.] The nodes L and H are popped and processed, leaving:
   STACK: ∅, A, B.
   (We stop the processing at H, since H has a right child.) Then set PTR := M, the right child of H.
6. Proceed down the left-most path rooted at PTR = M, pushing node M onto STACK:
   STACK; ∅, A, B, M.
   (No other node is pushed onto STACK, since M has no left child.)
7. [Backtracking.] The nodes M, B and A are popped and processed, leaving:
   STACK; ∅.
   (No other element of STACK is popped, since A does have a right child.) Set PTR := C, the right child of A.
8. Proceed down the left-most path rooted at PTR = C, pushing the nodes C and E onto STACK:
   STACK: ∅, C, E.
9. [Backtracking.] Node E is popped and processed. Since E has no right child, node C is popped and processed. Since C has no right child, the next element, NULL, is popped from STACK.

The algorithm is now finished, since NULL is popped from STACK. As seen from Steps 3, 5, 7 and 9, the nodes are processed in the order K, G, D, L, H, M, B, A, E, C. This is the required inorder traversal of the binary tree T.

A formal presentation of our inorder traversal algorithm follows:

Algorithm 7.2: INORD(INFO, LEFT, RIGHT, ROOT)
A binary tree is in memory. This algorithm does an inorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK.]
       (a) Set TOP := TOP + 1 and STACK[TOP] := PTR. [Saves node.]
       (b) Set PTR := LEFT[PTR]. [Updates PTR.]
   [End of loop.]
3. Set PTR := STACK[TOP] and TOP := TOP - 1. [Pops node from STACK.]
4. Repeat Steps 5 to 7 while PTR ≠ NULL: [Backtracking.]
5.     Apply PROCESS to INFO[PTR].
6.     [Right child?] If RIGHT[PTR] ≠ NULL, then:
           (a) Set PTR := RIGHT[PTR].
           (b) Go to Step 3.
       [End of If structure.]
7.     Set PTR := STACK[TOP] and TOP := TOP - 1. [Pops node.]
   [End of Step 4 loop.]
8. Exit.
```

Postorder Traversal

The postorder traversal algorithm is more complicated than the preceding two algorithms, because here we may have to save a node N in two different situations. We distinguish between the two cases by pushing either N or its negative, −N, onto STACK. (In actual practice, the location of N is pushed onto STACK, so −N has the obvious meaning.) Again, a variable PTR (pointer) is used which contains the location of the node N that is currently being scanned, as in Fig. 7-15.

Algorithm: Initially push NULL onto STACK (as a sentinel) and then set PTR := ROOT. Then repeat the following steps until NULL is popped from STACK.

(*a*) Proceed down the left-most path rooted at PTR. At each node N of the path, push N onto STACK and, if N has a right child R(N), push −R(N) onto STACK.

(*b*) [Backtracking.] Pop and process positive nodes on STACK. If NULL is popped, then Exit. If a negative node is popped, that is, if PTR = −N for some node N, set PTR = N (by assigning PTR := −PTR) and return to Step (*a*).

We emphasize that a node N is processed only when it is popped from STACK and it is positive.

EXAMPLE 7.11

Consider again the binary tree T in Fig. 7-17. We simulate the above algorithm with T, showing the contents of STACK.

1. Initially, push NULL onto STACK and set PTR := A, the root of T:
   STACK: Ø.
2. Proceed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G and K onto STACK. Furthermore, since A has a right child C, push −C onto STACK after A but before B, and since D has a right child H, push −H onto STACK after D but before G. This yields:
   STACK: Ø, A, −C, B, D, −H, G, K.
3. [Backtracking.] Pop and process K, and pop and process G. Since −H is negative, only pop −H. This leaves:
   STACK: Ø, A, −C, B, D.
   Now PTR = −H. Reset PTR = H and return to Step (*a*).
4. Proceed down the left-most path rooted at PTR = H. First push H onto STACK. Since H has a right child M, push −M onto STACK after H. Last, push L onto STACK. This gives:
   STACK: Ø, A, −C, B, D, H, −M, L.
5. [Backtracking.] Pop and process L, but only pop −M . This leaves:
   STACK: Ø, A, −C, B, D, H.
   Now PTR = −M. Reset PTR = M and return to Step (*a*).
6. Proceed down the left-most path rooted at PTR = M. Now, only M is pushed onto STACK. This yields:
   STACK: Ø, A, −C, B, D, H, M.
7. [Backtracking.] Pop and process M, H, D and B, but only pop −C. This leaves:
   STACK: Ø, A.
   Now PTR = −C. Reset PTR = C, and return to Step (*a*).
8. Proceed down the left-most path rooted at PTR = C. First C is pushed onto STACK and then E, yielding:
   STACK: Ø, A, C, E.
9. [Backtracking.] Pop and process E, C and A. When NULL is popped, STACK is empty and the algorithm is completed.

As seen from Steps 3, 5, 7 and 9, the nodes are processed in the order K, G, L, M, H, D, B E, C, A. This is the required postorder traversal of the binary tree T.

A formal presentation of our postorder traversal algorithm follows:

Algorithm 7.3: POSTORD(INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1.  [Push NULL onto STACK and initialize PTR.]
    Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2.  [Push left-most path onto STACK.]
    Repeat Steps 3 to 5 while PTR ≠ NULL:
3.      Set TOP := TOP + 1 and STACK[TOP] := PTR.
        [Pushes PTR on STACK.]
4.      If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
            Set TOP := TOP + 1 and STACK[TOP] := -RIGHT[PTR].
        [End of If structure.]
5.      Set PTR := LEFT[PTR]. [Updates pointer PTR.]
    [End of Step 2 loop.]
6.  Set PTR := STACK[TOP] and TOP := TOP - 1.
    [Pops node from STACK.]
7.  Repeat while PTR > 0:
        (a) Apply PROCESS to INFO[PTR].
        (b) Set PTR := STACK[TOP] and TOP := TOP - 1.
            [Pops node from STACK.]
    [End of loop.]
8.  If PTR < 0, then:
        (a) Set PTR := -PTR.
        (b) Go to Step 2.
    [End of If structure.]
9.  Exit.
```

7.6 HEADER NODES; THREADS

Consider a binary tree T. Variations of the linked representation of T are frequently used because certain operations on T are easier to implement by using the modifications. Some of these variations, which are analogous to header and circular linked lists, are discussed in this section.

Header Nodes

Suppose a binary tree T is maintained in memory by means of a linked representation. Sometimes an extra, special node, called a *header node*, is added to the beginning of T. When this extra node is used, the tree pointer variable, which we will call HEAD (instead of ROOT), will point to the header node, and the left pointer of the header node will point to the root of T. Figure 7-18 shows a schematic picture of the binary tree in Fig. 7-1 that uses a linked representation with a header node. (Compare with Fig. 7-6.)

Suppose a binary tree T is empty. Then T will still contain a header node, but the left pointer of the header node will contain the null value. Thus the condition

$$\text{LEFT[HEAD]} = \text{NULL}$$

will indicate an empty tree.

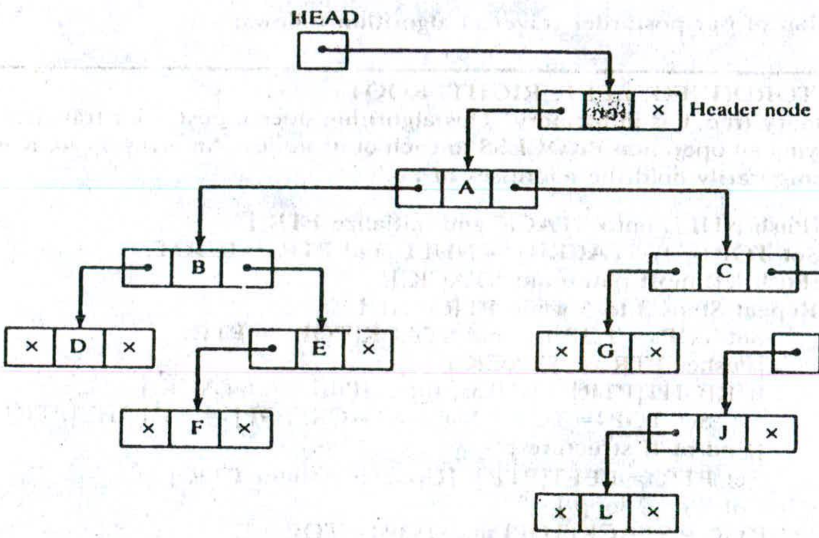


Fig. 7-18

Another variation of the above representation of a binary tree T is to use the header node as a sentinel. That is, if a node has an empty subtree, then the pointer field for the subtree will contain the address of the header node instead of the null value. Accordingly, no pointer will ever contain an invalid address, and the condition

$$\mathrm{LEFT[HEAD]} = \mathrm{HEAD}$$

will indicate an empty subtree.

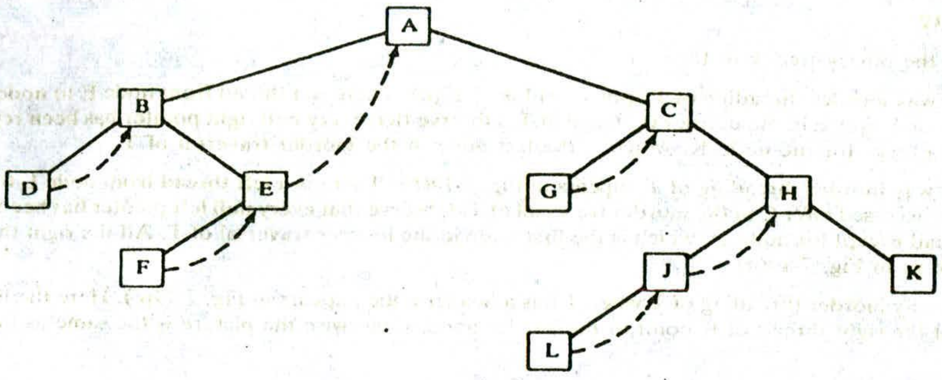
Threads; Inorder Threading

Consider again the linked representation of a binary tree T. Approximately half of the entries in the pointer fields LEFT and RIGHT will contain null elements. This space may be more efficiently used by replacing the null entries by some other type of information. Specifically, we will replace certain null entries by special pointers which point to nodes higher in the tree. These special pointers are called *threads*, and binary trees with such pointers are called *threaded trees*.

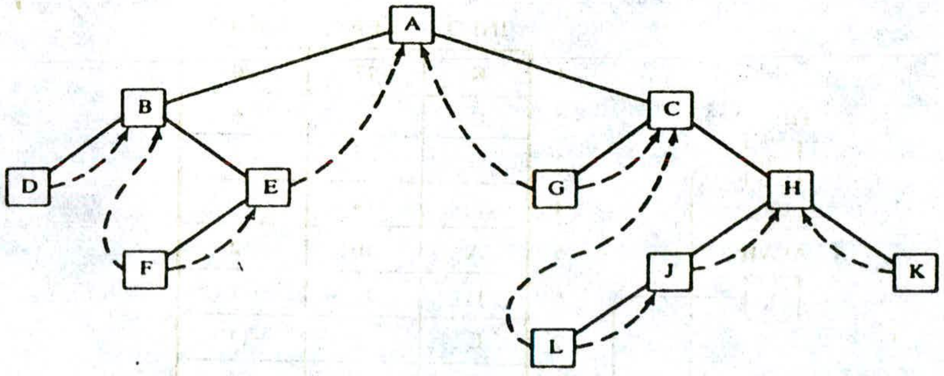
The threads in a threaded tree must be distinguished in some way from ordinary pointers. The threads in a diagram of a threaded tree are usually indicated by dotted lines. In computer memory, an extra 1-bit TAG field may be used to distinguish threads from ordinary pointers, or, alternatively, threads may be denoted by negative integers when ordinary pointers are denoted by positive integers.

There are many ways to thread a binary tree T, but each threading will correspond to a particular traversal of T. Also, one may choose a one-way threading or a two-way threading. Unless otherwise stated, our threading will correspond to the inorder traversal of T. Accordingly, in the one-way threading of T, a thread will appear in the right field of a node and will point to the next node in the inorder traversal of T; and in the two-way threading of T, a thread will also appear in the LEFT field of a node and will point to the preceding node in the inorder traversal of T. Furthermore, the left pointer of the first node and the right pointer of the last node (in the inorder traversal of T) will contain the null value when T does not have a header node, but will point to the header node when T does have a header node.

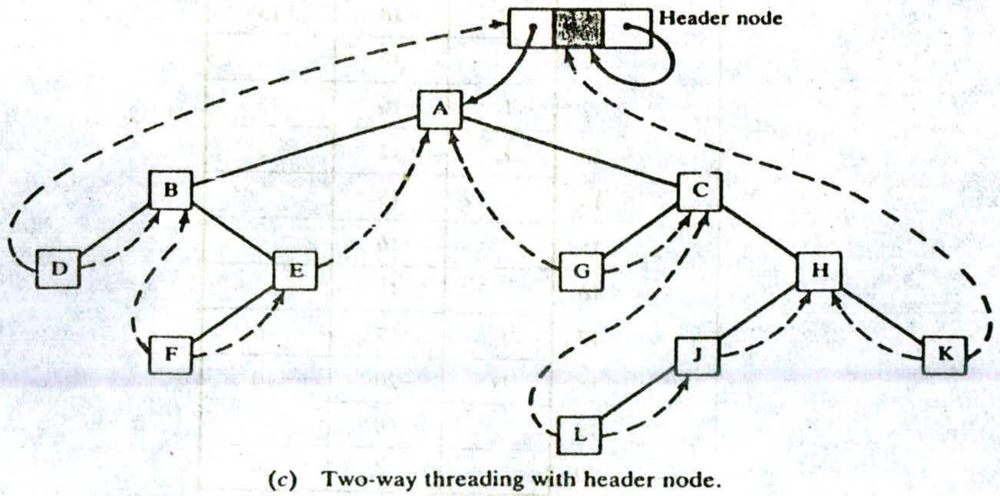
There is an analogous one-way threading of a binary tree T which corresponds to the preorder traversal of T. (See Prob. 7.13.) On the other hand, there is no threading of T which corresponds to the postorder traversal of T.



(*a*) One-way inorder threading.



(*b*) Two-way inorder threading.



(*c*) Two-way threading with header node.

Fig. 7-19

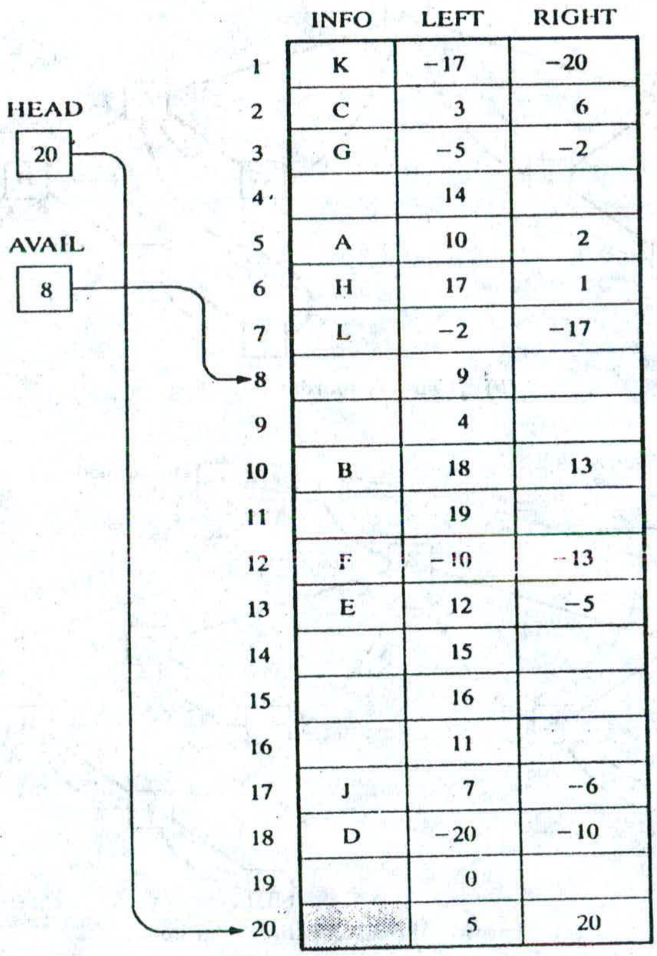
EXAMPLE 7.12

Consider the binary tree T in Fig. 7-1.

(*a*) The one-way inorder threading of T appears in Fig. 7-19(*a*). There is a thread from node E to node A, since A is accessed after E in the inorder traversal of T. Observe that every null right pointer has been replaced by a thread except for the node K, which is the last node in the inorder traversal of T.

(*b*) The two-way inorder threading of T appears in Fig. 7-19(*b*). There is a left thread from node L to node C, since L is accessed after C in the inorder traversal of T. Observe that every null left pointer has been replaced by a thread except for node D, which is the first node in the inorder traversal of T. All the right threads are the same as in Fig. 7-19(*a*).

(*c*) The two-way inorder threading of T when T has a header node appears in Fig. 7-19(*c*). Here the left thread of D and the right thread of K point to the header node. Otherwise the picture is the same as that in Fig. 7-19(*b*).



	INFO	LEFT	RIGHT
1	K	−17	−20
2	C	3	6
3	G	−5	−2
4		14	
5	A	10	2
6	H	17	1
7	L	−2	−17
8		9	
9		4	
10	B	18	13
11		19	
12	F	−10	−13
13	E	12	−5
14		15	
15		16	
16		11	
17	J	7	−6
18	D	−20	−10
19		0	
20		5	20

Fig. 7-20

(*d*) Figure 7-7 shows how T may be maintained in memory by using a linked representation. Figure 7-20 shows how the representation should be modified so that T is a two-way inorder threaded tree using INFO[20] as a header node. Observe that LEFT[12] = −10, which means there is a left thread from node F to node B. Analogously, RIGHT[17] = −6 means there is a right thread from node J to node H. Last, observe that RIGHT[20] = 20, which means there is an ordinary right pointer from the header node to itself. If T were empty, then we would set LEFT[20] = −20, which would mean there is a left thread from the header node to itself.

7.7 BINARY SEARCH TREES

This section discusses one of the most important data structures in computer science, a binary search tree. This structure enables one to search for and find an element with an average running time $f(n) = O(\log_2 n)$. It also enables one to easily insert and delete elements. This structure contrasts with the following structures:

(*a*) *Sorted linear array*. Here one can search for and find an element with a running time $f(n) = O(\log_2 n)$, but it is expensive to insert and delete elements.

(*b*) *Linked list*. Here one can easily insert and delete elements, but it is expensive to search for and find an element, since one must use a linear search with running time $f(n) = O(n)$.

Although each node in a binary search tree may contain an entire record of data, the definition of the binary tree depends on a given field whose values are distinct and may be ordered.

Suppose T is a binary tree. Then T is called a *binary search tree* (or *binary sorted tree*) if each node N of T has the following property: *The value at* N *is greater than every value in the left subtree of* N *and is less than every value in the right subtree of* N. (It is not difficult to see that this property guarantees that the inorder traversal of T will yield a sorted listing of the elements of T.)

EXAMPLE 7.13

(*a*) Consider the binary tree T in Fig. 7-21. T is a binary search tree; that is, every node N in T exceeds every number in its left subtree and is less than every number in its right subtree. Suppose the 23 were replaced by 35. Then T would still be a binary search tree. On the other hand, suppose the 23 were replaced by 40. Then T would not be a binary search tree, since the 38 would not be greater than the 40 in its left subtree.

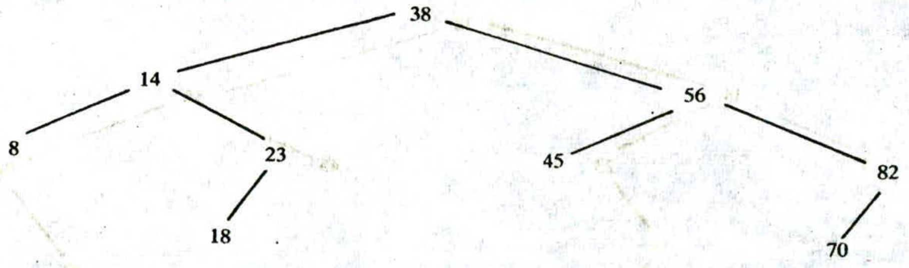


Fig. 7-21

(*b*) Consider the file in Fig. 7-8. As indicated by Fig. 7-9, the file is a binary search tree with respect to the key NAME. On the other hand, the file is not a binary search tree with respect to the social security number key SSN. This situation is similar to an array of records which is sorted with respect to one key but is unsorted with respect to another key.

The definition of a binary search tree given in this section assumes that all the node values are distinct. There is an analogous definition of a binary search tree which admits duplicates, that is, in which each node N has the following property: *The value at* N *is greater than every value in the left subtree of* N *and is less than or equal to every value in the right subtree of* N. When this definition is used, the operations in the next section must be modified accordingly.

7.8 SEARCHING AND INSERTING IN BINARY SEARCH TREES

Suppose T is a binary search tree. This section discusses the basic operations of searching and inserting with respect to T. In fact, the searching and inserting will be given by a single search and insertion algorithm. The operation of deleting is treated in the next section. Traversing in T is the same as traversing in any binary tree; this subject has been covered in Sec. 7.4.

Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(*a*) Compare ITEM with the root node N of the tree.
  (i) If ITEM < N, proceed to the left child of N.
  (ii) If ITEM > N, proceed to the right child of N.

(*b*) Repeat Step (*a*) until one of the following occurs:
  (i) We meet a node N such that ITEM = N. In this case the search is successful.
  (*ii*) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

EXAMPLE 7.14

(*a*) Consider the binary search tree T in Fig. 7-21. Suppose ITEM = 20 is given. Simulating the above algorithm, we obtain the following steps:

1. Compare ITEM = 20 with the root, 38, of the tree T. Since 20 < 38, proceed to the left child of 38, which is 14.
2. Compare ITEM = 20 with 14. Since 20 > 14, proceed to the right child of 14, which is 23.
3. Compare ITEM = 20 with 23. Since 20 < 23, proceed to the left child of 23, which is 18.
4. Compare ITEM = 20 with 18. Since 20 > 18 and 18 does not have a right child, insert 20 as the right child of 18.

Figure 7-22 shows the new tree with ITEM = 20 inserted. The shaded edges indicate the path down through the tree during the algorithm.

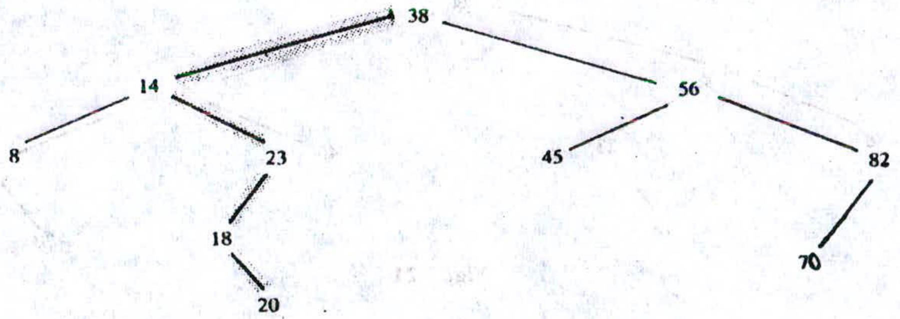


Fig. 7-22 ITEM = 20 inserted.

(*b*) Consider the binary search tree T in Fig. 7-9. Suppose ITEM = Davis is given. Simulating the above algorithm, we obtain the following steps:

1. Compare ITEM = Davis with the root of the tree, Harris. Since Davis < Harris, proceed to the left child of Harris, which is Cohen.
2. Compare ITEM = Davis with Cohen. Since Davis > Cohen, proceed to the right child of Cohen, which is Green.

3. Compare ITEM = Davis with Green. Since Davis < Green, proceed to the left child of Green, which is Davis.
4. Compare ITEM = Davis with the left child, Davis. We have found the location of Davis in the tree.

EXAMPLE 7.15

Suppose the following six numbers are inserted in order into an empty binary search tree:

40, 60, 50, 33, 55, 11

Figure 7-23 shows the six stages of the tree. We emphasize that if the six numbers were given in a different order, then the tree might be different and we might have a different depth.

The formal presentation of our search and insertion algorithm will use the following procedure, which finds the locations of a given ITEM and its parent. The procedure traverses down the tree using the pointer PTR and the pointer SAVE for the parent node. This procedure will also be used in the next section, on deletion.

```
Procedure 7.4: FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
    A binary search tree T is in memory and an ITEM of information is given. This
    procedure finds the location LOC of ITEM in T and also the location PAR of
    the parent of ITEM. There are three special cases:
        (i)   LOC = NULL and PAR = NULL will indicate that the tree is
              empty.
        (ii)  LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root
              of T.
        (iii) LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T
              and can be added to T as a child of the node N with location PAR.
    1. [Tree empty?]
       If ROOT = NULL, then: Set LOC := NULL and PAR := NULL, and
       Return.
    2. [ITEM at root?]
       If ITEM = INFO[ROOT], then: Set LOC := ROOT and PAR := NULL,
       and Return.
    3. [Initialize pointers PTR and SAVE.]
       If ITEM < INFO[ROOT], then:
           Set PTR := LEFT[ROOT] and SAVE := ROOT.
       Else:
           Set PTR := RIGHT[ROOT] and SAVE := ROOT.
       [End of If structure.]
    4. Repeat Steps 5 and 6 while PTR ≠ NULL:
    5.     [ITEM found?]
           If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE,
           and Return.
    6.     If ITEM < INFO[PTR], then:
               Set SAVE := PTR and PTR := LEFT[PTR].
           Else:
               Set SAVE := PTR and PTR := RIGHT[PTR].
           [End of If structure.]
       [End of Step 4 loop.]
    7. [Search unsuccessful.] Set LOC := NULL and PAR := SAVE.
    8. Exit.
```

Observe that, in Step 6, we move to the left child or the right child according to whether ITEM < INFO[PTR] or ITEM > INFO[PTR].

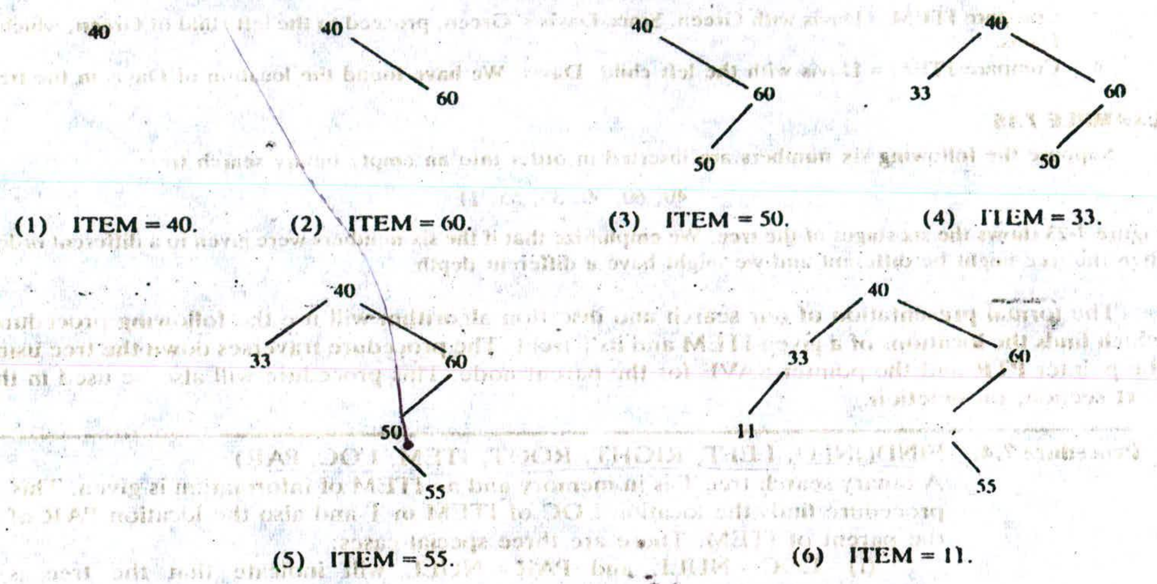


Fig. 7-23

The formal statement of our search and insertion algorithm follows.

```
Algorithm 7.5: INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)
    A binary search tree T is in memory and an ITEM of information is given. This
    algorithm finds the location LOC of ITEM in T or adds ITEM as a new node in T
    at location LOC.

    1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
       [Procedure 7.4.]
    2. If LOC ≠ NULL, then Exit.
    3. [Copy ITEM into new node in AVAIL list.]
       (a) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
       (b) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and
           INFO[NEW] := ITEM.
       (c) Set LOC := NEW, LEFT[NEW] := NULL and
           RIGHT[NEW] := NULL.
    4. [Add ITEM to tree.]
       If PAR = NULL, then:
           Set ROOT := NEW.
       Else if ITEM < INFO[PAR], then:
           Set LEFT[PAR] := NEW.
       Else:
           Set RIGHT[PAR] := NEW.
       [End of If structure.]
    5. Exit.
```

Observe that, in Step 4, there are three possibilities: (1) the tree is empty, (2) ITEM is added as a left child and (3) ITEM is added as a right child.

Complexity of the Searching Algorithm

Suppose we are searching for an item of information in a binary search tree T. Observe that the number of comparisons is bounded by the depth of the tree. This comes from the fact that we proceed down a single path of the tree. Accordingly, the running time of the search will be proportional to the depth of the tree.

Suppose we are given n data items, $A_1, A_2, \ldots, A_N$, and suppose the items are inserted in order into a binary search tree T. Recall that there are $n!$ permutations of the n items (Sec. 2.2). Each such permutation will give rise to a corresponding tree. It can be shown that the average depth of the $n!$ trees is approximately $c \log_2 n$, where $c = 1.4$. Accordingly, the average running time $f(n)$ to search for an item in a binary tree T with n elements is proportional to $\log_2 n$, that is, $f(n) = O(\log_2 n)$.

Application of Binary Search Trees

Consider a collection of n data items, $A_1, A_2, \ldots, A_N$. Suppose we want to find and delete all duplicates in the collection. One straightforward way to do this is as follows:

Algorithm A: Scan the elements from A_1 to A_N (that is, from left to right).

(*a*) For each element A_K, compare A_K with $A_1, A_2, \ldots, A_{K-1}$, that is, compare A_K with those elements which precede A_K.

(*b*) If A_K does occur among $A_1, A_2, \ldots, A_{K-1}$, then delete A_K.

After all elements have been scanned, there will be no duplicates.

EXAMPLE 7.16

Suppose Algorithm A is applied to the following list of 15 numbers:

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

Observe that the first four numbers (14, 10, 17 and 12) are not deleted. However,

$A_5 = 10$	is deleted, since	$A_5 = A_2$
$A_8 = 12$	is deleted, since	$A_8 = A_4$
$A_{11} = 20$	is deleted, since	$A_{11} = A_7$
$A_{14} = 11$	is deleted, since	$A_{14} = A_6$

When Algorithm A is finished running, the 11 numbers

14, 10, 17, 12, 11, 20, 18, 25, 8, 22, 23

which are all distinct, will remain.

Consider now the time complexity of Algorithm A, which is determined by the number of comparisons. First of all, we assume that the number d of duplicates is very small compared with the number n of data items. Observe that the step involving A_K will require approximately $k - 1$ comparisons, since we compare A_K with items $A_1, A_2, \ldots, A_{K-1}$ (less the few that may already have been deleted). Accordingly, the number $f(n)$ of comparisons required by Algorithm A is approximately

$$0 + 1 + 2 + 3 + \cdots + (n-2) + (n-1) = \frac{(n-1)n}{2} = O(n^2)$$

For example, for $n = 1000$ items, Algorithm A will require approximately 500 000 comparisons. In other words, the running time of Algorithm A is proportional to n^2.

Using a binary search tree, we can give another algorithm to find the duplicates in the set $A_1, A_2, \ldots, A_N$ of n data items.

Algorithm B: Build a binary search tree T using the elements $A_1, A_2, \ldots, A_N$. In building the tree, delete A_K from the list whenever the value of A_K already appears in the tree.

The main advantage of Algorithm B is that each element A_K is compared only with the elements in a single branch of the tree. It can be shown that the average length of such a branch is approximately $c \log_2 k$, where $c = 1.4$. Accordingly, the total number $f(n)$ of comparisons required by Algorithm B is approximately $n \log_2 n$, that is, $f(n) = O(n \log_2 n)$. For example, for $n = 1000$, Algorithm B will require approximately 10 000 comparisons rather than the 500 000 comparisons with Algorithm A. (We note that, for the worst case, the number of comparisons for Algorithm B is the same as for Algorithm A.)

EXAMPLE 7.17

Consider again the following list of 15 numbers:

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

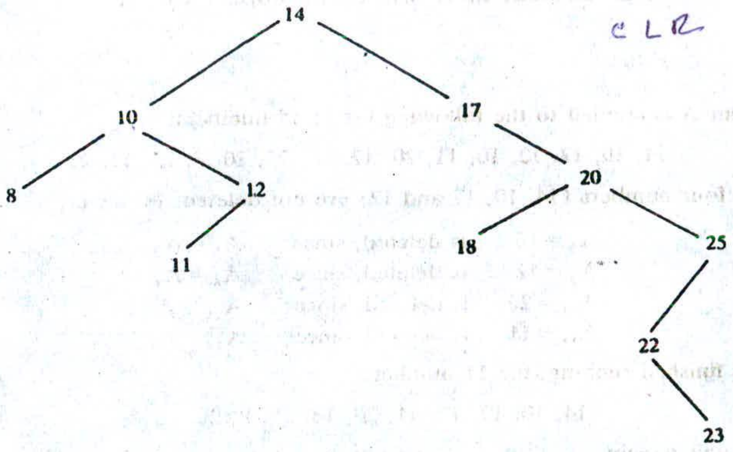
Applying Algorithm B to this list of numbers, we obtain the tree in Fig. 7-24. The exact number of comparisons is

$$0 + 1 + 1 + 2 + 2 + 3 + 2 + 3 + 3 + 3 + 3 + 2 + 4 + 4 + 5 = 38$$

On the other hand, Algorithm A requires

$$0 + 1 + 2 + 3 + 2 + 4 + 5 + 4 + 6 + 7 + 6 + 8 + 9 + 5 + 10 = 72$$

comparisons.



C L R

Fig. 7-24

7.9 DELETING IN A BINARY SEARCH TREE

Suppose T is a binary search tree, and suppose an ITEM of information is given. This section gives an algorithm which deletes ITEM from the tree T.

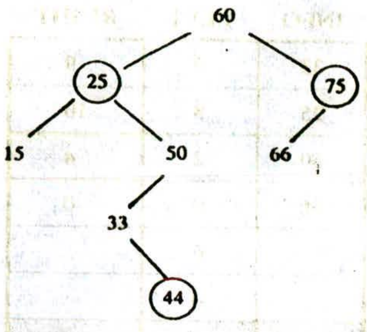
The deletion algorithm first uses Procedure 7.4 to find the location of the node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N. There are three cases:

Case 1. N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

Case 2. N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

Case 3. N has two children. Let S(N) denote the inorder successor of N. (The reader can verify that S(N) does not have a left child.) Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Observe that the third case is much more complicated than the first two cases. In all three cases, the memory space of the deleted node N is returned to the AVAIL list.



(*a*) Before deletions.

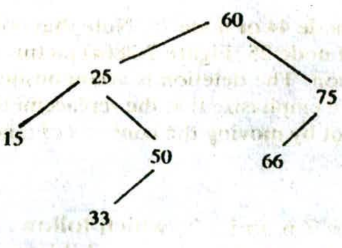
ROOT: 3

AVAIL: 5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

(*b*) Linked representation.

Fig. 7-25



(*a*) Node 44 is deleted.

ROOT: 3

AVAIL: 9

	INFO	LEFT	RIGHT
1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

(*b*) Linked representation.

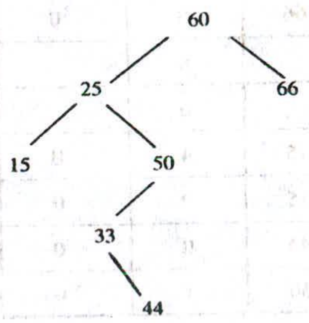
Fig. 7-26

EXAMPLE 7.18

Consider the binary search tree in Fig. 7-25(*a*). Suppose T appears in memory as in Fig. 7-25(*b*).

(*a*) Suppose we delete node 44 from the tree T in Fig. 7-25. Note that node 44 has no children. Figure 7-26(*a*) pictures the tree after 44 is deleted, and Fig. 7-26(*b*) shows the linked representation in memory. The deletion is accomplished by simply assigning NULL to the parent node, 33. (The shading indicates the changes.)

(*b*) Suppose we delete node 75 from the tree T in Fig. 7-25 instead of node 44. Note that node 75 has only one child. Figure 7-27(*a*) pictures the tree after 75 is deleted, and Fig. 7-27(*b*) shows the linked representation. The deletion is accomplished by changing the right pointer of the parent node 60, which originally pointed to 75, so that it now points to node 66, the only child of 75. (The shading indicates the changes.)



(*a*) Node 75 is deleted.

ROOT: 3

AVAIL: 7

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	4
4	66	0	0
5		6	
6		0	
7		5	
8	15	0	0
9	44	0	0
10	50	1	0

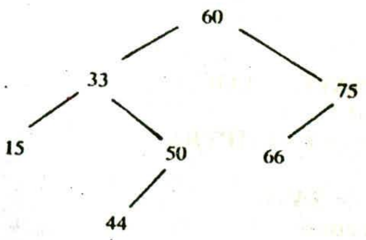
(*b*) Linked representation.

Fig. 7-27

(*c*) Suppose we delete node 25 from the tree T in Fig. 7-25 instead of node 44 or node 75. Note that node 25 has two children. Also observe that node 33 is the inorder successor of node 25. Figure 7-28(*a*) pictures the tree after 25 is deleted, and Fig. 7-28(*b*) shows the linked representation. The deletion is accomplished by first deleting 33 from the tree and then replacing node 25 by node 33. We emphasize that the replacement of node 25 by node 33 is executed in memory only by changing pointers, not by moving the contents of a node from one location to another. Thus 33 is still the value of INFO[1].

Our deletion algorithm will be stated in terms of Procedures 7.6 and 7.7, which follow. The first procedure refers to Cases 1 and 2, where the deleted node N does not have two children; and the second procedure refers to Case 3, where N does have two children. There are many subcases which reflect the fact that N may be a left child, a right child or the root. Also, in Case 2, N may have a left child or a right child.

Procedure 7.7 treats the case that the deleted node N has two children. We note that the inorder successor of N can be found by moving to the right child of N and then moving repeatedly to the left until meeting a node with an empty left subtree.



(*a*) Node 25 is deleted.

ROOT $\boxed{3}$

AVAIL $\boxed{2}$

	INFO	LEFT	RIGHT
1	33	8	10
2		5	
3	60	1	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	9	0

(*b*) Linked representation

Fig. 7-28

Procedure 7.6: CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at location LOC, where N does not have two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer CHILD gives the location of the only child of N, or else CHILD = NULL indicates N has no children.

```
1. [Initializes CHILD.]
   If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then:
       Set CHILD := NULL.
   Else if LEFT[LOC] ≠ NULL, then:
       Set CHILD := LEFT[LOC].
   Else
       Set CHILD := RIGHT[LOC].
   [End of If structure.]
2. If PAR ≠ NULL, then:
       If LOC = LEFT[PAR], then:
           Set LEFT[PAR] := CHILD.
       Else:
           Set RIGHT[PAR] := CHILD.
       [End of If structure.]
   Else:
       Set ROOT := CHILD.
   [End of If structure.]
3. Return.
```

Procedure 7.7: CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure will delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

```
1. [Find SUC and PARSUC.]
   (a) Set PTR := RIGHT[LOC] and SAVE := LOC.
   (b) Repeat while LEFT[PTR] ≠ NULL:
           Set SAVE := PTR and PTR := LEFT[PTR].
       [End of loop.]
   (c) Set SUC := PTR and PARSUC := SAVE.
2. [Delete inorder successor, using Procedure 7.6.]
   Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).
3. [Replace node N by its inorder successor.]
   (a) If PAR ≠ NULL, then:
           If LOC = LEFT[PAR], then:
               Set LEFT[PAR] := SUC.
           Else:
               Set RIGHT[PAR] := SUC.
           [End of If structure.]
       Else:
           Set ROOT := SUC.
       [End of If structure.]
   (b) Set LEFT[SUC] := LEFT[LOC] and
       RIGHT[SUC] := RIGHT[LOC].
4. Return.
```

We can now formally state our deletion algorithm, using Procedures 7.6 and 7.7 as building blocks.

Algorithm 7.8: DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

```
1. [Find the locations of ITEM and its parent, using Procedure 7.4.]
   Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
2. [ITEM in tree?]
   If LOC = NULL, then: Write: ITEM not in tree, and Exit.
3. [Delete node containing ITEM.]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL, then:
       Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   Else:
       Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LEFT[LOC] := AVAIL and AVAIL := LOC.
5. Exit.
```

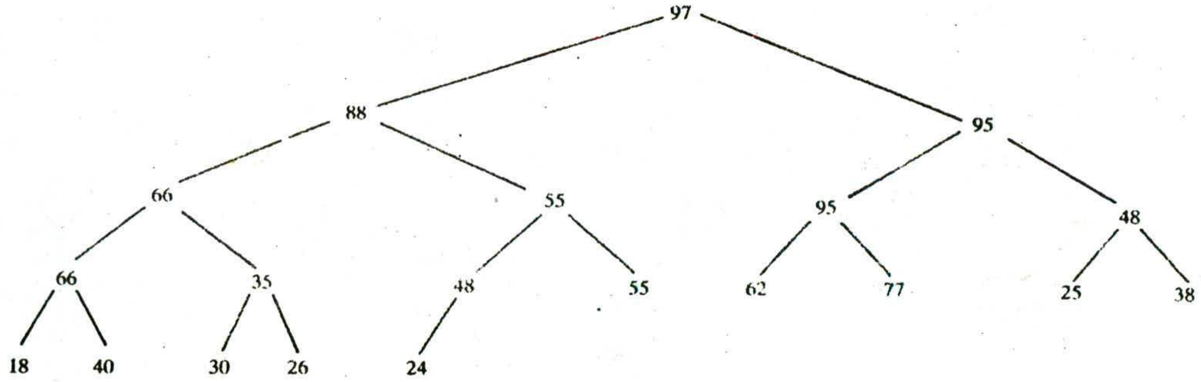
7.10 HEAP; HEAPSORT

This section discusses another tree structure, called a *heap*. The heap is used in an elegant sorting algorithm called *heapsort*. Although sorting will be treated mainly in Chap. 9, we give the heapsort algorithm here and compare its complexity with that of the bubble sort and quicksort algorithms, which were discussed, respectively, in Chaps. 4 and 6.

Suppose H is a complete binary tree with *n* elements. (Unless otherwise stated, we assume that H is maintained in memory by a linear array TREE using the sequential representation of H, not a linked representation.) Then H is called a *heap*, or a *maxheap*, if each node N of H has the following property: *The value at* N *is greater than or equal to the value at each of the children of* N. Accordingly, the value at N is greater than or equal to the value at any of the descendants of N. (A *minheap* is defined analogously: The value at N is less than or equal to the value at any of the children of N.)

EXAMPLE 7.19

Consider the complete tree H in Fig. 7-29(*a*). Observe that H is a heap. This means, in particular, that the largest element in H appears at the "top" of the heap, that is, at the root of the tree. Figure 7-29(*b*) shows the sequential representation of H by the array TREE. That is, TREE[1] is the root of the tree H, and the left and right children of node TREE[K] are, respectively, TREE[2K] and TREE[2K + 1]. This means, in particular, that the parent of any nonroot node TREE[J] is the node TREE[J ÷ 2] (where J ÷ 2 means integer division). Observe that the nodes of H on the same level appear one after the other in the array TREE.



(*a*) Heap.

TREE

97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(*b*) Sequential representation.

Fig. 7-29

Inserting into a Heap

Suppose H is a heap with N elements, and suppose an ITEM of information is given. We insert ITEM into the heap H as follows:

(1) First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.

(2) Then let ITEM rise to its "appropriate place" in H so that H is finally a heap.

We illustrate the way this procedure works before stating the procedure formally.

EXAMPLE 7.20

Consider the heap H in Fig. 7-29. Suppose we want to add ITEM = 70 to H. First we adjoin 70 as the next element in the complete tree; that is, we set TREE[21] = 70. Then 70 is the right child of TREE[10] = 48. The path from 70 to the root of H is pictured in Fig. 7-30(*a*). We now find the appropriate place of 70 in the heap as follows:

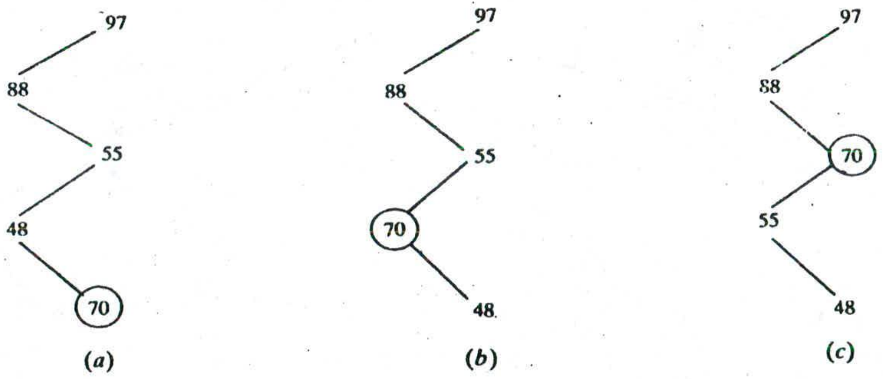
(*a*) Compare 70 with its parent, 48. Since 70 is greater than 48, interchange 70 and 48; the path will now look like Fig. 7-30(*b*).

(*b*) Compare 70 with its new parent, 55. Since 70 is greater than 55, interchange 70 and 55; the path will now look like Fig. 7-30(*c*).

(*c*) Compare 70 with its new parent, 88. Since 70 does not exceed 88, ITEM = 70 has risen to its appropriate place in H.

Figure 7-30(*d*) shows the final tree. A dotted line indicates that an exchange has taken place.

Remark: One must verify that the above procedure does always yield a heap as a final tree, that is, that nothing else has been disturbed. This is easy to see, and we leave this verification to the reader.



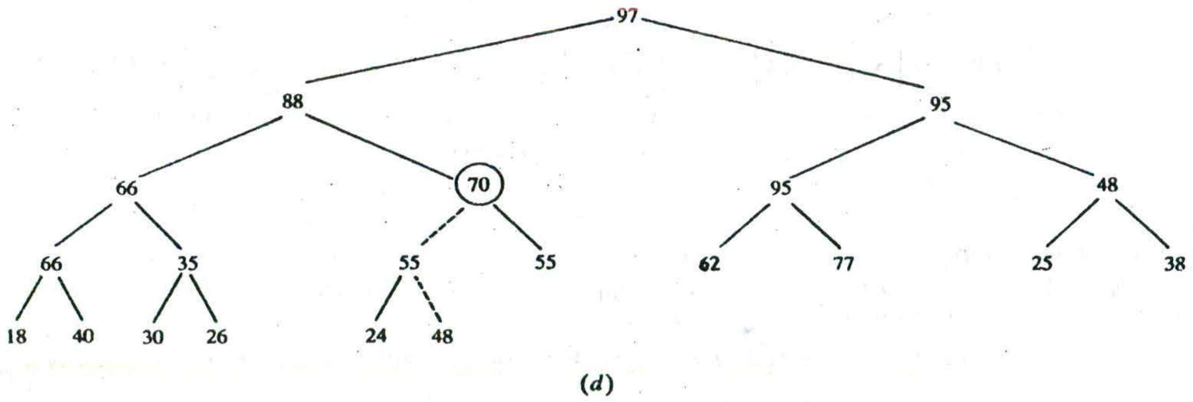


Fig. 7-30 ITEM = 70 is inserted.

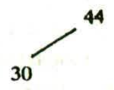
EXAMPLE 7.21

Suppose we want to build a heap H from the following list of numbers:

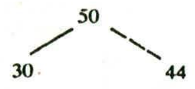
44, 30, 50, 22, 60, 55, 77, 55

This can be accomplished by inserting the eight numbers one after the other into an empty heap H using the above procedure. Figure 7-31(*a*) through (*h*) shows the respective pictures of the heap after each of the eight elements has been inserted. Again, the dotted line indicates that an exchange has taken place during the insertion of the given ITEM of information.

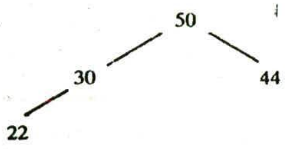
(*a*) ITEM = 44.



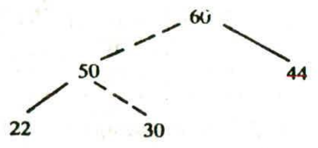
(*b*) ITEM = 30.



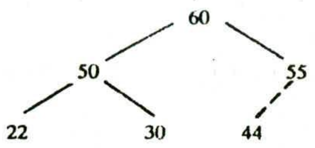
(*c*) ITEM = 50.



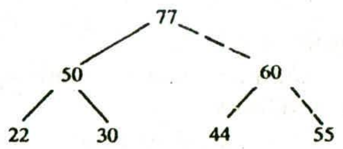
(*d*) ITEM = 22.



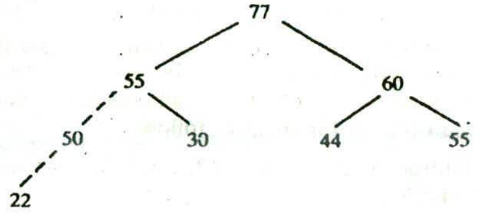
(*e*) ITEM = 60.



(*f*) ITEM = 55.



(*g*) ITEM = 77.



(*h*) ITEM = 55.

Fig. 7-31 Building a heap.

The formal statement of our insertion procedure follows:

```
Procedure 7.9: INSHEAP(TREE, N, ITEM)
               A heap H with N elements is stored in the array TREE, and an ITEM of
               information is given. This procedure inserts ITEM as a new element of H. PTR
               gives the location of ITEM as it rises in the tree, and PAR denotes the location of
               the parent of ITEM.

               1. [Add new node to H and initialize PTR.]
                  Set N := N + 1 and PTR := N.
               2. [Find location to insert ITEM.]
                  Repeat Steps 3 to 6 while PTR < 1.
               3.     Set PAR := ⌊PTR/2⌋. [Location of parent node.]
               4.     If ITEM ≤ TREE[PAR], then:
                          Set TREE[PTR] := ITEM, and Return.
                      [End of If structure.]
               5.     Set TREE[PTR] := TREE[PAR]. [Moves node down.]
               6.     Set PTR := PAR. [Updates PTR.]
                  [End of Step 2 loop.]
               7. [Assign ITEM as the root of H.]
                  Set TREE[1] := ITEM.
               8. Return.
```

Observe that ITEM is not assigned to an element of the array TREE until the appropriate place for ITEM is found. Step 7 takes care of the special case that ITEM rises to the root TREE[1].

Suppose an array A with N elements is given. By repeatedly applying Procedure 7.9 to A, that is, by executing

Call INSHEAP(A, J, A[J + 1])

for J = 1, 2, . . . , N − 1, we can build a heap H out of the array A.

Deleting the Root of a Heap

Suppose H is a heap with N elements, and suppose we want to delete the root R of H. This is accomplished as follows:

(1) Assign the root R to some variable ITEM.

(2) Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.

(3) (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.

Again we illustrate the way the procedure works before stating the procedure formally.

EXAMPLE 7.22

Consider the heap H in Fig. 7-32(*a*), where R = 95 is the root and L = 22 is the last node of the tree. Step 1 of the above procedure deletes R = 95, and Step 2 replaces R = 95 by L = 22. This gives the complete tree in Fig. 7-32(*b*), which is not a heap. Observe, however, that both the right and left subtrees of 22 are still heaps. Applying Step 3, we find the appropriate place of 22 in the heap as follows:

(*a*) Compare 22 with its two children, 85 and 70. Since 22 is less than the larger child, 85, interchange 22 and 85 so the tree now looks like Fig. 7-32(*c*).

(*b*) Compare 22 with its two new children, 55 and 33. Since 22 is less than the larger child, 55, interchange 22 and 55 so the tree now looks like Fig. 7-32(*d*).

(*c*) Compare 22 with its new children, 15 and 20. Since 22 is greater than both children, node 22 has dropped to its appropriate place in H.

Thus Fig. 7-32(*d*) is the required heap H without its original root R.

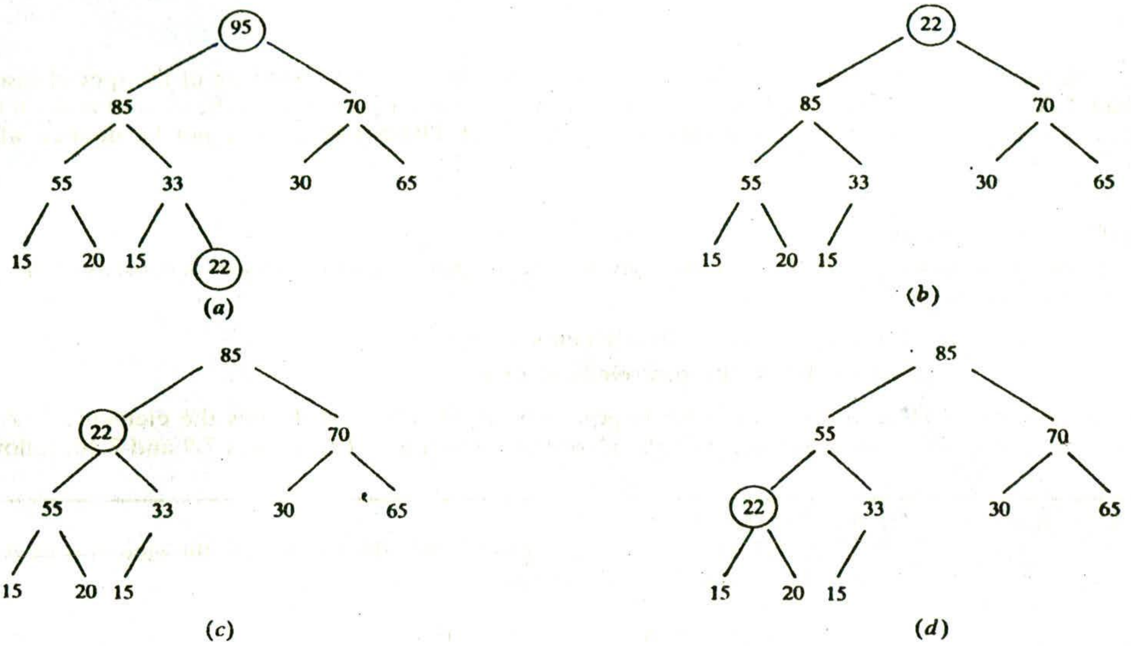


Fig. 7-32 Reheaping.

Remark: As with inserting an element into a heap, one must verify that the above procedure does always yield a heap as a final tree. Again we leave this verification to the reader. We also note that Step 3 of the procedure may not end until the node L reaches the bottom of the tree, i.e., until L has no children.

The formal statement of our procedure follows.

```
Procedure 7.10: DELHEAP(TREE, N, ITEM)
    A heap H with N elements is stored in the array TREE. This procedure assigns
    the root TREE[1] of H to the variable ITEM and then reheaps the remaining
    elements. The variable LAST saves the value of the original last node of H. The
    pointers PTR, LEFT and RIGHT give the locations of LAST and its left and
    right children as LAST sinks in the tree.

     1. Set ITEM := TREE[1]. [Removes root of H.]
     2. Set LAST := TREE[N] and N := N − 1. [Removes last node of H.]
     3. Set PTR := 1, LEFT := 2 and RIGHT := 3. [Initializes pointers.]
     4. Repeat Steps 5 to 7 while RIGHT ≤ N:
     5.     If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT], then:
                Set TREE[PTR] := LAST and Return.
            [End of If structure.]
     6.     IF TREE[RIGHT] ≤ TREE[LEFT], then:
                Set TREE[PTR] := TREE[LEFT] and PTR := LEFT.
            Else:
                Set TREE[PTR] := TREE[RIGHT] and PTR := RIGHT.
            [End of If structure.]
     7.     Set LEFT := 2*PTR and RIGHT := LEFT + 1.
        [End of Step 4 loop.]
     8. If LEFT = N and if LAST < TREE[LEFT], then: Set PTR := LEFT.
     9. Set TREE[PTR] := LAST.
    10. Return.
```

The Step 4 loop repeats as long as LAST has a right child. Step 8 takes care of the special case in which LAST does not have a right child but does have a left child (which has to be the last node in H). The reason for the two "If" statements in Step 8 is that TREE[LEFT] may not be defined when LEFT > N.

Application to Sorting

Suppose an array A with N elements is given. The heapsort algorithm to sort A consists of the two following phases:

Phase A: Build a heap H out of the elements of A.

Phase B: Repeatedly delete the root element of H.

Since the root of H always contains the largest node in H, Phase B deletes the elements of A in decreasing order. A formal statement of the algorithm, which uses Procedures 7.9 and 7.10, follows.

```
Algorithm 7.11: HEAPSORT(A, N)
    An array A with N elements is given. This algorithm sorts the elements of A.
    1. [Build a heap H, using Procedure 7.9.]
       Repeat for J = 1 to N - 1:
           Call INSHEAP(A, J, A[J + 1]).
       [End of loop.]
    2. [Sort A by repeatedly deleting the root of H, using Procedure 7.10.]
       Repeat while N > 1:
           (a) Call DELHEAP(A, N, ITEM).
           (b) Set A[N + 1] := ITEM.
       [End of Loop.]
    3. Exit.
```

The purpose of Step 2(*b*) is to save space. That is, one could use another array B to hold the sorted elements of A and replace Step 2(*b*) by

Set B[N + 1] := ITEM

However, the reader can verify that the given Step 2(*b*) does not interfere with the algorithm, since A[N + 1] does not belong to the heap H.

Complexity of Heapsort

Suppose the heapsort algorithm is applied to an array A with n elements. The algorithm has two phases, and we analyze the complexity of each phase separately.

Phase A. Suppose H is a heap. Observe that the number of comparisons to find the appropriate place of a new element ITEM in H cannot exceed the depth of H. Since H is a complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. Accordingly, the total number $g(n)$ of comparisons to insert the n elements of A into H is bounded as follows:

$$g(n) \leq n \log_2 n$$

Consequently, the running time of Phase A of heapsort is proportional to $n \log_2 n$.

Phase B. Suppose H is a complete tree with m elements, and suppose the left and right subtrees of H are heaps and L is the root of H. Observe that reheaping uses 4 comparisons to move the node L one step down the tree H. Since the depth of H does not exceed $\log_2 m$, reheaping uses at most $4 \log_2 m$ comparisons to find the appropriate place of L in the tree H. This means that the total number $h(n)$ of comparisons to delete the n elements of A from H, which requires reheaping n times, is bounded as follows:

$$h(n) \leq 4n \log_2 n$$

Accordingly, the running time of Phase B of heapsort is also proportional to $n \log_2 n$.

Since each phase requires time proportional to $n \log_2 n$, the running time to sort the n-element array A using heapsort is proportional to $n \log_2 n$, that is, $f(n) = O(n \log_2 n)$. Observe that this gives a worst-case complexity of the heapsort algorithm. This contrasts with the following two sorting algorithms already studied:

(1) *Bubble sort* (Sec. 4.6). The running time of bubble sort is $O(n^2)$.

(2) *Quicksort* (Sec. 6.5). The average running time of quicksort is $O(n \log_2 n)$, the same as heapsort, but the worst-case running time of quicksort is $O(n^2)$, the same as bubble sort.

Other sorting algorithms are investigated in Chap. 9.

7.11 PATH LENGTHS; HUFFMAN'S ALGORITHM

Recall that an *extended binary tree* or *2-tree* is a binary tree T in which each node has either 0 or 2 children. The nodes with 0 children are called *external nodes*, and the nodes with 2 children are called *internal nodes*. Figure 7-33 shows a 2-tree where the internal nodes are denoted by circles and the external nodes are denoted by squares. In any 2-tree, the number N_E of external nodes is 1 more than the number N_I of internal nodes; that is,

$$N_E = N_I + 1$$

For example, for the 2-tree in Fig. 7-33, $N_I = 6$, and $N_E = N_I + 1 = 7$.

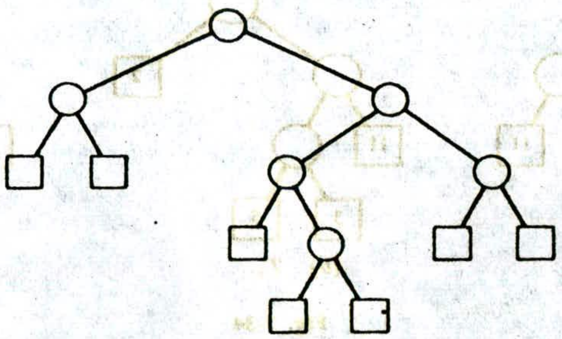


Fig. 7-33

Frequently, an algorithm can be represented by a 2-tree T where the internal nodes represent tests and the external nodes represent actions. Accordingly, the running time of the algorithm may depend on the lengths of the paths in the tree. With this in mind, we define the *external path length* L_E of a 2-tree T to be the sum of all path lengths summed over each path from the root R of T to an external node. The *internal path length* L_I of T is defined analogously, using internal nodes instead of external nodes. For the tree in Fig. 7-33,

$$L_E = 2 + 2 + 3 + 4 + 4 + 3 + 3 = 21 \qquad \text{and} \qquad L_I = 0 + 1 + 1 + 2 + 3 + 2 = 9$$

Observe that

$$L_I + 2n = 9 + 2 \cdot 6 = 9 + 12 = 21 = L_E$$

where $n = 6$ is the number of internal nodes. In fact, the formula

$$L_E = L_I + 2n$$

is true for any 2-tree with n internal nodes.

Suppose T is a 2-tree with n external nodes, and suppose each of the external nodes is assigned a (nonnegative) weight. The (external) *weighted path length* P of the tree T is defined to be the sum of the weighted path lengths; i.e.,

$$P = W_1L_1 + W_2L_2 + \cdots + W_nL_n$$

where W_i and L_i denote, respectively, the weight and path length of an external node N_i.

Consider now the collection of all 2-trees with n external nodes. Clearly, the complete tree among them will have a minimal external path length L_E. On the other hand, suppose each tree is given the same n weights for its external nodes. Then it is not clear which tree will give a minimal weighted path length P.

EXAMPLE 7.23

Figure 7-34 shows three 2-trees, T_1, T_2 and T_3, each having external nodes with weights 2, 3, 5 and 11. The weighted path lengths of the three trees are as follows:

$$P_1 = 2 \cdot 2 + 3 \cdot 2 + 5 \cdot 2 + 11 \cdot 2 = 42$$
$$P_2 = 2 \cdot 1 + 3 \cdot 3 + 5 \cdot 3 + 11 \cdot 2 = 48$$
$$P_3 = 2 \cdot 3 + 3 \cdot 3 + 5 \cdot 2 + 11 \cdot 1 = 36$$

The quantities P_1 and P_3 indicate that the complete tree need not give a minimum length P, and the quantities P_2 and P_3 indicate that similar trees need not give the same lengths.

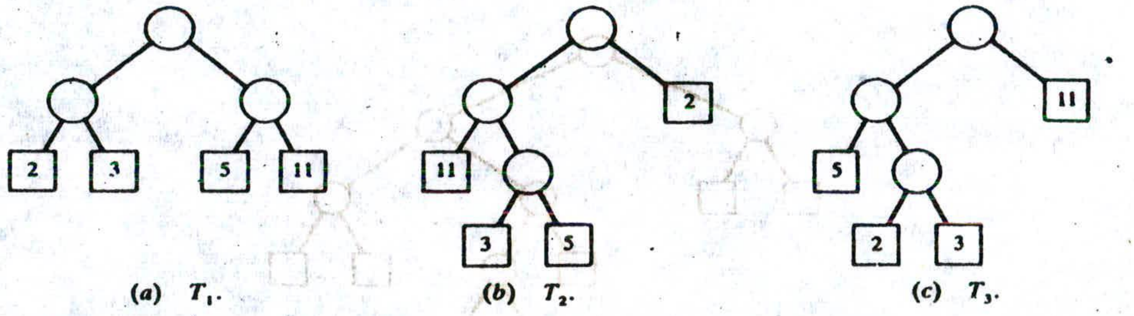


Fig. 7-34

The general problem that we want to solve is as follows. Suppose a list of weights is given:

$$w_1, w_2, \ldots, w_n$$

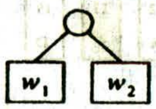
Among all the 2-trees with n external nodes and with the given n weights, find a tree T with a minimum-weighted path length. (Such a tree T is seldom unique.) Huffman gave an algorithm, which we now state, to find such a tree T.

Observe that the Huffman algorithm is recursively defined in terms of the number of weights and the solution for one weight is simply the tree with one node. On the other hand, in practice, we use an equivalent iterated form of the Huffman algorithm constructing the tree from the bottom up rather than from the top down.

Huffman's Algorithm: Suppose w_1 and w_2 are two minimum weights among the n given weights $w_1, w_2, \ldots, w_n$. Find a tree T' which gives a solution for the $n-1$ weights

$$w_1 + w_2, w_3, w_4, \ldots, w_n$$

Then, in the tree T', replace the external node

$\boxed{w_1 + w_2}$ by the subtree 

The new 2-tree T is the desired solution.

EXAMPLE 7.24

Suppose A, B, C, D, E, F, G and H are 8 data items, and suppose they are assigned weights as follows:

Data item:	A	B	C	D	E	F	G	H
Weight:	22	5	11	19	2	11	25	5

Figure 7-35(*a*) through (*h*) shows how to construct the tree T with minimum-weighted path length using the above data and Huffman's algorithm. We explain each step separately.

(*a*) Here each data item belongs to its own subtree. Two subtrees with the smallest possible combination of weights, the one weighted 2 and one of those weighted 5, are shaded.

(*b*) Here the subtrees that were shaded in Fig. 7-35(*a*) are joined together to form a subtree with weight 7. Again, the current two subtrees of lowest weight are shaded

(*c*) to (*g*) Each step joins together two subtrees having the lowest existing weights (always the ones that were shaded in the preceding diagram), and again, the two resulting subtree of lowest weight are shaded.

(*h*) This is the final desired tree T, formed when the only two remaining subtrees are joined together.

Computer Implementation of Huffman's Algorithm

Consider again the data in Example 7.24. Suppose we want to implement the Huffman algorithm using the computer. First of all, we require an extra array WT to hold the weights of the nodes; i.e., our tree will be maintained by four parallel arrays, INFO, WT, LEFT and RIGHT. Figure 7-36(*a*) shows how the given data may be stored in the computer initially. Observe that there is sufficient room for the additional nodes. Observe that NULL appears in the left and right pointers for the initial nodes, since these nodes will be terminal in the final tree.

During the execution of the algorithm, one must be able to keep track of all the different subtrees and one must also be able to find the subtrees with minimum weights. This may be accomplished by maintaining an auxiliary minheap, where each node contains the weight and the location of the root of a current subtree. The initial minheap appears in Fig. 7-36(*b*). (The minheap is used rather than a maxheap since we want the node with the lowest weight to be on the top of the heap.)

The first step in building the required Huffman tree T involves the following substeps:

(i) Remove the node $N_1 = [2, 5]$ and the node $N_2 = [5, 2]$ from the heap. (Each time a node is deleted, one must reheap.)

(ii) Use the data in N_1 and N_2 and the first available space AVAIL = 9 to add a new node as follows:

$$WT[9] = 2 + 5 = 7 \qquad LEFT[9] = 5 \qquad RIGHT[9] = 2$$

Thus N_1 is the left child of the new node and N_2 is the right child of the new node.

(iii) Adjoin the weight and location of the new node, that is, [7, 9], to the heap.

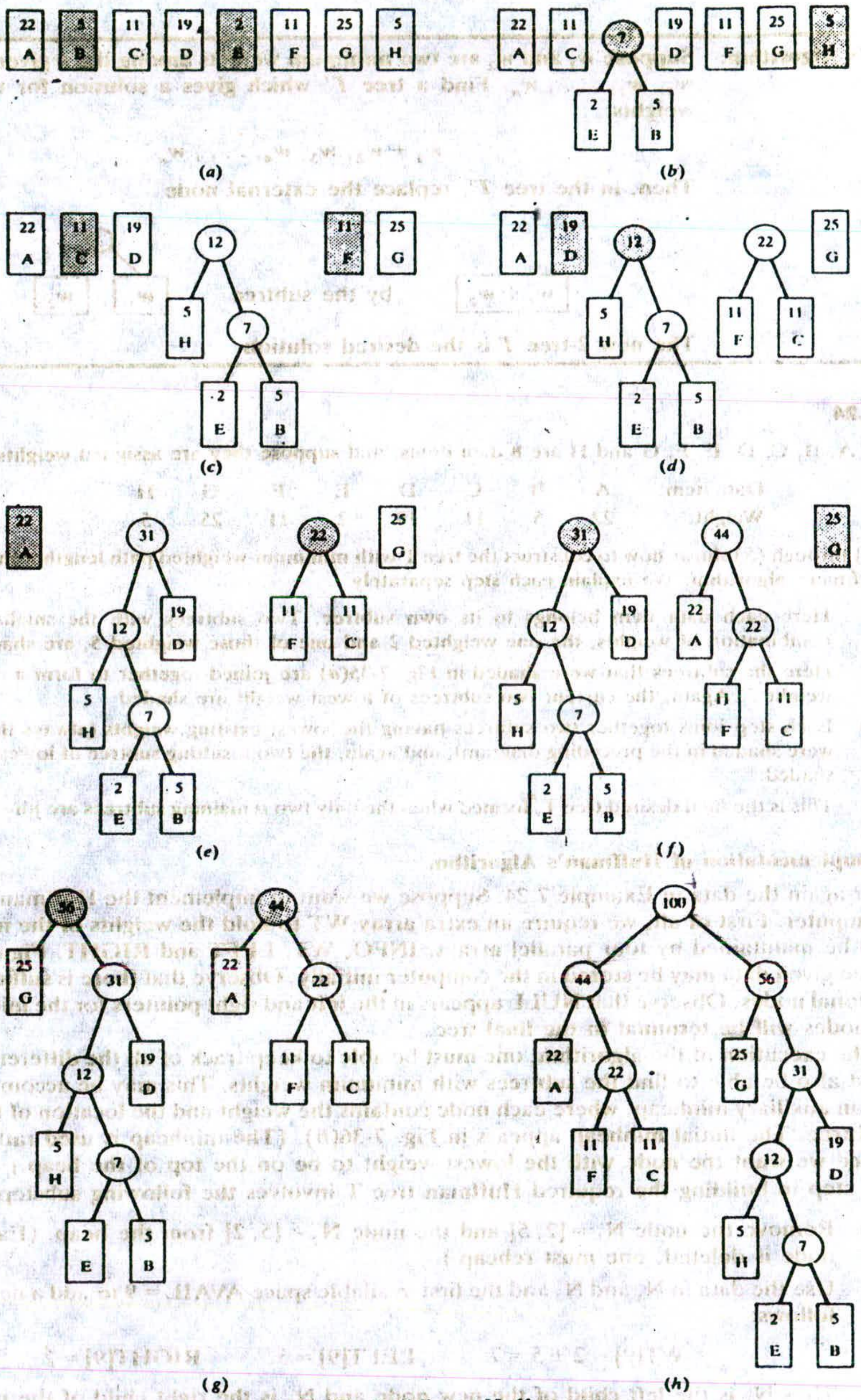


Fig. 7-35 Building a Huffman tree.

	INFO	WT	LEFT	RIGHT
1	A	22	0	0
2	B	5	0	0
3	C	11	0	0
4	D	19	0	0
5	E	2	0	0
6	F	11	0	0
7	G	25	0	0
8	H	5	0	0
9			10	
10			11	
11			12	
12			13	
13			14	
14			15	
15			16	
16			0	

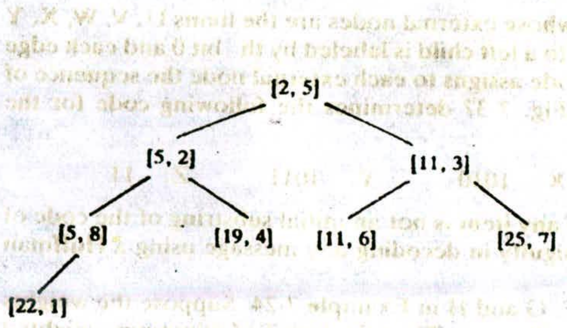
AVAIL = 9

(*a*)

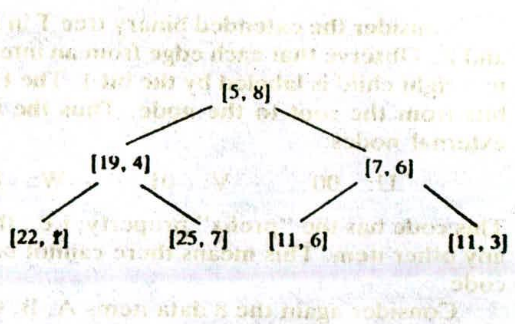
	INFO	WT	LEFT	RIGHT
1	A	22	0	0
2	B	5	0	0
3	C	11	0	0
4	D	19	0	0
5	E	2	0	0
6	F	11	0	0
7	G	25	0	0
8	H	5	0	0
9		7	5	2
10		12	8	9
11		22	6	3
12		31	10	4
13		44	1	11
14		56	7	12
15		100	13	14
16				

ROOT = 15, AVAIL = 16

(*c*)



(*b*)



(*d*)

Fig. 7-36 Implementation of Huffman's algorithmn.

The shaded area in Fig. 7-36(c) shows the new node, and Fig. 7-36(d) shows the new heap, which has one less element than the heap in Fig. 7-36(b).

Repeating the above step until the heap is empty, we obtain the required tree T in Fig. 7-36(c). We must set ROOT = 15, since this is the location of the last node added to the tree.

Application to Coding

Suppose a collection of n data items, $A_1, A_2, \ldots, A_N$, are to be coded by means of strings of bits. One way to do this is to code each item by an r-bit string where

$$2^{r-1} < n \le 2^r$$

For example, a 48-character set is frequently coded in memory by using 6-bit strings. One cannot use 5-bit strings, since $2^5 < 48 < 2^6$.

Suppose the data items do not occur with the same probability. Then memory space may be conserved by using variable-length strings, where items which occur frequently are assigned shorter strings and items which occur infrequently are assigned longer strings. This section discusses a coding using variable-length strings that is based on the Huffman tree T for weighted data items.

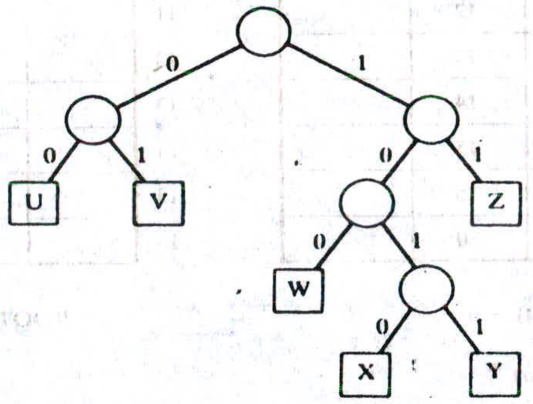


Fig. 7-37

Consider the extended binary tree T in Fig. 7-37 whose external nodes are the items U, V, W, X, Y and Z. Observe that each edge from an internal node to a left child is labeled by th bit 0 and each edge to a right child is labeled by the bit 1. The Huffman code assigns to each external node the sequence of bits from the root to the node. Thus the tree T in Fig. 7-37 determines the following code for the external nodes:

U: 00	V: 01	W: 100	X: 1010	Y: 1011	Z: 11

This code has the "prefix" property; i.e., the code of any item is not an initial substring of the code of any other item. This means there cannot be any ambiguity in decoding any message using a Huffman code.

Consider again the 8 data items A, B, C, D, E, F, G and H in Example 7-24. Suppose the weights represent the percentage probabilities that the items will occur. Then the tree T of minimum-weighted path length constructed in Fig. 7-35, appearing with the bit labels in Fig. 7-38, will yield an efficient coding of the data items. The reader can verify that the tree T yields the following code:

A: 00	B: 11011	C: 011	D: 111
E: 11010	F: 010	G: 10	H: 1100

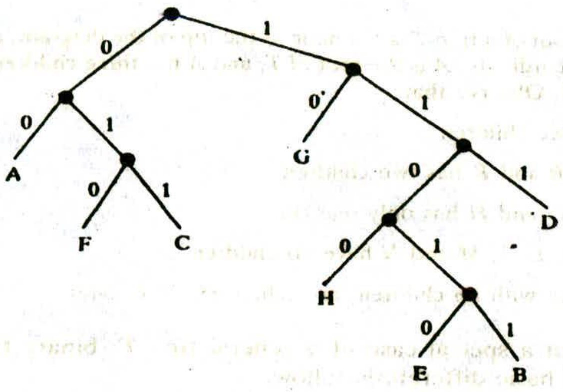


Fig. 7-38

7.12 GENERAL TREES

A *general tree* (sometimes called a *tree*) is defined to be a nonempty finite set T of elements, called *nodes*, such that:

(1) T contains a distinguished element R, called the *root* of T.

(2) The remaining elements of T form an ordered collection of zero or more disjoint trees $T_1, T_2, \ldots, T_m$.

The trees $T_1, T_2, \ldots, T_m$ are called *subtrees* of the root R, and the roots of $T_1, T_2, \ldots, T_m$ are called *successors* of R.

Terminology from family relationships, graph theory and horticulture is used for general trees in the same way as for binary trees. In particular, if N is a node with successors $S_1, S_2, \ldots, S_m$, then N is called the *parent* of the S_i's, the S_i's are called *children* of N, and the S_i's are called *siblings* of each other.

The term "tree" comes up, with slightly different meanings, in many different areas of mathematics and computer science. Here we assume that our general tree T is *rooted*, that is, that T has a distinguished node R called the root of T; and that T is *ordered*, that is, that the children of each node N of T have a specific order. These two properties are not always required for the definition of a tree.

EXAMPLE 7.25

Figure 7-39 pictures a general tree T with 13 nodes,

$$A, B, C, D, E, F, G, H, J, K, L, M, N$$

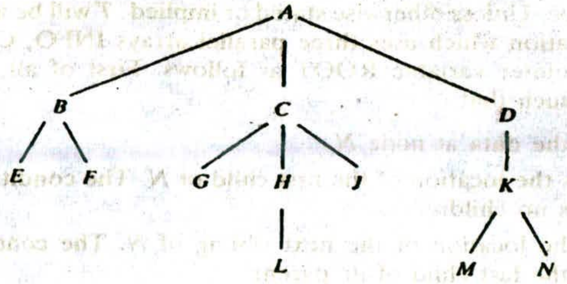


Fig. 7-39

Unless otherwise stated, the root of a tree T is the node at the top of the diagram, and the children of a node are ordered from left to right. Accordingly, A is the root of T, and A has three children; the first child B, the second child C and the third child D. Observe that:

(*a*) The node C has three children.

(*b*) Each of the nodes B and K has two children.

(*c*) Each of the nodes D and H has only one child.

(*d*) The nodes E, F, G, L, J, M and N have no children.

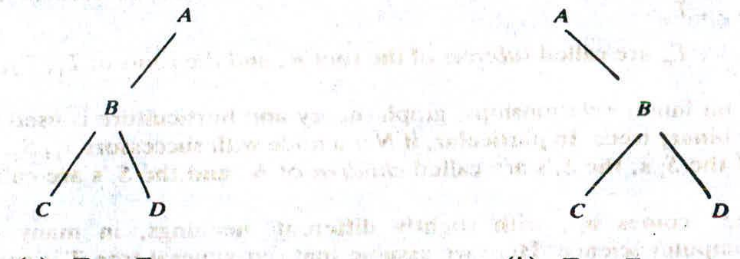
The last group of nodes, those with no children, are called *terminal nodes*.

A binary tree T' is not a special case of a general tree T: binary trees and general trees are different objects. The two basic differences follow:

(1) A binary tree T' may be empty, but a general tree T is nonempty.

(2) Suppose a node N has only one child. Then the child is distinguished as a left child or right child in a binary tree T', but no such distinction exists in a general tree T.

The second difference is illustrated by the trees T_1 and T_2 in Fig. 7-40. Specifically, as binary trees, T_1 and T_2 are distinct trees, since B is the left child of A in the tree T_1 but B is the right child of A in the tree T_2. On the other hand, there is no difference between the trees T_1 and T_2 as general trees.



(*a*) Tree T_1. (*b*) Tree T_2.

Fig. 7-40

A *forest* F is defined to be an ordered collection of zero or more distinct trees. Clearly, if we delete the root R from a general tree T, then we obtain the forest F consisting of the subtrees of R (which may be empty). Conversely, if F is a forest, then we may adjoin a node R to F to form a general tree T where R is the root of T and the subtrees of R consist of the original trees in F.

Computer Representation of General Trees

Suppose T is a general tree. Unless otherwise stated or implied, T will be maintained in memory by means of a linked representation which uses three parallel arrays INFO, CHILD (or DOWN) and SIBL (or HORZ), and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that:

(1) INFO[K] contains the data at node N.

(2) CHILD[K] contains the location of the first child of N. The condition CHILD[K] = NULL indicates that N has no children.

(3) SIBL[K] contains the location of the next sibling of N. The condition SIBL[K] = NULL indicates that N is the last child of its parent.

Furthermore, ROOT will contain the location of the root R of T. Although this representation may seem artificial, it has the important advantage that each node N of T, regardless of the number of children of N, will contain exactly three fields.

The above representation may easily be extended to represent a forest F consisting of trees $T_1, T_2, \ldots, T_m$ by assuming the roots of the trees are siblings. In such a case, ROOT will contain the location of the root R_1 of the first tree T_1; or when F is empty, ROOT will equal NULL.

EXAMPLE 7.26

Consider the general tree T in Fig. 7-39. Suppose the data of the nodes of T are stored in an array INFO as in Fig. 7-41(a). The structural relationships of T are obtained by assigning values to the pointer ROOT and the arrays CHILD and SIBL as follows:

(a) Since the root A of T is stored in INFO[2], set ROOT := 2.

(b) Since the first child of A is the node B, which is stored in INFO[3], set CHILD[2] := 3. Since A has no sibling, set SIBL[2] := NULL.

(c) Since the first child of B is the node E, which is stored in INFO[15], set CHILD[3] := 15. Since node C is the next sibling of B and C is stored in INFO[4], set SIBL[3] := 4.

And so on. Figure 7-41(b) gives the final values in CHILD and SIBL. Observe that the AVAIL list of empty nodes is maintained by the first array, CHILD, where AVAIL = 1.

	INFO
1	
2	A
3	B
4	C
5	
6	G
7	H
8	J
9	N
10	M
11	L
12	K
13	
14	F
15	E
16	D

(a)

	CHILD	SIBL
1	5	
2	3	0
3	15	4
4	6	16
5	13	
6	0	7
7	11	8
8	0	0
9	0	0
10	0	9
11	0	0
12	10	0
13	0	
14	0	0
15	0	14
16	12	0

ROOT = 2, AVAIL = 13

(b)

Fig. 7-41

Correspondence between General Trees and Binary Trees

Suppose T is a general tree. Then we may assign a unique binary tree T' to T as follows. First of all, the nodes of the binary tree T' will be the same as the nodes of the general tree T, and the root of

T' will be the root of T. Let N be an arbitrary node of the binary tree T'. Then the left child of N in T' will be the first child of the node N in the general tree T and the right child of N in T' will be the next sibling of N in the general tree T.

EXAMPLE 7.27

Consider the general tree T in Fig. 7-39. The reader can verify that the binary tree T' in Fig. 7-42 corresponds to the general tree T. Observe that by rotating counterclockwise the picture of T' in Fig. 7-42 until the edges pointing to right children are horizontal, we obtain a picture in which the nodes occupy the same relative position as the nodes in Fig. 7-39.

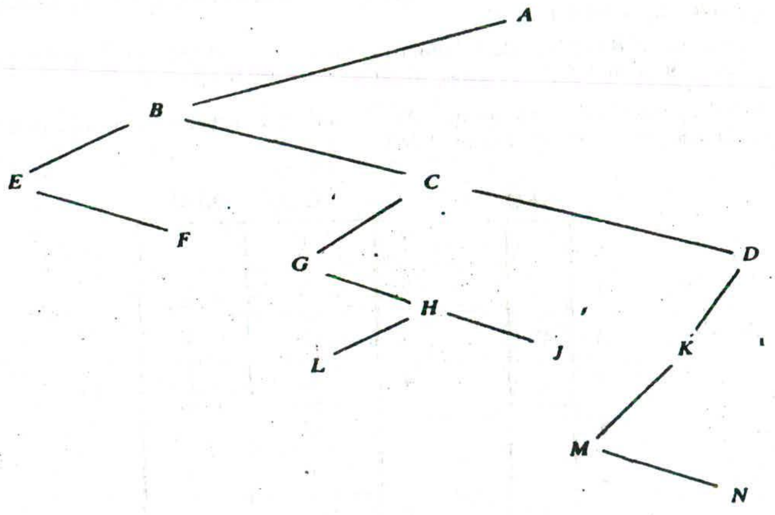


Fig. 7-42 Binary tree T'.

The computer representation of the general tree T and the linked representation of the corresponding binary tree T' are exactly the same except that the names of the arrays CHILD and SIBL for the general tree T will correspond to the names of the arrays LEFT and RIGHT for the binary tree T'. The importance of this correspondence is that certain algorithms that applied to binary trees, such as the traversal algorithms, may now apply to general trees.

Solved Problems

BINARY TREES

7.1 Suppose T is the binary tree stored in memory as in Fig. 7-43. Draw the diagram of T.

The tree T is drawn from its root R downward as follows:

(a) The root R is obtained from the value of the pointer ROOT. Note that ROOT = 5. Hence INFO[5] = 60 is the root R of T.

ROOT $\boxed{5}$

AVAIL $\boxed{9}$

	INFO	LEFT	RIGHT
1	20	0	0
2	30	1	13
3	40	0	0
4	50	0	0
5	60	2	6
6	70	0	8
7	80	0	0
8	90	7	14
9		10	
10		0	
11	35	0	12
12	45	3	4
13	55	11	0
14	95	0	0

Fig. 7-43

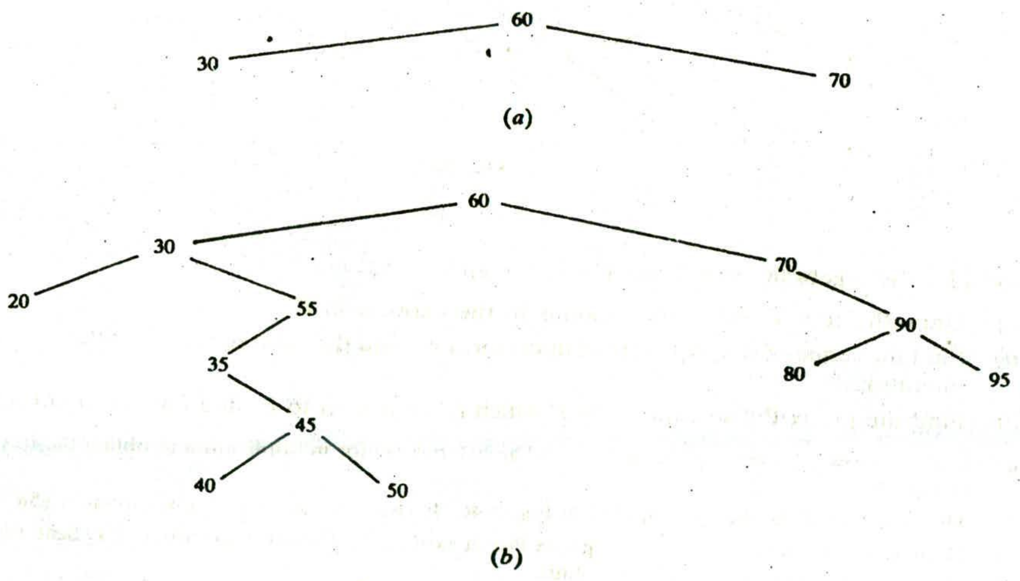


Fig. 7-44

(*b*) The left child of R is obtained from the left pointer field of R. Note that LEFT[5] = 2. Hence INFO[2] = 30 is the left child of R.

(*c*) The right child of R is obtained from the right pointer field of R. Note that RIGHT[5] = 6. Hence INFO[6] = 70 is the right child of R.

We can now draw the top part of the tree as pictured in Fig. 7-44(*a*). Repeating the above process with each new node, we finally obtain the required tree T in Fig. 7-44(*b*).

7.2 A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the following sequences of nodes:

Inorder:	E A C K | F | H D B G
Preorder:	F A E K C D H G B

Draw the tree T.

The tree T is drawn from its root downward as follows.

(*a*) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T.

(*b*) The left child of the node F is obtained as follows. First use the inorder of T to find the nodes in the left subtree T_1 of F. Thus T_1 consists of the nodes E, A, C and K. Then the left child of F is obtained by choosing the first node in the preorder of T_1 (which appears in the preorder of T). Thus A is the left son of F.

(*c*) Similarly, the right subtree T_2 of F consists of the nodes H, D, B and G, and D is the root of T_2, that is, D is the right child of F.

Repeating the above process with each new node, we finally obtain the required tree in Fig. 7-45.

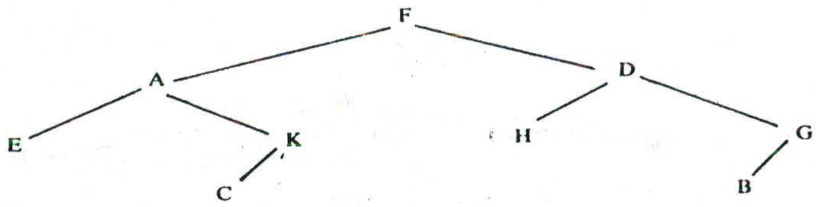


Fig. 7-45

7.3 Consider the algebraic expression $E = (2x + y)(5a - b)^3$.

(*a*) Draw the tree *T* which corresponds to the expression *E*.

(*b*) Find the *scope* of the exponential operator; i.e., find the subtree rooted at the exponential operator.

(*c*) Find the prefix Polish expression *P* which is equivalent to *E*, and find the preorder of *T*.

(*a*) Use an arrow (↑) for exponentiation and an asterisk (*) for multiplication to obtain the tree shown in Fig. 7-46.

(*b*) The scope of ↑ is the tree shaded in Fig. 7-46. It corresponds to the subexpression $(5a - b)^3$.

(*c*) There is no difference between the prefix Polish expression *P* and the preorder of *T*. Scan the tree *T* from the left, as in Fig. 7-12, to obtain:

$$* \quad + \quad * \quad 2 \quad x \quad y \quad \uparrow \quad - \quad * \quad 5 \quad a \quad b \quad 3$$

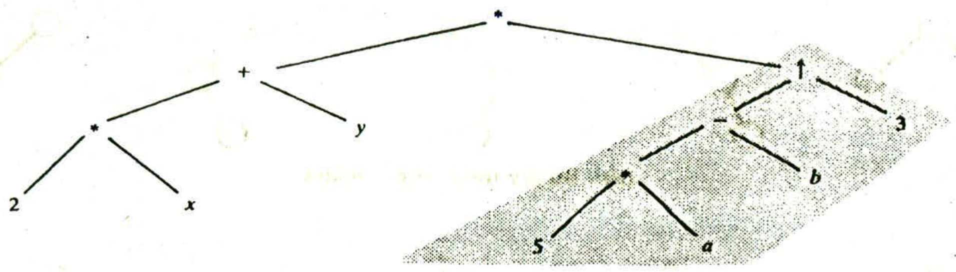


Fig. 7-46

7.4 Suppose a binary tree T is in memory. Write a recursive procedure which finds the number NUM of nodes in T.

The number NUM of nodes in T is 1 more than the number NUML of nodes in the left subtree of T plus the number NUMR of nodes in the right subtree of T. Accordingly:

```
Procedure P7.4: COUNT(LEFT, RIGHT, ROOT, NUM)
    This procedure finds the number NUM of nodes in a binary tree T in memory.

    1. If ROOT = NULL, then: Set NUM := 0, and Return.
    2. Call COUNT(LEFT, RIGHT, LEFT[ROOT], NUML).
    3. Call COUNT(LEFT, RIGHT, RIGHT[ROOT], NUMR).
    4. Set NUM := NUML + NUMR + 1.
    5. Return.
```

(Observe that the array INFO does not play any role in this procedure.)

7.5 Suppose a binary tree T is in memory. Write a recursive procedure which finds the depth DEP of T.

The depth DEP of T is 1 more than the maximum of the depths of the left and right subtrees of T. Accordingly:

```
Procedure P7.5: DEPTH(LEFT, RIGHT, ROOT, DEP)
    This procedure finds the depth DEP of a binary tree T in memory.

    1. If ROOT = NULL, then: Set DEP := 0, and Return.
    2. Call DEPTH(LEFT, RIGHT, LEFT[ROOT], DEPL).
    3. Call DEPTH(LEFT, RIGHT, RIGHT[ROOT], DEPR).
    4. If DEPL ≥ DEPR, then:
           Set DEP := DEPL + 1.
       Else:
           Set DEP := DEPR + 1.
       [End of If structure.]
    5. Return.
```

(Observe that the array INFO does not play any role in this procedure.)

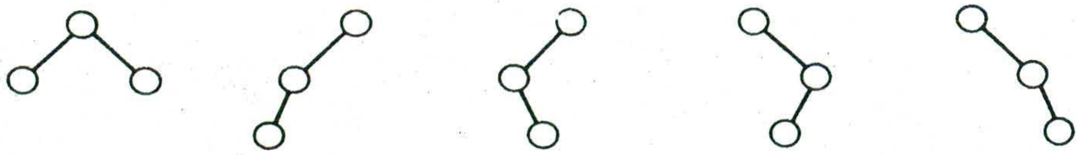
7.6 Draw all the possible nonsimilar trees T where:

(*a*) T is a binary tree with 3 nodes.

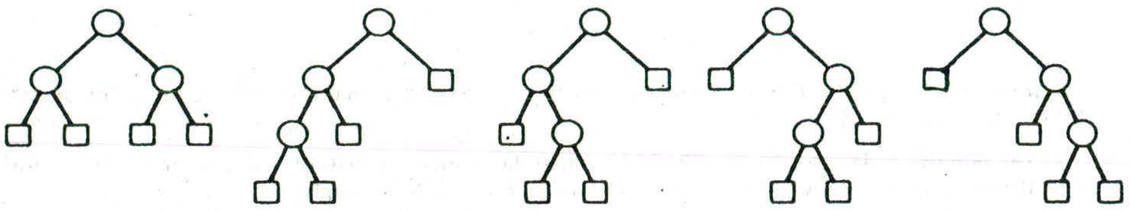
(*b*) T is a 2-tree with 4 external nodes.

(*a*) There are five such trees, which are pictured in Fig. 7-47(*a*).

(*b*) Each 2-tree with 4 external nodes is determined by a binary tree with 3 nodes, i.e., by a tree in part (*a*). Thus there are five such trees, which are pictured in Fig. 7-47(*b*).



(*a*) Binary trees with 3 nodes.



(*b*) Extended binary trees with 4 external nodes.

Fig. 7-47

BINARY SEARCH TREES; HEAPS

7.7 Consider the binary search tree T in Fig. 7-44(*b*), which is stored in memory as in Fig. 7-43. Suppose ITEM = 33 is added to the tree T. (*a*) Find the new tree T. (*b*) Which changes occur in Fig. 7-43?

(*a*) Compare ITEM = 33 with the root, 60. Since 33 < 60, move to the left child, 30. Since 33 > 30, move to the right child, 55. Since 33 < 55, move to the left child, 35. Now 33 < 35, but 35 has no left child. Hence add ITEM = 33 as a left child of the node 35 to give the tree in Fig. 7-48. The shaded edges indicate the path down through the tree during the insertion algorithm.

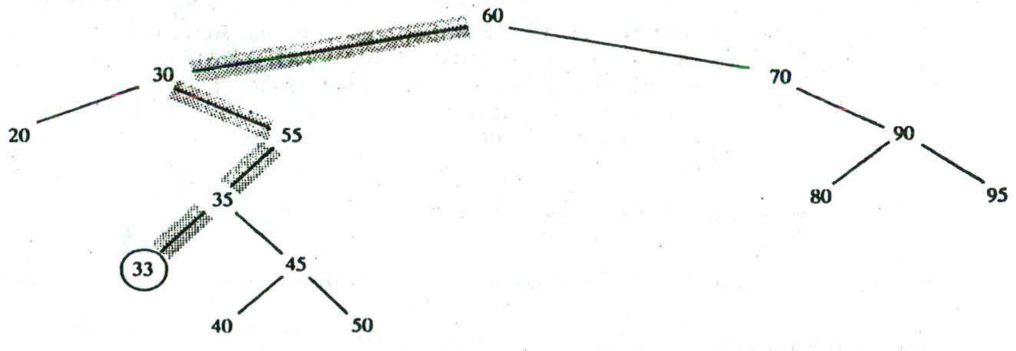


Fig. 7-48

(*b*) First, ITEM = 33 is assigned to the first available node. Since AVAIL = 9, set INFO[9] := 33 and set LEFT[9] := 0 and RIGHT[9] := 0. Also, set AVAIL := 10, the next available node. Finally, set LEFT[11] := 9 so that ITEM = 33 is the left child of INFO[11] = 35. Figure 7-49 shows the updated tree T in memory. The shading indicates the changes from the original picture.

ROOT $\boxed{5}$

AVAIL $\boxed{10}$

	INFO	LEFT	RIGHT
1	20	0	0
2	30	1	13
3	40	0	0
4	50	0	0
5	60	2	6
6	70	0	8
7	80	0	0
8	90	7	14
9	33	0	0
10		0	
11	35	9	12
12	45	3	4
13	55	11	0
14	95	0	0

Fig. 7-49

7.8 Suppose the following list of letters is inserted in order into an empty binary search tree:

J, R, D, G, T, E, M, H, P, A, F, Q

(*a*) Find the final tree T and (*b*) find the inorder traversal of T.

(*a*) Insert the nodes one after the other to obtain the tree in Fig. 7-50.

(*b*) The inorder traversal of T follows:

A, D, E, F, G, H, J, M, P, Q, R, T

Observe that this is the alphabetical listing of the letters.

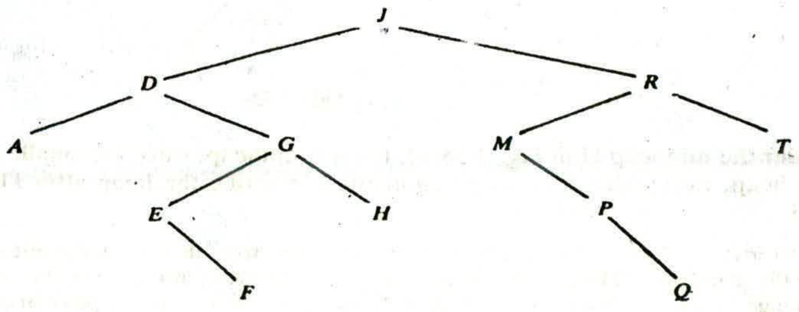


Fig. 7-50

7.9 Consider the binary search tree T in Fig. 7-50. Describe the tree after (*a*) the node *M* is deleted and (*b*) the node *D* is also deleted.

(*a*) The node *M* has only one child, *P*. Hence delete *M* and let *P* become the left child of *R* in place of *M*.

(*b*) The node *D* has two children. Find the inorder successor of *D*, which is the node *E*. First delete *E* from the tree, and then replace *D* by the node *E*.

Figure 7-51 shows the updated tree.

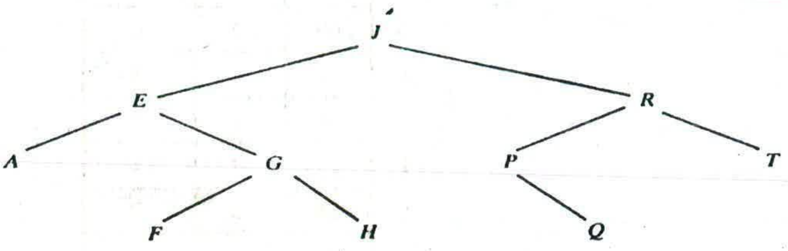


Fig. 7-51

7.10 Suppose *n* data items $A_1, A_2, \ldots, A_N$ are already sorted, i.e.,

$$A_1 < A_2 < \cdots < A_N$$

(*a*) Assuming the items are inserted in order into an empty binary search tree, describe the final tree T.

(*b*) What is the depth D of the tree T?

(*c*) Compare D with the average depth AD of a binary search tree with n nodes for (i) $n = 50$, (ii) $n = 100$ and (iii) $n = 500$.

(*a*) The tree will consist of one branch which extends to the right, as pictured in Fig. 7-52.

(*b*) Since T has a branch with all n nodes, $D = n$.

(*c*) It is known that $AD = c \log_2 n$, where $c \approx 1.4$. Hence $D(50) = 50$, $AD(50) \approx 9$; $D(100) = 100$, $AD(100) \approx 10$; $D(500) = 500$, $AD(500) \approx 12$.

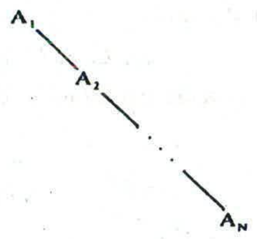
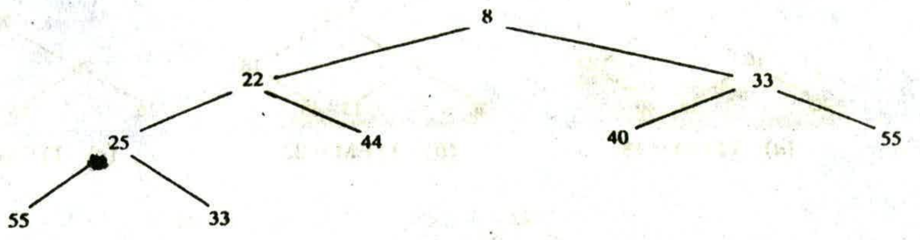


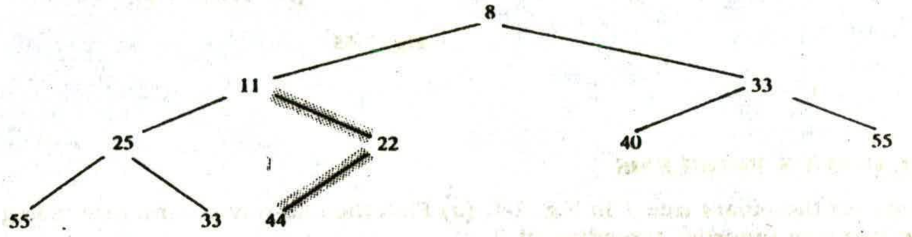
Fig. 7-52

7.11 Consider the minheap H in Fig. 7-53(*a*). (H is a minheap, since the smaller elements are on top of the heap, rather than the larger elements.) Describe the heap after ITEM = 11 is inserted into H.

First insert ITEM as the next node in the complete tree, that is, as the left child of node 44. Then repeatedly compare ITEM with its parent. Since $11 < 44$, interchange 11 and 44. Since $11 < 22$, interchange 11 and 22. Since $11 > 8$, ITEM = 11 has found its appropriate place in the heap. Figure 7-53(*b*) shows the updated heap H. The shaded edges indicate the path of ITEM up the tree.



(*a*)



(*b*)

Fig. 7-53

7.12 Consider the complete tree T with N = 6 nodes in Fig. 7-54. Suppose we form a heap out of T by applying

Call INSHEAP(A, J, A[J + 1])

for J = 1, 2, . . . , N − 1. (Here T is stored sequentially in the array A.) Describe the different steps.

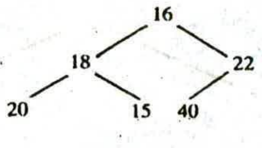


Fig. 7-54

Figure 7-55 shows the different steps. We explain each step separately.

(*a*) J = 1 and ITEM = A[2] = 18. Since 18 > 16, interchange 18 and 16.

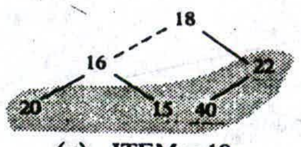
(*b*) J = 2 and ITEM = A[3] = 22. Since 22 > 18, interchange 22 and 18.

(*c*) J = 3 and ITEM = A[4] = 20. Since 20 > 16 but 20 < 22, interchange only 20 and 16.

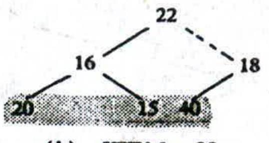
(*d*) J = 4 and ITEM = A[5] = 15. Since 15 < 20, no interchanges take place.

(*e*) J = 5 and ITEM = A[6] = 40. Since 40 > 18 and 40 > 22, first interchange 40 and 18 and then interchange 40 and 22.

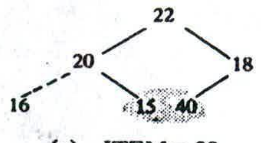
The tree is now a heap. The dotted edges indicate that an exchange has taken place. The unshaded area indicates that part of the tree which forms a heap. Observe that the heap is created from the top down (although individual elements move up the tree).



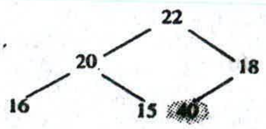
(*a*) ITEM = 18.



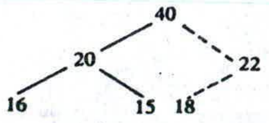
(*b*) ITEM = 22.



(*c*) ITEM = 20.



(*d*) ITEM = 15.



(*e*) ITEM = 40.

Fig. 7-55

MISCELLANEOUS PROBLEMS

7.13 Consider the binary tree T in Fig. 7-1. (*a*) Find the one-way preorder threading of T. (*b*) Find the two-way preorder threading of T.

(*a*) Replace the right null subtree of a terminal node N by a thread pointing to the successor of N in the preorder traversal of T. Thus there is a thread from D to E, since E is visited after D in the preorder traversal of T. Similarly, there is a thread from F to C, from G to H and from L to K. The threaded tree appears in Fig. 7-56. The terminal node K has no thread, since it is the last node in the preorder traversal of T. (On the other hand, if T had a header node Z, then there would be a thread from K back to Z.)

(*b*) There is no two-way preorder threading of T that is analogous to the two-way inorder threading of T

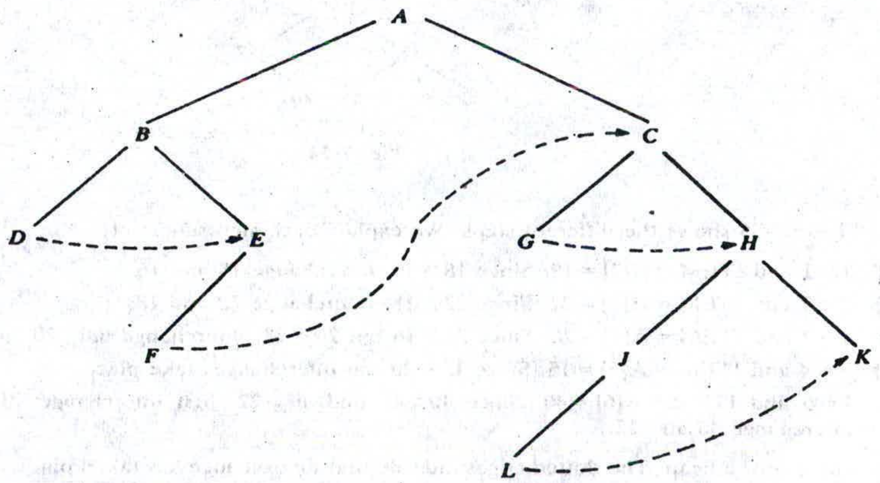


Fig. 7-56 Preorder threaded tree.

7.14 **Consider the weighted 2-tree T in Fig. 7-57. Find the weighted path length P of the tree T.**

Multiply each weight W_i by the length L_i of the path from the root of T to the node containing the weight, and then sum all such products to obtain P. Thus:

$$P = 4 \cdot 2 + 15 \cdot 4 + 25 \cdot 4 + 5 \cdot 3 + 8 \cdot 2 + 16 \cdot 2 = 8 + 60 + 100 + 15 + 16 + 32 = 231$$

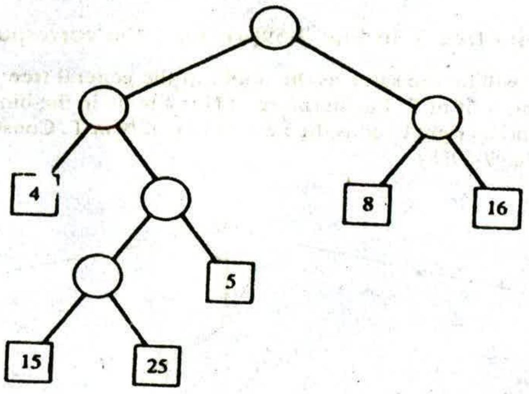


Fig. 7-57

7.15 **Suppose the six weights 4, 15, 25, 5, 8, 16 are given. Find a 2-tree T with the given weights and a minimum weighted path length P. (Compare T with the tree in Fig. 7-57.)**

Use the Huffman algorithm. That is, repeatedly combine the two subtrees with minimum weights into a single subtree as follows:

(*a*)	4,	15,	25,	5,	8,	16
(*b*)		15,	25,	$\bigcirc$ 9,	8,	16
(*c*)		15,	25,		$\bigcirc$ 17,	16
(*d*)			25,		17,	$\bigcirc$ 31
(*e*)					$\bigcirc$ 42,	31
(*f*)						$\bigcirc$ 73

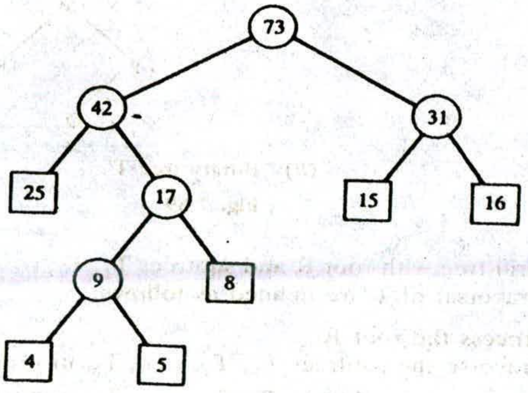


Fig. 7-58

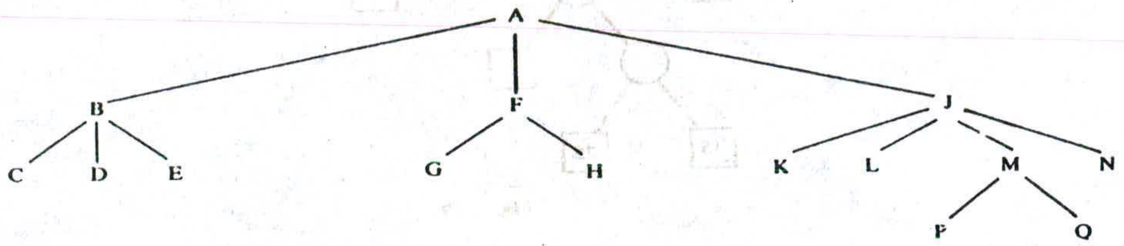
(The circled number indicates the root of the new subtree in the step.) The tree T is drawn from Step (f) backward, yielding Fig. 7-58. With the tree T, compute

$$P = 25 \cdot 2 + 4 \cdot 4 + 5 \cdot 4 + 8 \cdot 3 + 15 \cdot 2 + 16 \cdot 2 = 50 + 16 + 20 + 24 + 30 + 32 = 172$$

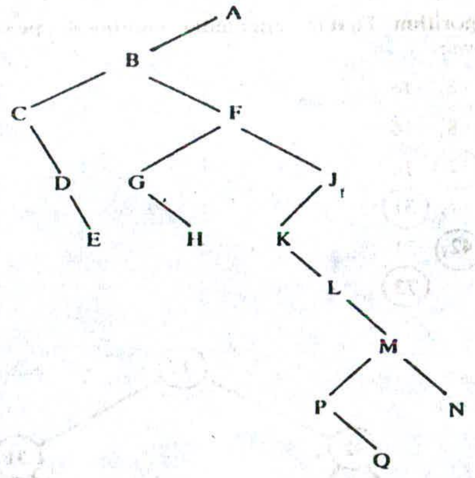
(The tree in Fig. 7-57 has weighted path length 231.)

7.16 Consider the general tree T in Fig. 7-59(a). Find the corresponding binary tree T'.

The nodes of T' will be the same as the nodes of the general tree T, and in particular, the root of T' will be the same as the root of T. Furthermore, if N is a node in the binary tree T', then its left child is the first child of N in T and its right child is the next sibling of N in T. Constructing T' from the root down, we obtain the tree in Fig. 7-59(b).



(a) General tree T.



(b) Binary tree T'.

Fig. 7-59

7.17 Suppose T is a general tree with root R and subtrees $T_1, T_2, \ldots, T_M$. The preorder traversal and the postorder traversal of T are defined as follows:

Preorder: (1) Process the root R.
(2) Traverse the subtrees $T_1, T_2, \ldots, T_M$ in preorder.

Postorder: (1) Traverse the subtrees $T_1, T_2, \ldots, T_M$ in postorder.
(2) Process the root R.

Let T be the general tree in Fig. 7-59(*a*). (*a*) Traverse T in preorder. (*b*) Traverse T in postorder.

Note that T has the root A and subtrees T_1, T_2 and T_3 such that:

T_1 consists of nodes B, C, D and E.
T_2 consists of nodes F, G and H.
T_3 consists of nodes J, K, L, M, N, P and Q.

(*a*) The preorder traversal of T consists of the following steps:

(i)	Process root A.	
(ii)	Traverse T_1 in preorder:	Process nodes B, C, D, E.
(iii)	Traverse T_2 in preorder:	Process nodes F, G, H.
(iv)	Traverse T_3 in preorder:	Process nodes J, K, L, M, P, Q, N.

That is, the preorder traversal of T is as follows:

A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N

(*b*) The postorder traversal of T consists of the following steps:

(i)	Traverse T_1 in postorder:	Process nodes C, D, E, B.
(ii)	Traverse T_2 in postorder:	Process nodes G, H, F.
(iii)	Traverse T_3 in postorder:	Process nodes K, L, P, Q, M, N, J.
(iv)	Process root A.	

In other words, the postorder traversal of T is as follows:

C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

7.18 Consider the binary tree T′ in Fig. 7-59(*b*). Find the preorder, inorder and postorder traversals of T′. Compare them with the preorder and postorder traversals obtained in Prob. 7.17 of the general tree T in Fig. 7-59(*a*).

Using the binary tree traversal algorithms in Sec. 7.4, we obtain the following traversals of T′:

Preorder:	A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N
Inorder:	C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A
Postorder:	E, D, C, H, G, Q, P, N, M, L, K, J, F, B, A

Observe that the preorder of the binary tree T′ is identical to the preorder of the general T, and that the inorder traversal of the binary tree T′ is identical to the postorder traversal of the general tree T. There is no natural traversal of the general tree T which corresponds to the postorder traversal of its corresponding binary tree T′.

Supplementary Problems

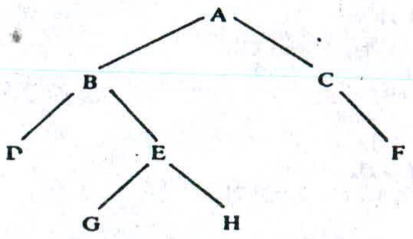
BINARY TREES

7.19 Consider the tree T in Fig. 7-60(*a*).

(*a*) Fill in the values for ROOT, LEFT and RIGHT in Fig. 7-60(*b*) so that T will be stored in memory.
(*b*) Find (i) the depth D of T, (ii) the number of null subtrees and (iii) the descendants of node B.

7.20 List the nodes of the tree T in Fig. 7-60(*a*) in (*a*) preorder, (*b*) inorder and (*c*) postorder

7.21 Draw the diagram of the tree T in Fig. 7-61.



(*a*)

ROOT

AVAIL 5

	INFO	LEFT	RIGHT
1	A		
2	C		
3	D		
4	G		
5		6	
6		0	
7	H		
8	F		
9	E		
10	B		

(*b*)

Fig. 7-60

ROOT 14

AVAIL 8

	INFO	LEFT	RIGHT
1	H	4	11
2	R	0	0
3		17	
4	P	0	0
5	B	18	7
6		3	
7	E	1	0
8		6	
9	C	0	10
10	F	15	16
11	Q	0	12
12	S	0	0
13		0	
14	A	5	9
15	K	2	0
16	L	0	0
17		13	
18	D	0	0

Fig. 7-61

7.22 Suppose the following sequences list the nodes of a binary tree T in preorder and inorder, respectively:

Preorder:	G, B, Q, A, C, K, F, P, D, E, R, H
Inorder:	Q, B, K, C, F, A, G, P, E, D, H, R

Draw the diagram of the tree.

7.23 Suppose a binary tree T is in memory and an ITEM of information is given.

(*a*) Write a procedure which finds the location LOC of ITEM in T (assuming the elements of T are distinct).

(*b*) Write a procedure which finds the location LOC of ITEM and the location PAR of the parent of ITEM in T.

(*c*) Write a procedure which finds the number NUM of times ITEM appears in T (assuming the elements of T are not necessarily distinct).

Remark: T is not necessarily a binary search tree.

7.24 Suppose a binary tree T is in memory. Write a nonrecursive procedure for each of the following:

(*a*) Finding the number of nodes in T.

(*b*) Finding the depth D of T.

(*c*) Finding the number of terminal nodes in T.

7.25 Suppose a binary tree T is in memory. Write a procedure which deletes all the terminal nodes in T.

7.26 Suppose ROOTA points to a binary tree T_1 in memory. Write a procedure which makes a copy T_2 of the tree T_1 using ROOTB as a pointer.

BINARY SEARCH TREES

7.27 Suppose the following eight numbers are inserted in order into an empty binary search tree T:

50, 33, 44, 22, 77, 35, 60, 40

Draw the tree T.

7.28 Consider the binary search tree T in Fig. 7-62. Draw the tree T if each of the following operations is applied to the original tree T. (That is, the operations are applied independently, not successively.)

(*a*) Node 20 is added to T.
(*b*) Node 15 is added to T.
(*c*) Node 88 is added to T.
(*d*) Node 22 is deleted from T.
(*e*) Node 25 is deleted from T.
(*f*) Node 75 is deleted from T.

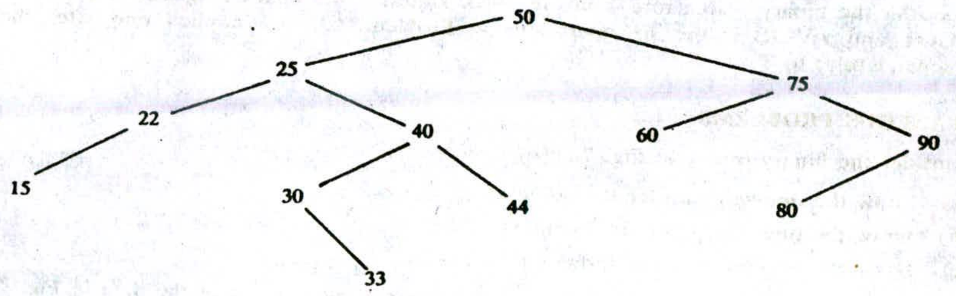


Fig. 7-62

7.29 Consider the binary search tree T in Fig. 7-62. Draw the final tree T if the six operations in Problem 7.28 are applied one after the other (not independently) to T.

7.30 Draw the binary search tree T in Fig. 7-63.

ROOT $\boxed{4}$

AVAIL $\boxed{3}$

	INFO	LEFT	RIGHT
1	Jones	7	0
2	Fox	11	1
3		8	
4	Murphy	2	15
5		13	
6	Thomas	0	0
7	Green	0	0
8		9	
9		10	
10		5	
11	Conroy	0	0
12	Parker	0	0
13		14	
14		0	
15	Rosen	12	6

Fig. 7-63

7.31 Consider the binary search tree T in Fig. 7-63. Describe the changes in INFO, LEFT, RIGHT, ROOT and AVAIL if each of the following operations is applied independently (not successively) to T.

(*a*) Davis is added to T.
(*b*) Harris is added to T.
(*c*) Smith is added to T.
(*d*) Parker is deleted from T.
(*e*) Fox is deleted from T.
(*f*) Murphy is deleted from T.

7.32 Consider the binary search tree T in Fig. 7-63. Describe the final changes in INFO, LEFT, RIGHT, ROOT and AVAIL if the six operations in Problem 7.31 are applied one after the other (not independently) to T.

MISCELLANEOUS PROBLEMS

7.33 Consider the binary tree T in Fig. 7-60(*a*).

(*a*) Draw the one-way inorder threading of T.
(*b*) Draw the one-way preorder threading of T.
(*c*) Draw the two-way inorder threading of T.

In each case, show how the threaded tree will appear in memory using the data in Fig. 7-60(*b*).

7.34 Consider the complete tree T with N = 10 nodes in Fig. 7-64. Suppose a maxheap is formed out of T by applying

Call INSHEAP(A, J, A[J + 1])

for J = 1, 2, . . . , N − 1. (Assume T is stored sequentially in the array A.) Find the final maxheap.

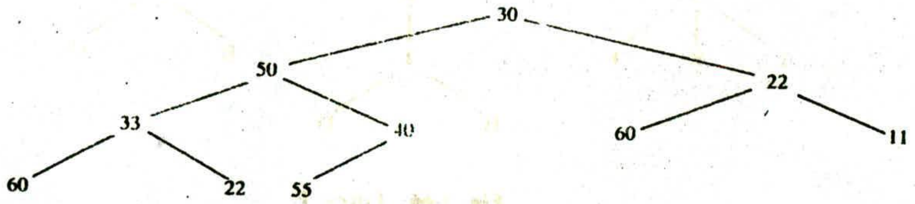


Fig. 7-64

7.35 Repeat Problem 7.34 for the tree T in Fig. 7-64, except now form a minheap out of T instead of a maxheap.

7.36 Draw the 2-tree corresponding to each of the following algebraic expressions:

(*a*) $E_1 = (a - 3b)(2x - y)^3$

(*b*) $E_2 = (2a + 5b)^3(x - 7y)^4$

7.37 Consider the 2-tree in Fig. 7-65. Find the Huffman coding for the seven letters determined by the tree T.

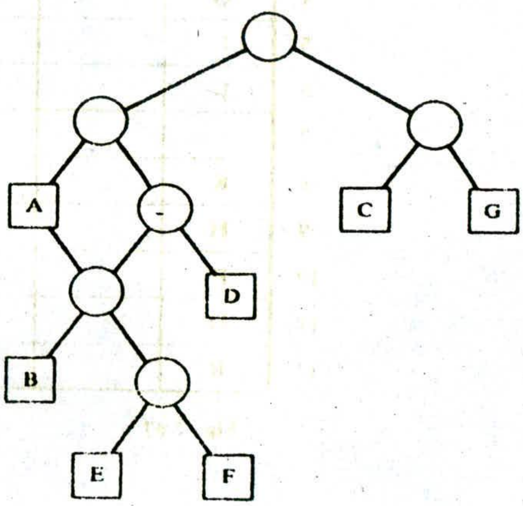


Fig. 7-65

7.38 Suppose the 7 data items A, B, . . . , G are assigned the following weights:

(A, 13), (B, 2), (C, 19), (D, 23), (E, 29), (F, 5), (G, 9)

Find the weighted path length *P* of the tree in Fig. 7-65.

7.39 Using the data in Problem 7.38, find a 2-tree with a minimum weighted path length *P*. What is the Huffman coding for the 7 letters using this new tree?

7.40 Consider the forest F in Fig.7-66, which consists of three trees with roots A, B and C, respectively.

(*a*) Find the binary tree F′ corresponding to the forest F.

(*b*) Fill in values for ROOT, CHILD and SIB in Fig. 7-67 so that F will be stored in memory.

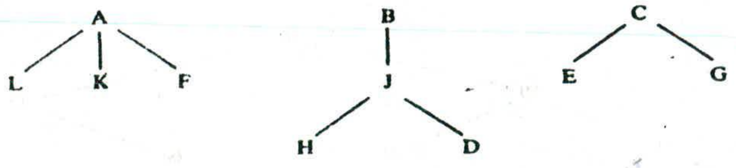


Fig. 7-66 Forest F

ROOT $\boxed{}$

	INFO	CHILD	SIB
1	A		
2	C		
3	E		
4	G		
5	J		
6	L		
7			
8	K		
9	H		
10	F		
11	D		
12	B		

Fig. 7-67

7.41 Suppose T is a complete tree with n nodes and depth D. Prove (*a*) $2^{D-1} - 1 < n \leq 2^D - 1$ and (*b*) $D \approx \log_2 n$.
Hint: Use the following identity with $x = 2$:

$$1 + x + x^2 + x^3 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

7.42 Suppose T is an extended binary tree. Prove:

(*a*) $N_E = N_I + 1$, where N_E is the number of external nodes and N_I is the number of internal nodes.

(*b*) $L_E = L_I + 2n$, where L_E is the external path length, L_I is the internal path lengtl and n is the number of internal nodes.

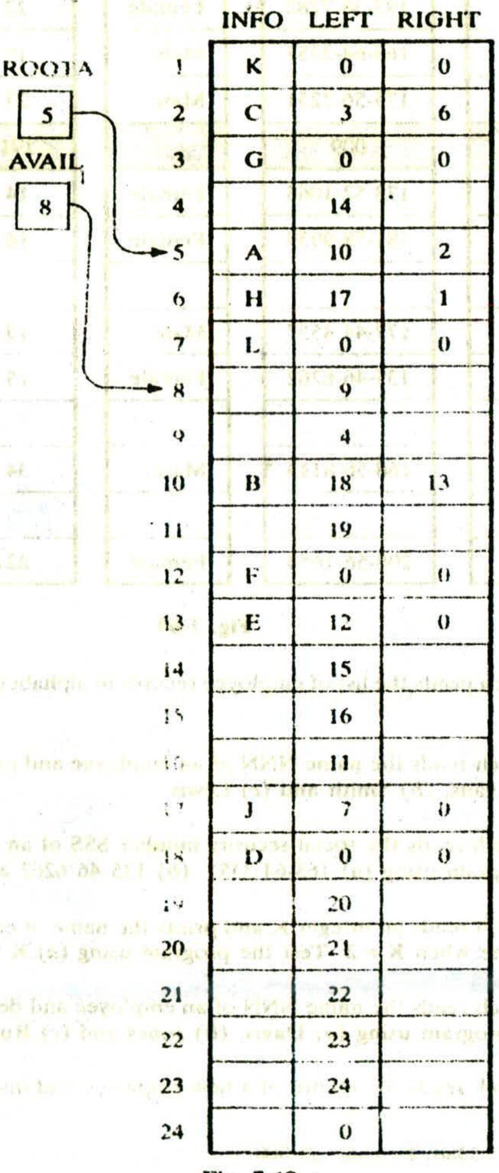
Programming Problems

Problems 7.43 to 7.45 refer to the tree *T* in Fig. 7-1, which is stored in memory as in Fig. 7-68.

7.43 Write a program which prints the nodes of *T* in (*a*) preorder, (*b*) inorder and (*c*) postorder.

7.44 Write a program which prints the terminal nodes of *T* in (*a*) preorder (*b*) inorder and (*c*) postorder. (*Note*: All three lists should be the same.)

7.45 Write a program which makes a copy *T'* of *T* using ROOTB as a pointer. Test the program by printing the nodes of *T'* in preorder and inorder and comparing the lists with those obtained in Prob. 7.43.



	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		21	
21		22	
22		23	
23		24	
24		0	

Fig. 7-68

7.46 Translate heapsort into a subprogram HEAPSORT(A, N) which sorts the array A with N elements. Test the program using

(*a*) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66 (*b*) D, A, T, A, S, T, R, U, C, T, U, R, E, S

Problems 7.47 to 7.52 refer to the list of employee records which are stored either as in Fig. 7-8 or as in Fig. 7-69. Each is a binary search tree with respect to the NAME key, but Fig. 7-69 uses a header node, which also acts as a sentinel. (Compare these problems with Probs. 5.41 to 5.46 in Chap. 5.)

HEAD: 5 → 5

AVAIL: 8 → 8

	NAME	SSN	SEX	SALARY	LEFT	RIGHT
1					0	
2	Davis	192-38-7282	Female	22 800	5	12
3	Kelly	165-64-3351	Male	19 000	5	5
4	Green	175-56-2251	Male	27 200	2	5
5	▨	009	▨	191 600	14	5
6	Brown	178-52-1065	Female	14 700	5	5
7	Lewis	181-58-9939	Female	16 400	3	10
8					11	
9	Cohen	177-44-4557	Male	19 000	6	4
10	Rubin	135-46-6262	Female	15 500	5	5
11					13	
12	Evans	168-56-8113	Male	34 200	5	5
13					1	
14	Harris	208-56-1654	Female	22 800	9	7

Fig. 7-69

7.47 Write a program which prints the list of employee records in alphabetical order. (*Hint*: Print the records in inorder.)

7.48 Write a program which reads the name NNN of an employee and prints the employee's record. Test the program using (*a*) Evans, (*b*) Smith and (*c*) Lewis.

7.49 Write a program which reads the social security number SSS of an employee and prints the employee's record. Test the program using (*a*) 165-64-3351, (*b*) 135-46-6262 and (*c*) 177-44-5555.

7.50 Write a program which reads an integer K and prints the name of each male employee when K = 1 or of each female employee when K = 2. Test the program using (*a*) K = 2, (*b*) K = 5 and (*c*) K = 1.

7.51 Write a program which reads the name NNN of an employee and deletes the employee's record from the structure. Test the program using (*a*) Davis, (*b*) Jones and (*c*) Rubin.

7.52 Write a program which reads the record of a new employee and inserts the record into the file. Test the program using:

(*a*) Fletcher; 168-52-3388; Female; 21 000

(*b*) Nelson; 175-32-2468, Male; 19 000

Chapter 8

Graphs and Their Applications

8.1 INTRODUCTION

This chapter investigates another nonlinear data structure: the *graph*. As we have done with other data structures, we discuss the representation of graphs in memory and present various operations and algorithms on them. In particular, we discuss the breadth-first search and the depth-first search of our graphs. Certain applications of graphs, including topological sorting, are also covered.

8.2 GRAPH THEORY TERMINOLOGY

This section summarizes some of the main terminology associated with the theory of graphs. Unfortunately, there is no standard terminology in graph theory. The reader is warned, therefore, that our definitions may be slightly different from the definitions used by other texts on data structures and graph theory.

Graphs and Multigraphs

A graph G consists of two things:

(1) A set V of elements called *nodes* (or *points* or *vertices*)

(2) A set E of *edges* such that each edge e in E is identified with a unique (unordered) pair $[u, v]$ of nodes in V, denoted by $e = [u, v]$

Sometimes we indicate the parts of a graph by writing $G = (V, E)$.

Suppose $e = [u, v]$. Then the nodes u and v are called the *endpoints* of e, and u and v are said to be *adjacent nodes* or *neighbors*. The *degree* of a node u, written $\deg(u)$, is the number of edges containing u. If $\deg(u) = 0$—that is, if u does not belong to any edge—then u is called an *isolated* node.

A *path* P of *length* n from a node u to a node v is defined as a sequence of $n + 1$ nodes.

$$P = (v_0, v_1, v_2, \ldots, v_n)$$

such that $u = v_0$; v_{i-1} is adjacent to v_i for $i = 1, 2, \ldots, n$; and $v_n = v$. The path P is said to be *closed* if $v_0 = v_n$. The path P is said to be *simple* if all the nodes are distinct, with the exception that v_0 may equal v_n; that is, P is simple if the nodes $v_0, v_1, \ldots, v_{n-1}$ are distinct and the nodes $v_1, v_2, \ldots, v_n$ are distinct. A *cycle* is a closed simple path with length 3 or more. A cycle of length k is called a *k-cycle*.

A graph G is said to be *connected* if there is a path between any two of its nodes. We will show (in Prob. 8.18) that if there is a path P from a node u to a node v, then, by eliminating unnecessary edges, one can obtain a simple path Q from u to v; accordingly, we can state the following proposition.

Proposition 8.1: A graph G is connected if and only if there is a simple path between any two nodes in G.

A graph G is said to be *complete* if every node u in G is adjacent to every other node v in G. Clearly such a graph is connected. A complete graph with n nodes will have $n(n-1)/2$ edges.

A connected graph T without any cycles is called a *tree graph* or *free tree* or, simply, a *tree*. This means, in particular, that there is a unique simple path P between any two nodes u and v in T (Prob. 8.18). Furthermore, if T is a finite tree with m nodes, then T will have $m - 1$ edges (Prob. 8.20).

A graph G is said to be *labeled* if its edges are assigned data. In particular, G is said to be *weighted* each edge e in G is assigned a nonnegative numerical value $w(e)$ called the *weight* or *length* of e. In such a case, each path P in G is assigned a *weight* or *length* which is the sum of the weights of the edges along the path P. If we are given no other information about weights, we may assume any graph G to be weighted by assigning the weight $w(e) = 1$ to each edge e in G.

The definition of a graph may be generalized by permitting the following:

(1) *Multiple edges.* Distinct edges e and e' are called *multiple edges* if they connect the same endpoints, that is, if $e = [u, v]$ and $e' = [u, v]$.

(2) *Loops.* An edge e is called a *loop* if it has identical endpoints, that is, if $e = [u, u]$.

Such a generalization M is called a *multigraph*. In other words, the definition of a graph usually does not allow either multiple edges or loops.

A multigraph M is said to be *finite* if it has a finite number of nodes and a finite number of edges Observe that a graph G with a finite number of nodes must automatically have a finite number of edges and so must be finite; but this is not necessarily true for a multigraph M, since M may have multiple edges. Unless otherwise specified, graphs and multigraphs in this text shall be finite.

EXAMPLE 8.1

(*a*) Figure 8-1(*a*) is a picture of a connected graph with 5 nodes—A, B, C, D and E—and 7 edges:

$$[A, B], \quad [B, C], \quad [C, D], \quad [D, E], \quad [A, E], \quad [C, E] \quad [A, C]$$

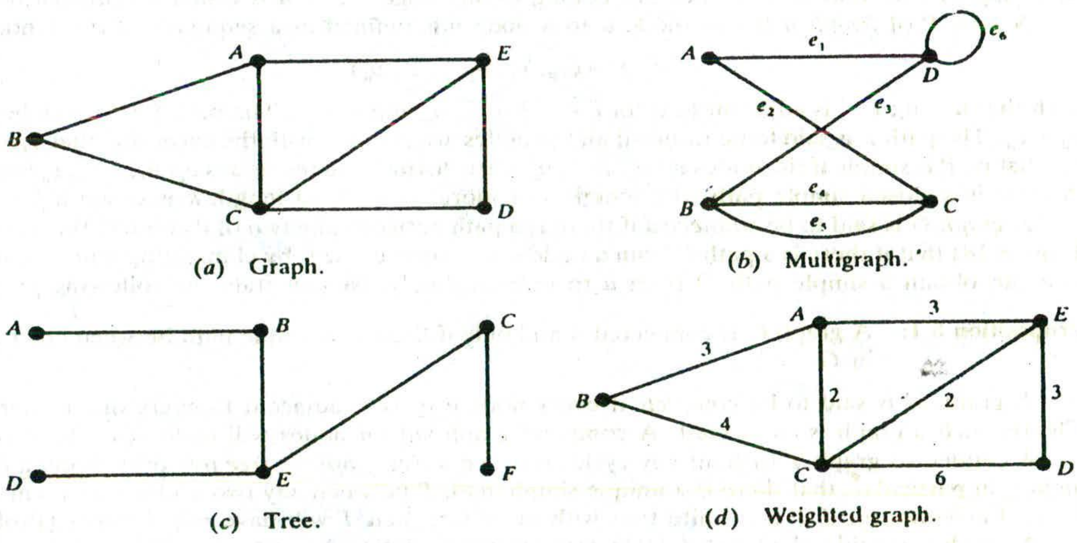
There are two simple paths of length 2 from B to E: (B, A, E) and (B, C, E). There is only one simple path of length 2 from B to D: (B, C, D). We note that (B, A, D) is not a path, since $[A, D]$ is not an edge. There are two 4-cycles in the graph:

$$[A, B, C, E, A] \quad \text{and} \quad [A, C, D, E, A].$$

Note that $\deg(A) = 3$, since A belongs to 3 edges. Similarly, $\deg(C) = 4$ and $\deg(D) = 2$.

(*b*) Figure 8-1(*b*) is not a graph but a multigraph. The reason is that it has multiple edges—$e_4 = [B, C]$ and $e_5 = [B, C]$—and it has a loop, $e_6 = [D, D]$. The definition of a graph usually does not allow either multiple edges or loops.

(c) Figure 8-1(*c*) is a tree graph with $m = 6$ nodes and, consequently, $m - 1 = 5$ edges. The reader can verify that there is a unique simple path between any two nodes of the tree graph.



(*a*) Graph. (*b*) Multigraph. (*c*) Tree. (*d*) Weighted graph.

Fig. 8-1

(*d*) Figure 8-1(*d*) is the same graph as in Fig. 8-1(*a*), except that now the graph is weighted. Observe that $P_1 = (B, C, D)$ and $P_2 = (B, A, E, D)$ are both paths from node B to node D. Although P_2 contains more edges than P_1, the weight $w(P_2) = 9$ is less than the weight $w(P_1) = 10$.

Directed Graphs

A *directed graph G*, also called a *digraph* or *graph*, is the same as a multigraph except that each edge e in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u, v) of nodes in G rather than an unordered pair $[u, v]$.

Suppose G is a directed graph with a directed edge $e = (u, v)$. Then e is also called an *arc*. Moreover, the following terminology is used:

(1) e *begins* at u and *ends* at v.

(2) u is the *origin* or *initial point* of e, and v is the *destination* or *terminal point* of e.

(3) u is a *predecessor* of v, and v is a *successor* or *neighbor* of u.

(4) u is *adjacent to* v, and v is *adjacent to* u.

The *outdegree* of a node u in G, written outdeg(u), is the number of edges beginning at u. Similarly, the *indegree* of u, written indeg(u), is the number of edges ending at u. A node u is called a *source* if it has a positive outdegree but zero indegree. Similarly, u is called a *sink* if it has a zero outdegree but a positive indegree.

The notions of *path*, *simple path* and *cycle* carry over from undirected graphs to directed graphs except that now the direction of each edge in a path (cycle) must agree with the direction of the path (cycle). A node v is said to be *reachable* from a node u if there is a (directed) path from u to v.

A directed graph G is said to be *connected*, or *strongly connected*, if for each pair u, v of nodes in G there is a path from u to v and there is also a path from v to u. On the other hand, G is said to be *unilaterally connected* if for any pair u, v of nodes in G there is a path from u to v or a path from v to u.

EXAMPLE 8.2

Figure 8-2 shows a directed graph G with 4 nodes and 7 (directed) edges. The edges e_2 and e_3 are said to be *parallel*, since each begins at B and ends at A. The edge e_7 is a *loop*, since it begins and ends at the same point, B. The sequence $P_1 = (D, C, B, A)$ is not a path, since (C, B) is not an edge—that is, the direction of the edge $e_5 = (B, C)$ does not agree with the direction of the path P_1. On the other hand, $P_2 = (D, B, A)$ is a path from D to A, since (D, B) and (B, A) are edges. Thus A is reachable from D. There is no path from C to any other node, so G is not strongly connected. However, G is unilaterally connected. Note that indeg(D) = 1 and outdeg(D) = 2. Node C is a sink, since indeg(C) = 2 but outdeg(C) = 0. No node in G is a source.

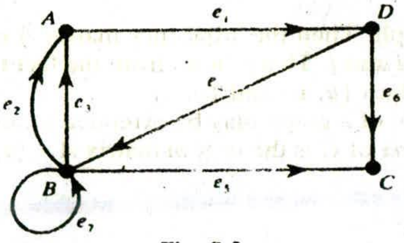


Fig. 8-2

Let T be any nonempty tree graph. Suppose we choose any node R in T. Then T with this designated node R is called a *rooted tree* and R is called its *root*. Recall that there is a unique simple path from the root R to any other node in T. This defines a direction to the edges in T, so the rooted tree T may be viewed as a directed graph. Furthermore, suppose we also order the successors of each node v in T. Then T is called an *ordered rooted tree*. Ordered rooted trees are nothing more than the general trees discussed in Chap. 7.

A directed graph G is said to be *simple* if G has no parallel edges. A simple graph G may have loops, but it cannot have more than one loop at a given node. A nondirected graph G may be viewed as a simple directed graph by assuming that each edge $[u, v]$ in G represents two directed edges, (u, v) and (v, u). (Observe that we use the notation $[u, v]$ to denote an unordered pair and the notation (u, v) to denote an ordered pair.)

Warning: The main subject matter of this chapter is simple directed graphs. Accordingly, unless otherwise stated or implied, the term "graph" shall mean simple directed graph, and the term "edge" shall mean directed edge.

8.3 SEQUENTIAL REPRESENTATION OF GRAPHS; ADJACENCY MATRIX; PATH MATRIX

There are two standard ways of maintaining a graph G in the memory of a computer. One way, called the *sequential representation* of G, is by means of its adjacency matrix A. The other way, called the *linked representation* of G, is by means of linked lists of neighbors. This section covers the first representation, and shows how the adjacency matrix A of G can be used to easily answer certain questions of connectivity in G. The linked representation of G will be covered in Sec. 8.5.

Regardless of the way one maintains a graph G in the memory of the computer, the graph G is normally input into the computer by using its formal definition: a collection of nodes and a collection of edges.

Adjacency Matrix

Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, \ldots, v_m$. Then the *adjacency matrix* $A = (a_{ij})$ of the graph G is the $m \times m$ matrix defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j\text{, that is, if there is an edge } (v_i, v_j) \\ 0 & \text{otherwise} \end{cases}$$

Such a matrix A, which contains entries of only 0 and 1, is called a *bit matrix* or a *Boolean matrix*.

The adjacency matrix A of the graph G does depend on the ordering of the nodes of G; that is, a different ordering of the nodes may result in a different adjacency matrix. However, the matrices resulting from two different orderings are closely related in that one can be obtained from the other by simply interchanging rows and columns. Unless otherwise stated, we will assume that the nodes of our graph G have a fixed ordering.

Suppose G is an undirected graph. Then the adjacency matrix A of G will be a *symmetric matrix*, i.e., one in which $a_{ij} = a_{ji}$ for every i and j. This follows from the fact that each undirected edge $[u, v]$ corresponds to the two directed edges (u, v) and (v, u).

The above matrix representation of a graph may be extended to multigraphs. Specifically, if G is a multigraph, then the *adjacency matrix* of G is the $m \times m$ matrix $A = (a_{ij})$ defined by setting a_{ij} equal to the number of edges from v_i to v_j.

EXAMPLE 8.3

Consider the graph G in Fig. 8-3. Suppose the nodes are stored in memory in a linear array DATA as follows:

DATA: X, Y, Z, W

Then we assume that the ordering of the nodes in G is as follows: $v_1 = X$, $v_2 = Y$, $v_3 = Z$ and $v_4 = W$. The adjacency matrix A of G is as follows:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Note that the number of 1's in A is equal to the number of edges in G.

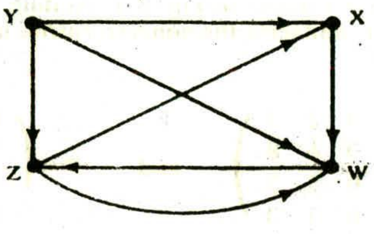


Fig. 8-3

Consider the powers $A, A^2, A^3, \ldots$ of the adjacency matrix A of a graph G. Let

$$a_K(i, j) = \text{the } ij \text{ entry in the matrix } A^K$$

Observe that $a_1(i, j) = a_{ij}$ gives the number of paths of length 1 from node v_i to node v_j. One can show that $a_2(i, j)$ gives the number of paths of length 2 from v_i to v_j. In fact, we prove in Prob. 8.19 the following general result.

Proposition 8.2: Let A be the adjacency matrix of a graph G. Then $a_K(i, j)$, the ij entry in the matrix A^K, gives the number of paths of length K from v_i to v_j.

Consider again the graph G in Fig. 8-3, whose adjacency matrix A is given in Example 8.3. The powers A^2, A^3 and A^4 of the matrix A follow:

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad A^4 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$ #

Accordingly, in particular, there is a path of length 2 from v_4 to v_1, there are two paths of length 3 from v_2 to v_3, and there are three paths of length 4 from v_2 to v_4. (Here, $v_1 = X$, $v_2 = Y$, $v_3 = Z$ and $v_4 = W$.)

Suppose we now define the matrix B_r as follows:

$$B_r = A + A^2 + A^3 + \cdots + A^r$$

Then the ij entry of the matrix B_r gives the number of paths of length r or less from node v_i to v_j.

Path Matrix

Let G be a simple directed graph with m nodes, $v_1, v_2, \ldots, v_m$. The *path matrix* or *reachability matrix* of G is the m-square matrix $P = (p_{ij})$ defined as follows:

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Suppose there is a path from v_i to v_j. Then there must be a simple path from v_i to v_j when $v_i \neq v_j$, or there must be a cycle from v_i to v_j when $v_i = v_j$. Since G has only m nodes, such a simple path must have length $m - 1$ or less, or such a cycle must have length m or less. This means that there is a nonzero ij

entry in the matrix B_m, defined at the end of the preceding subsection. Accordingly, we have the following relationship between the path matrix P and the adjacency matrix A.

Proposition 8.3: Let A be the adjacency matrix and let $P = (p_{ij})$ be the path matrix of a digraph G. Then $p_{ij} = 1$ if and only if there is a nonzero number in the ij entry of the matrix

$$B_m = A + A^2 + A^3 + \cdots + A^m$$

Consider the graph G with $m = 4$ nodes in Fig. 8-3. Adding the matrices A, A^2, A^3 and A^4, we obtain the following matrix B_4, and, replacing the nonzero entries in B_4 by 1, we obtain the path matrix P of the graph G:

$$B_4 = \begin{pmatrix} 1 & 0 & 2 & 3 \\ 5 & 0 & 6 & 8 \\ 3 & 0 & 3 & 5 \\ 2 & 0 & 3 & 3 \end{pmatrix} \quad \text{and} \quad P = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Examining the matrix P, we see that the node v_2 is not reachable from any of the other nodes.

Recall that a directed graph G is said to be *strongly connected* if, for any pair of nodes u and v in G, there are both a path from u to v and a path from v to u. Accordingly, G is strongly connected if and only if the path matrix P of G has no zero entries. Thus the graph G in Fig. 8-3 is not strongly connected.

The *transitive closure* of a graph G is defined to be the graph G' such that G ..as the same nodes as G and there is an edge (v_i, v_j) in G' whenever there is a path from v_i to v_j in G. Accordingly, the path matrix P of the graph G is precisely the adjacency matrix of its transitive closure G'. Furthermore, a graph G is strongly connected if and only if its transitive closure is a complete graph.

Remark: The adjacency matrix A and the path matrix P of a graph G may be viewed as logical (Boolean) matrices, where 0 represents "false" and 1 represents "true." Thus, the logical operations of $\wedge$(AND) and $\vee$(OR) may be applied to the entries of A and P. The values of $\wedge$ and $\vee$ appear in Fig. 8-4. These operations will be used in the next section.

$\wedge$	0	1
0	0	0
1	0	1

(*a*) AND.

$\vee$	0	1
0	0	1
1	1	1

(*b*) OR.

Fig. 8-4

8.4 WARSHALL'S ALGORITHM; SHORTEST PATHS

Let G be a directed graph with m nodes, $v_1, v_2, \ldots, v_m$. Suppose we want to find the path matrix P of the graph G. Warshall gave an algorithm for this purpose that is much more efficient than calculating the powers of the adjacency matrix A and using Proposition 8.3. This algorithm is described in this section, and a similar algorithm is used to find shortest paths in G when G is weighted.

First we define m-square Boolean matrices $P_0, P_1, \ldots, P_m$ as follows. Let $P_k[i, j]$ denote the ij entry of the matrix P_k. Then we define:

$$P_k[i, j] = \begin{cases} 1 & \text{if there is a simple path from } v_i \text{ to } v_j \\ & \text{which does not use any other nodes} \\ & \text{except possibly } v_1, v_2, \ldots, v_k \\ 0 & \text{otherwise} \end{cases}$$

In other words,

$P_0[i, j] = 1$	if there is an edge from v_i to v_j
$P_1[i, j] = 1$	if there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1
$P_2[i, j] = 1$	if there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1 and v_2

. . . .

First observe that the matrix $P_0 = A$, the adjacency matrix of G. Furthermore, since G has only m nodes, the last matrix $P_m = P$, the path matrix of G.

Warshall observed that $P_k[i, j] = 1$ can occur only if one of the following two cases occurs:

(1) There is a simple path from v_i to v_j which does not use any other nodes except possibly $v_1, v_2, \ldots, v_{k-1}$; hence

$$P_{k-1}[i, j] = 1$$

(2) There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except possibly $v_1, v_2, \ldots, v_{k-1}$; hence

$$P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

These two cases are pictured, respectively, in Fig. 8-5(a) and (b), where

$$\longrightarrow \cdots \longrightarrow$$

denotes part of a simple path which does not use any nodes except possibly $v_1, v_2, \ldots, v_{k-1}$.

$$v_i \longrightarrow \cdots \longrightarrow v_j \qquad\qquad v_i \longrightarrow \cdots \longrightarrow v_k \longrightarrow \cdots \longrightarrow v_j$$

(a) (b)

Fig. 8-5

Accordingly, the elements of the matrix P_k can be obtained by

$$P_k[i, j] = P_{k-1}[i, j] \vee (P_{k-1}[i, k] \wedge P_{k-1}[k, j])$$

where we use the logical operations of $\wedge$(AND) and $\vee$(OR). In other words we can obtain each entry in the matrix P_k by looking at only three entries in the matrix P_{k-1}. Warshall's algorithm follows.

Algorithm 8.1: (Warshall's Algorithm) A directed graph G with M nodes is maintained in memory by its adjacency matrix A. This algorithm finds the (Boolean) path matrix P of the graph G.

```
1. Repeat for I, J = 1, 2, . . . , M: [Initializes P.]
       If A[I, J] = 0, then: Set P[I, J] := 0;
       Else: Set P[I, J] := 1.
   [End of loop.]
2. Repeat Steps 3 and 4 for K = 1, 2, . . . , M: [Updates P.]
3.     Repeat Step 4 for I = 1, 2, . . . , M:
4.         Repeat for J = 1, 2, . . . , M:
               Set P[I, J] := P[I, J] ∨ (P[I, K] ∧ P[K, J]).
           [End of loop.]
       [End of Step 3 loop.]
   [End of Step 2 loop.]
5. Exit.
```

Shortest-Path Algorithm

Let G be a directed graph with m nodes, $v_1, v_2, \ldots, v_m$. Suppose G is *weighted*; that is, suppose each edge e in G is assigned a nonnegative number $w(e)$ called the *weight* or *length* of the edge e. Then G may be maintained in memory by its *weight matrix* $W = (w_{ij})$, defined as follows:

$$w_{ij} = \begin{cases} w(e) & \text{if there is an edge } e \text{ from } v_i \text{ to } v_j \\ 0 & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

The path matrix P tells us whether or not there are paths between the nodes. Now we want to find a matrix Q which will tell us the lengths of the shortest paths between the nodes or, more exactly, a matrix $Q = (q_{ij})$ where

$$q_{ij} = \text{length of a shortest path from } v_i \text{ to } v_j$$

Next we describe a modification of Warshall's algorithm which finds us the matrix Q.

Here we define a sequence of matrices $Q_0, Q_1, \ldots, Q_m$ (analogous to the above matrices $P_0, P_1, \ldots, P_m$) whose entries are defined as follows:

$Q_k[i, j]$ = the smaller of the length of the preceding path from v_i to v_j or the sum of the lengths of the preceding paths from v_i to v_k and from v_k to v_j

More exactly,

$$Q_k[i, j] = \text{MIN}(Q_{k-1}[i, j], \quad Q_{k-1}[i, k] + Q_{k-1}[k, j])$$

The initial matrix Q_0 is the same as the weight matrix W except that each 0 in W is replaced by ∞ (or a very, very large number). The final matrix Q_m will be the desired matrix Q.

EXAMPLE 8.4

Consider the weighted graph G in Fig. 8-6. Assume $v_1 = \text{R}$, $v_2 = \text{S}$, $v_3 = \text{T}$ and $v_4 = \text{U}$. Then the weight matrix W of G is as follows:

$$W = \begin{pmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{pmatrix}$$

Applying the modified Warshall's algorithm, we obtain the following matrices Q_0, Q_1, Q_2, Q_3 and $Q_4 = Q$. To the right of each matrix Q_k, we show the matrix of paths which correspond to the lengths in the matrix Q_k.

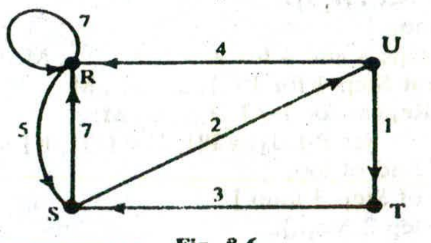


Fig. 8-6

$$Q_0 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{pmatrix} \qquad \begin{pmatrix} RR & RS & - & - \\ SR & - & - & SU \\ - & TS & - & - \\ UR & - & UT & - \end{pmatrix}$$

$$Q_1 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & 12 & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \textcircled{9} & 1 & \infty \end{pmatrix} \qquad \begin{pmatrix} RR & RS & - & - \\ SR & SRS & - & SU \\ - & TS & - & - \\ UR & URS & UT & - \end{pmatrix}$$

$$Q_2 = \begin{pmatrix} 7 & 5 & \textcircled{∞} & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & 9 & 1 & 11 \end{pmatrix} \qquad \begin{pmatrix} RR & RS & - & RSU \\ SR & SRS & - & SU \\ TSR & TS & - & TSU \\ UR & URS & UT & URS \end{pmatrix}$$

$$Q_3 = \begin{pmatrix} 7 & 5 & \infty & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & \textcircled{4} & 1 & 6 \end{pmatrix} \qquad \begin{pmatrix} RR & RS & - & RSU \\ SR & SRS & - & SU \\ TSR & TS & - & TSU \\ UR & UTS & UT & UTSU \end{pmatrix}$$

$$Q_4 = \begin{pmatrix} 7 & 5 & 8 & 7 \\ 7 & 11 & 3 & 2 \\ \textcircled{9} & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{pmatrix} \qquad \begin{pmatrix} RR & RS & RSUT & RSU \\ SR & SURS & SUT & SU \\ TSUR & TS & TSUT & TSU \\ UR & UTS & UT & UTSU \end{pmatrix}$$

We indicate how the circled entries are obtained:

$$Q_1[4, 2] = \text{MIN}(Q_0[4, 2],\ Q_0[4, 1] + Q_0[1, 2]) = \text{MIN}(\infty, 4 + 5) = 9$$
$$Q_2[1, 3] = \text{MIN}(Q_1[1, 3],\ Q_1[1, 2] + Q_1[2, 3]) = \text{MIN}(\infty, 5 + \infty) = \infty$$
$$Q_3[4, 2] = \text{MIN}(Q_2[4, 2],\ Q_2[4, 3] + Q_2[3, 2]) = \text{MIN}(9, 3 + 1) = 4$$
$$Q_4[3, 1] = \text{MIN}(Q_3[3, 1],\ Q_3[3, 4] + Q_3[4, 1]) = \text{MIN}(10, 5 + 4) = 9$$

The formal statement of the algorithm follows.

Algorithm 8.2: **(Shortest-Path Algorithm) A weighted graph G with M nodes is maintained in memory by its weight matrix W. This algorithm finds a matrix Q such that Q[I, J] is the length of a shortest path from node V_I to node V_J. INFINITY is a very large number, and MIN is the minimum value function.**

```
1. Repeat for I, J = 1, 2, . . . , M: [Initializes Q.]
      If W[I, J] = 0, then: Set Q[I, J] := INFINITY;
      Else: Set Q[I, J] := W[I, J].
   [End of loop.]
2. Repeat Steps 3 and 4 for K = 1, 2, . . . , M: [Updates Q.]
3.    Repeat Step 4 for I = 1, 2, . . . , M:
4.       Repeat for J = 1, 2, . . . , M:
            Set Q[I, J] := MIN(Q[I, J], Q[I, K] + Q[K, J]).
         [End of loop.]
      [End of Step 3 loop.]
   [End of Step 2 loop.]
5. Exit.
```

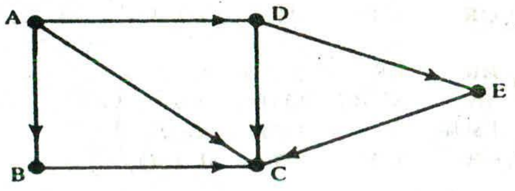
Observe the similarity between Algorithm 8.1 and Algorithm 8.2.

Algorithm 8.2 can also be used for a graph G without weights by simply assigning the weight $w(e) = 1$ to each edge e in G.

8.5 LINKED REPRESENTATION OF A GRAPH

Let G be a directed graph with m nodes. The sequential representation of G in memory—i.e., the representation of G by its adjacency matrix A—has a number of major drawbacks. First of all, it may be difficult to insert and delete nodes in G. This is because the size of A may need to be changed and the nodes may need to be reordered, so there may be many, many changes in the matrix A. Furthermore, if the number of edges is $O(m)$ or $O(m \log_2 m)$, then the matrix A will be sparse (will contain many zeros); hence a great deal of space will be wasted. Accordingly, G is usually represented in memory by a linked *representation*, also called an *adjacency structure*, which is described in this section.

Consider the graph G in Fig. 8-7(a). The table in Fig. 8-7(b) shows each node in G followed by its *adjacency list*, which is its list of adjacent nodes, also called its *successors* or *neighbors*. Figure 8-8 shows a schematic diagram of a linked representation of G in memory. Specifically, the linked representation will contain two lists (or files), a node list NODE and an edge list EDGE, as follows.



(a) Graph G.

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

(b) Adjacency lists of G.

Fig. 8-7

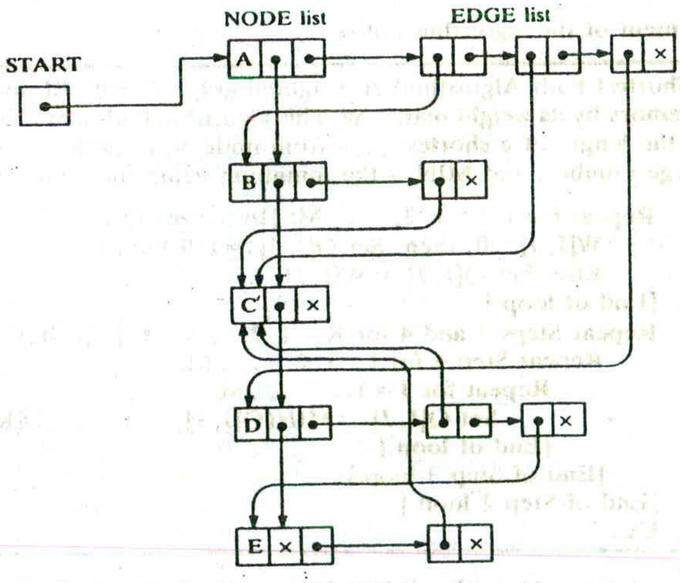


Fig. 8-8

(*a*) *Node list.* Each element in the list NODE will correspond to a node in *G*, and it will be a record of the form:

NODE	NEXT	ADJ	▨

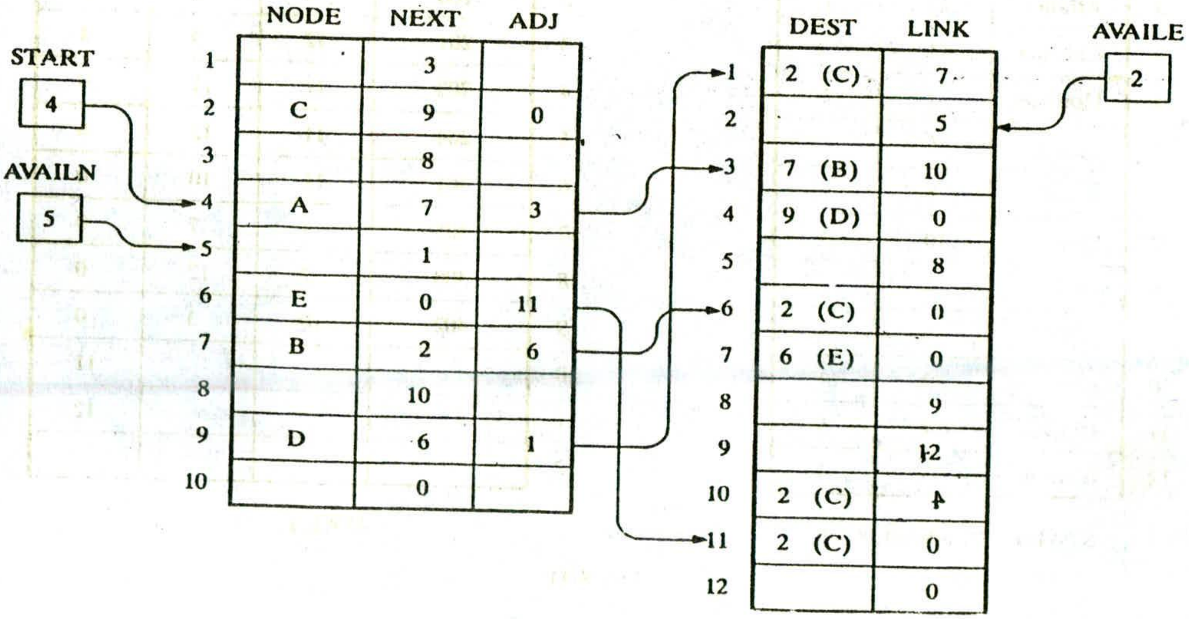
Here NODE will be the name or key value of the node, NEXT will be a pointer to the next node in the list NODE and ADJ will be a pointer to the first element in the adjacency list of the node, which is maintained in the list EDGE. The shaded area indicates that there may be other information in the record, such as the indegree INDEG of the node, the outdegree OUTDEG of the node, the STATUS of the node during the execution of an algorithm, and so on. (Alternatively, one may assume that NODE is an array of records containing fields such as NAME, INDEG, OUTDEG, STATUS, ...) The nodes themselves, as pictured in Fig. 8-7, will be organized as a linked list and hence will have a pointer variable START for the beginning of the list and a pointer variable AVAILN for the list of available space. Sometimes, depending on the application, the nodes may be organized as a sorted array or a binary search tree instead of a linked list.

(*b*) *Edge list.* Each element in the list EDGE will correspond to an edge of *G* and will be a record of the form:

DEST	LINK	▨

The field DEST will point to the location in the list NODE of the destination or terminal node of the edge. The field LINK will link together the edges with the same initial node, that is, the nodes in the same adjacency list. The shaded area indicates that there may be other information in the record corresponding to the edge, such as a field EDGE containing the labeled data of the edge when *G* is a labeled graph, a field WEIGHT containing the weight of the edge when *G* is a weighted graph, and so on. We also need a pointer variable AVAILE for the list of available space in the list EDGE.

Figure 8-9 shows how the graph *G* in Fig. 8-7(*a*) may appear in memory. The choice of 10 locations for the list NODE and 12 locations for the list EDGE is arbitrary.



	NODE	NEXT	ADJ
1		3	
2	C	9	0
3		8	
4	A	7	3
5		1	
6	E	0	11
7	B	2	6
8		10	
9	D	6	1
10		0	

	DEST	LINK
1	2 (C)	7
2		5
3	7 (B)	10
4	9 (D)	0
5		8
6	2 (C)	0
7	6 (E)	0
8		9
9		12
10	2 (C)	4
11	2 (C)	0
12		0

The linked representation of a graph G that we have been discussing may be denoted by

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

The representation may also include an array WEIGHT when G is weighted or may include an array EDGE when G is a labeled graph.

EXAMPLE 8.5

Suppose Friendly Airways has nine daily flights, as follows:

103	Atlanta to Houston	203	Boston to Denver	305	Chicago to Miami
106	Houston to Atlanta	204	Denver to Boston	308	Miami to Boston
201	Boston to Chicago	301	Denver to Reno	402	Reno to Chicago

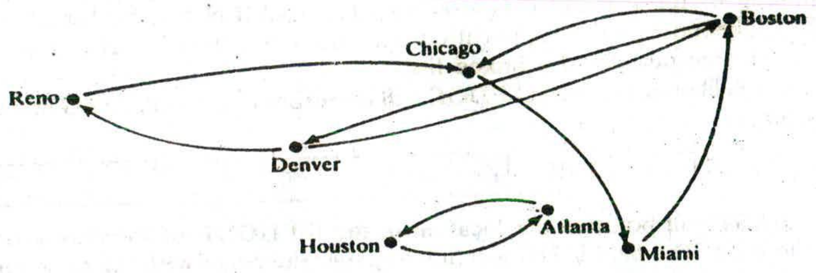


Fig. 8-10

NODE list

	CITY	NEXT	ADJ
1		0	
2	Atlanta	12	1
3	Chicago	11	7
4	Houston	7	2
5		6	
6		8	
7	Miami	10	8
8		9	
9		1	
10	Reno	0	9
11	Denver	4	5
12	Boston	3	3

START = 2, AVAILN = 5

EDGE list

	NUMBER	ORIG	DEST	LINK
1	103	2	4	0
2	106	4	2	0
3	201	12	3	4
4	203	12	11	0
5	204	11	12	6
6	301	11	10	0
7	305	3	7	0
8	308	7	12	0
9	402	10	3	0
10				11
11				12
12				0

AVAILE = 10

Fig. 8-11

Clearly, the data may be stored efficiently in a file where each record contains three fields:

Flight Number, City of Origin, City of Destination

However, such a representation does not easily answer the following natural questions:

(*a*) Is there a direct flight from city X to city Y?

(*b*) Can one fly, with possible stops, from city X to city Y?

(*c*) What is the most direct route, i.e., the route with the smallest number of stops, from city X to city Y?

To make the answers to these questions more readily available, it may be very useful for the data to be organized also as a graph *G* with the cities as nodes and with the flights as edges. Figure 8-10 is a picture of the graph *G*.

Figure 8-11 shows how the graph *G* may appear in memory using the linked representation. We note that *G* is a labeled graph, not a weighted graph, since the flight number is simply for identification. Even though the data are organized as a graph, one still would require some type of algorithm to answer questions (*b*) and (*c*). Such algorithms are discussed later in the chapter.

8.6 OPERATIONS ON GRAPHS

Suppose a graph *G* is maintained in memory by the linked representation

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

as discussed in the preceding section. This section discusses the operations of searching, inserting and deleting nodes and edges in the graph *G*. The operation of traversing is treated in the next section.

The operations in this section use certain procedures from Chap. 5, on linked lists. For completeness, we restate these procedures below, but in a slightly different manner than in Chap. 5. Naturally, if a circular linked list or a binary search tree is used instead of a linked list, then the analogous procedures must be used.

Procedure 8.3 (originally Algorithm 5.2) finds the location LOC of an ITEM in a linked list.

Procedure 8.4 (originally Procedure 5.9 and Algorithm 5.10) deletes a given ITEM from a linked list. Here we use a logical variable FLAG to tell whether or not ITEM originally appears in the linked list.

Searching in a Graph

Suppose we want to find the location LOC of a node N in a graph *G*. This can be accomplished by using Procedure 8.3, as follows:

Call FIND(NODE, NEXT, START, N, LOC)

That is, this Call statement searches the list NODE for the node N.

On the other hand, suppose we want to find the location LOC of an edge (A, B) in the graph *G*. First we must find the location LOCA of A and the location LOCB of B in the list NODE. Then we must find in the list of successors of A, which has the list pointer ADJ[LOCA], the location LOC of LOCB. This is implemented by Procedure 8.5, which also checks to see whether A and B are nodes in *G*. Observe that LOC gives the location of LOCB in the list EDGE.

Inserting in a Graph

Suppose a node N is to be inserted in the graph *G*. Note that N will be assigned to NODE[AVAILN], the first available node. Moreover, since N will be an isolated node, one must also set ADJ[AVAILN] := NULL. Procedure 8.6 accomplishes this task using a logical variable FLAG to indicate overflow.

Clearly, Procedure 8.6 must be modified if the list NODE is maintained as a sorted list or a binary search tree.

```
Procedure 8.3:  FIND(INFO, LINK START, ITEM, LOC) [Algorithm 5.2,
                Finds the location LOC of the first node containing ITEM, or sets
                LOC := NULL.

                1. Set PTR := START.
                2. Repeat while PTR ≠ NULL:
                       If ITEM = INFO[PTR], then: Set LOC := PTR, and Return.
                       Else: Set PTR := LINK[PTR].
                   [End of loop.]
                3. Set LOC := NULL, and Return.
```

```
Procedure 8.4:  DELETE(INFO, LINK, START, AVAIL, ITEM, FLAG) [Algorithm 5.10]
                Deletes the first node in the list containing ITEM, or sets FLAG := FALSE
                when ITEM does not appear in the list.

                1. [List empty?] If START = NULL, then: Set FLAG := FALSE, and Return.
                2. [ITEM in first node?] If INFO[START] = ITEM, then:
                       Set PTR := START, START := LINK[START],
                       LINK[PTR] := AVAIL, AVAIL := PTR,
                       FLAG := TRUE, and Return.
                   [End of If structure.]
                3. Set PTR := LINK[START] and SAVE := START. [Initializes pointers.]
                4. Repeat Steps 5 and 6 while PTR ≠ NULL:
                5.     If INFO[PTR] = ITEM, then:
                           Set LINK[SAVE] := LINK[PTR], LINK[PTR] := AVAIL,
                           AVAIL := PTR, FLAG := TRUE, and Return.
                       [End of If structure.]
                6.     Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers]
                   [End of Step 4 loop.]
                7. Set FLAG := FALSE, and Return.
```

```
Procedure 8.5:  FINDEDGE(NODE, NEXT, ADJ, START, DEST, LINK, A, B, LOC)
                This procedure finds the location LOC of an edge (A, B) in the graph G, or sets
                LOC := NULL.

                1. Call FIND(NODE, NEXT, START, A, LOCA).
                2. CALL FIND(NODE, NEXT, START, B, LOCB).
                3. If LOCA = NULL or LOCB = NULL, then: Set LOC := NULL.
                   Else: Call FIND(DEST, LINK, ADJ[LOCA], LOCB, LOC).
                4. Return.
```

```
Procedure 8.6:  INSNODE(NODE, NEXT, ADJ, START, AVAILN, N, FLAG)
                This procedure inserts the node N in the graph G.

                1. [OVERFLOW?] If AVAILN = NULL, then: Set FLAG := FALSE, and
                   Return.
                2. Set ADJ[AVAILN] := NULL.
                3. [Removes node from AVAILN list.]
                   Set NEW := AVAILN and AVAILN := NEXT[AVAILN].
                4. [Inserts node N in the NODE list.]
                   Set NODE[NEW] := N, NEXT[NEW] := START and START := NEW.
                5. Set FLAG := TRUE, and Return.
```

Suppose an edge (A, B) is to be inserted in the graph *G*. (The procedure will assume that both A and B are already nodes in the graph *G*.) The procedure first finds the location LOCA of A and the location LOCB of B in the node list. Then (A, B) is inserted as an edge in *G* by inserting LOCB in the list of successors of A, which has the list pointer ADJ[LOCA]. Again, a logical variable FLAG is used to indicate overflow. The procedure follows.

```
Procedure 8.7: INSEDGE(NODE, NEXT, ADJ, START, DEST, LINK, AVAILE, A, B,
               FLAG)
               This procedure inserts the edge (A, B) in the graph G.
               1. Call FIND(NODE, NEXT, START, A, LOCA).
               2. Call FIND(NODE, NEXT, START, B, LOCB).
               3. [OVERFLOW?] If AVAILE = NULL, then: Set FLAG := FALSE, and
                  Return.
               4. [Remove node from AVAILE list.] Set NEW := AVAILE and
                  AVAILE := LINK[AVAILE].
               5. [Insert LOCB in list of successors of A.]
                  Set DEST[NEW] := LOCB, LINK[NEW] := ADJ[LOCA] and
                  ADJ[LOCA] := NEW.
               6. Set FLAG := TRUE, and Return.
```

The procedure must be modified by using Procedure 8.6 if A or B is not a node in the graph *G*.

Deleting from a Graph

Suppose an edge (A, B) is to be deleted from the graph *G*. (Our procedure will assume that A and B are both nodes in the graph *G*.) Again, we must first find the location LOCA of A and the location LOCB of B in the node list. Then we simply delete LOCB from the list of successors of A, which has the list pointer ADJ[LOCA]. A logical variable FLAG is used to indicate that there is no such edge in the graph *G*. The procedure follows.

```
Procedure 8.8: DELEDGE(NODE, NEXT, ADJ, START, DEST, LINK, AVAILE, A, B,
               FLAG)
               This procedure deletes the edge (A, B) from the graph G.
               1. Call FIND(NODE, NEXT, START, A, LOCA). [Locates node A.]
               2. Call FIND(NODE, NEXT, START, B, LOCB). [Locates node B.]
               3. Call DELETE(DEST, LINK, ADJ[LOCA], AVAILE, LOCB, FLAG).
                  [Uses Procedure 8.4.]
               4. Return.
```

Suppose a node N is to be deleted from the graph *G*. This operation is more complicated than the search and insertion operations and the deletion of an edge, because we must also delete all the edges that contain N. Note these edges come in two kinds; those that begin at N and those that end at N. Accordingly, our procedure will consist mainly of the following four steps:

(1) Find the location LOC of the node N in *G*.

(2) Delete all edges ending at N; that is, delete LOC from the list of successors of each node M in *G*. (This step requires traversing the node list of *G*.)

(3) Delete all the edges beginning at N. This is accomplished by finding the location BEG of the first successor and the location END of the last successor of N, and then adding the successor list of N to the free AVAILE list.

(4) Delete N itself from the list NODE.

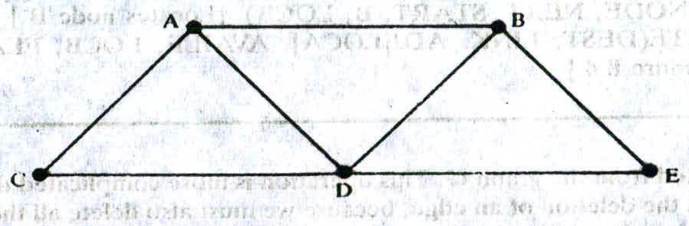
The procedure follows.

```
Procedure 8.9: DELNODE(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK,
              AVAILE, N, FLAG)
              This procedure deletes the node N from the graph G.

    1. Call FIND(NODE, NEXT, START, N, LOC). [Locates node N.]
    2. If LOC = NULL, then: Set FLAG := FALSE, and Return.
    3. [Delete edges ending at N.]
       (a) Set PTR := START.
       (b) Repeat while PTR ≠ NULL:
               (i)  Call DELETE(DEST, LINK, ADJ[PTR], AVAILE, LOC,
                    FLAG).
               (ii) Set PTR := NEXT[PTR].
           [End of loop.]
    4. [Successor list empty?] If ADJ[LOC] = NULL, then: Go to Step 7.
    5. [Find the first and last successor of N.]
       (a) Set BEG := ADJ[LOC], END := ADJ[LOC] and
           PTR := LINK[END].
       (b) Repeat while PTR ≠ NULL:
               Set END := PTR and PTR := LINK[PTR].
           [End of loop.]
    6. [Add successor list of N to AVAILE list.]
       Set LINK[END] := AVAILE and AVAILE := BEG.
    7. [Delete N using Procedure 8.4.]
       Call DELETE(NODE, NEXT, START, AVAILN, N, FLAG).
    8. Return.
```

EXAMPLE 8.6

Consider the (undirected) graph G in Fig. 8-12(a), whose adjacency lists appear in Fig. 8-12(b). Observe that G has 14 directed edges, since there are 7 undirected edges.



(a)

Adjacency Lists
A: B, C, D
B: A, D, E
C: A, D
D: A, B, C, E
E: B, D

(b)

Fig. 8-12

Suppose G is maintained in memory as in Fig. 8-13(a). Furthermore, suppose node B is deleted from G by using Procedure 8.9. We obtain the following steps:

	NODE	NEXT	ADJ
1	A	2	1
2	Ⓑ	3	4
3	C	4	7
4	D	5	9
5	E	0	13
6		7	
7		8	
8		0	

START = 1
AVAILN = 6

	DEST	LINK
1	②	2
2	3	3
3	4	0
4	①	5
5	④	6
6	⑤	0
7	1	8
8	4	0
9	1	10
10	②	11
11	3	12
12	5	0
13	②	14
14	4	0
15		16
16		0

AVAILE = 16

(*a*) Before deletion.

	NODE	NEXT	ADJ
1	A	3	2
2		6	
3	C	4	7
4	D	5	9
5	E	0	14
6		7	
7		8	
8		0	

START = 1
AVAILN = 2

	DEST	LINK
1		15
2	3	3
3	4	0
4		5
5		6
6		13
7	1	8
8	4	0
9	1	11
10		1
11	3	12
12	5	0
13		10
14	4	0
15		16
16		0

AVAILE = 4

(*b*) After deleting B.

Fig. 8-13

Step 1. Finds LOC = 2, the location of B in the node list.
Step 3. Deletes LOC = 2 from the edge list, that is, from each list of successors.
Step 5. Finds BEG = 4 and END = 6, the first and last successors of B.
Step 6. Deletes the list of successors from the edge list.
Step 7. Deletes node B from the node list.
Step 8. Returns.

The deleted elements are circled in Fig. 8-13(*a*). Figure 8-13(*b*) shows *G* in memory after node B (and its edges) are deleted.

8.7 TRAVERSING A GRAPH

Many graph algorithms require one to systematically examine the nodes and edges of a graph *G*. There are two standard ways that this is done. One way is called a breadth-first search, and the other is called a depth-first search. The breadth-first search will use a queue as an auxiliary structure to hold nodes for future processing, and analogously, the depth-first search will use a stack.

During the execution of our algorithms, each node N of *G* will be in one of three states, called the *status* of N, as follows:

STATUS = 1: (Ready state.) The initial state of the node N.
STATUS = 2: (Waiting state.) The node N is on the queue or stack, waiting to be processed.
STATUS = 3: (Processed state.) The node N has been processed.

We now discuss the two searches separately.

Breadth-First Search

The general idea behind a breadth-first search beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of the neighbors of A. And so on. Naturally, we need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a queue to hold nodes that are waiting to be processed, and by using a field STATUS which tells us the current status of any node. The algorithm follows.

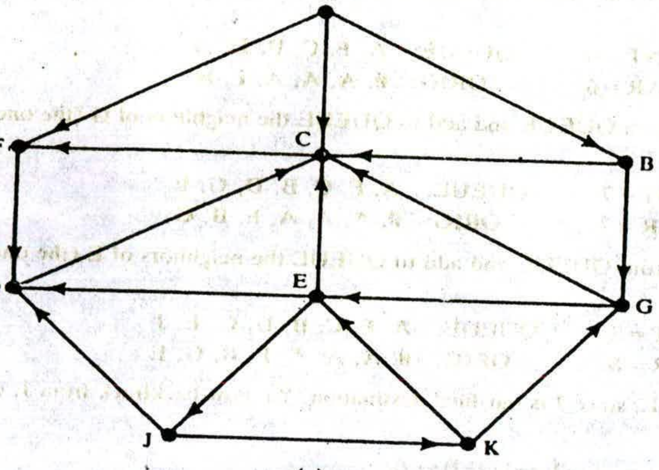
Algorithm A: This algorithm executes a breadth first search on a graph G beginning at a starting node A.

```
1. Initialize all nodes to the ready state (STATUS = 1).
2. Put the starting node A in QUEUE and change its status to the waiting state
   (STATUS = 2).
3. Repeat Steps 4 and 5 until QUEUE is empty:
4.     Remove the front node N of QUEUE. Process N and change the status of N
       to the processed state (STATUS = 3).
5.     Add to the rear of QUEUE all the neighbors of N that are in the steady state
       (STATUS = 1), and change their status to the waiting state (STATUS = 2).
   [End of Step 3 loop.]
6. Exit.
```

The above algorithm will process only those nodes which are reachable from the starting node A. Suppose one wants to examine all the nodes in the graph *G*. Then the algorithm must be modified so that it begins again with another node (which we will call B) that is still in the ready state. This node B can be obtained by traversing the list of nodes.

EXAMPLE 8.7

Consider the graph G in Fig. 8-14(a). (The adjacency lists of the nodes appear in Fig. 8-14(b).) Suppose G represents the daily flights between cities of some airline, and suppose we want to fly from city A to city J with the minimum number of stops. In other words, we want the minimum path P from A to J (where each edge has length 1).



(a)

Adjacency Lists
A: F, C, B
B: G, C
C: F
D: C
E: D, C, J
F: D
G: C, E
J: D, K
K: E, G

(b)

Fig. 8-14

The minimum path P can be found by using a breadth-first search beginning at city A and ending when J is encountered. During the execution of the search, we will also keep track of the origin of each edge by using an array ORIG together with the array QUEUE. The steps of our search follow.

(a) Initially, add A to QUEUE and add NULL to ORIG as follows:

```
FRONT = 1        QUEUE: A
REAR = 1          ORIG: Ø,
```

(b) Remove the front element A from QUEUE by setting FRONT := FRONT + 1, and add to QUEUE the neighbors of A as follows:

```
FRONT = 2        QUEUE: A, F, C, B
REAR = 4          ORIG: Ø, A, A, A
```

Note that the origin A of each of the three edges is added to ORIG.

(c) Remove the front element F from QUEUE by setting FRONT := FRONT + 1, and add to QUEUE the neighbors of F as follows:

```
FRONT = 3        QUEUE: A, F, C, B, D
REAR = 5          ORIG: Ø, A, A, A, F
```

(d) Remove the front element C from QUEUE, and add to QUEUE the neighbors of C (which are in the ready state) as follows:

```
FRONT = 4        QUEUE: A, F, C, B, D
REAR = 5          ORIG: Ø, A, A, A, F
```

Note that the neighbor F of C is not added to QUEUE, since F is not in the ready state (because F has already been added to QUEUE).

(*e*) Remove the front element B from QUEUE, and add to QUEUE the neighbors of B (the ones in the ready state) as follows:

```
FRONT = 5     QUEUE:  A, F, C, B, D, G
 REAR = 6      ORIG:  ∅, A, A, A, F, B
```

Note that only G is added to QUEUE, since the other neighbor, C is not in the ready state.

(*f*) Remove the front element D from QUEUE, and add to QUEUE the neighbors of D (the ones in the ready state) as follows:

```
FRONT = 6     QUEUE:  A, F, C, B, D, G
 REAR = 6      ORIG:  ∅, A, A, A, F, B
```

(*g*) Remove the front element G from QUEUE and add to QUEUE the neighbors of G (the ones in the ready state) as follows:

```
FRONT = 7     QUEUE:  A, F, C, B, D, G, E
 REAR = 7      ORIG:  ∅, A, A, A, F, B, G
```

(*h*) Remove the front element E from QUEUE and add to QUEUE the neighbors of E (the ones in the ready state) as follows:

```
FRONT = 8     QUEUE:  A, F, C, B, D, G, E, J
 REAR = 8      ORIG:  ∅, A, A, A, F, B, G, E
```

We stop as soon as J is added to QUEUE, since J is our final destination. We now backtrack from J, using the array ORIG to find the path *P*. Thus

$$J \leftarrow E \leftarrow G \leftarrow B \leftarrow A$$

is the required path *P*.

Depth-First Search

The general idea behind a depth-first search beginning at a starting node A is as follows. First we examine the starting node A. Then we examine each node N along a path *P* which begins at A; that is, we process a neighbor of A, then a neighbor of a neighbor of A, and so on. After coming to a "dead end," that is, to the end of the path *P*, we backtrack on *P* until we can continue along another path *P'*. And so on. (This algorithm is similar to the inorder traversal of a binary tree, and the algorithm is also similar to the way one might travel through a maze.) The algorithm is very similar to the breadth-first search except now we use a stack instead of the queue. Again, a field STATUS is used to tell us the current status of a node. The algorithm follows.

Algorithm B: This algorithm executes a depth-first search on a graph G beginning at a starting node A.

```
1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A onto STACK and change its status to the waiting state
   (STATUS = 2).
3. Repeat Steps 4 and 5 until STACK is empty.
4.     Pop the top node N of STACK. Process N and change its status to the
       processed state (STATUS = 3).
5.     Push onto STACK all the neighbors of N that are still in the ready state
       (STATUS = 1), and change their status to the waiting state (STATUS = 2).
   [End of Step 3 loop.]
6. Exit.
```

Again, the above algorithm will process only those nodes which are reachable from the starting node A. Suppose one wants to examine all the nodes in *G*. Then the algorithm must be modified so that it begins again with another node which we will call B—that is still in the ready state. This node B can be obtained by traversing the list of nodes.

EXAMPLE 8.8

Consider the graph *G* in Fig. 8-14(*a*). Suppose we want to find and print all the nodes reachable from the node J (including J itself). One way to do this is to use a depth-first search of *G* starting at the node J. The steps of our search follow.

(*a*) Initially, push J onto the stack as follows:

STACK: J

(*b*) Pop and print the top element J, and then push onto the stack all the neighbors of J (those that are in the ready state) as follows:

Print J STACK: D, K

(*c*) Pop and print the top element K, and then push onto the stack all the neighbors of K (those that are in the ready state) as follows:

Print K STACK: D, E, G

(*d*) Pop and print the top element G, and then push onto the stack all the neighbors of G (those in the ready state) as follows:

Print G STACK: D, E, C

Note that only C is pushed onto the stack, since the other neighbor, E, is not in the ready state (because E has already been pushed onto the stack).

(*e*) Pop and print the top element C, and then push onto the stack all the neighbors of C (those in the ready state) as follows:

Print C STACK: D, E, F

(*f*) Pop and print the top element F, and then push onto the stack all the neighbors of F (those in the ready state) as follows:

Print F STACK: D, E

Note that the only neighbor D of F is not pushed onto the stack, since D is not in the ready state (because D has already been pushed onto the stack).

(*g*) Pop and print the top element E, and push onto the stack all the neighbors of E (those in the ready state) as follows:

Print E STACK: D

(Note that none of the three neighbors of E is in the ready state.)

(*h*) Pop and print the top element D, and push onto the stack all the neighbors of D (those in the ready state) as follows:

Print D STACK: ϕ

The stack is now empty, so the depth-first search of *G* starting at J is now complete. Accordingly, the nodes which were printed,

J, K, G, C, F, E, D

are precisely the nodes which are reachable from J.

8.8 POSETS; TOPOLOGICAL SORTING

Suppose *S* is a graph such that each node v_i of *S* represents a task and each edge (u, v) means that the completion of the task *u* is a prerequisite for starting the task *v*. Suppose such a graph *S* contains a cycle, such as

$$P = (u, v, w, u)$$

This means that we cannot begin *v* until completing *u*, we cannot begin *w* until completing *v* and we cannot begin u until completing w. Thus we cannot complete any of the tasks in the cycle. Accordingly, such a graph S, representing tasks and a prerequisite relation, cannot have cycles.

Suppose S is a graph without cycles. Consider the relation $<$ on S defined as follows:

$$u < v \qquad \text{if there is a path from } u \text{ to } v$$

This relation has the following three properties:

(1) For each element u in S, we have $u \not< u$. (Irreflexivity.)

(2) If $u < v$, then $v \not< u$. (Asymmetry.)

(3) If $u < v$ and $v < w$, then $u < w$. (Transitivity.)

Such a relation $<$ on S is called a *partial ordering* of S, and S with such an ordering is called a *partially ordered set*, or *poset*. Thus a graph S without cycles may be regarded as a partially ordered set.

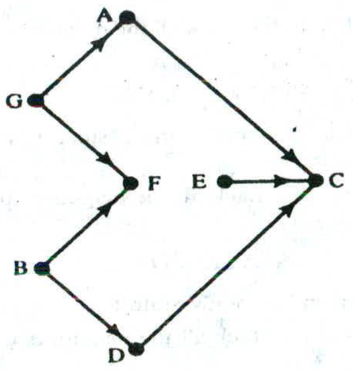
On the other hand, suppose S is a partially ordered set with the partial ordering denoted by $<$. Then S may be viewed as a graph whose nodes are the elements of S and whose edges are defined as follows:

$$(u, v) \qquad \text{is an edge in } S \text{ if} \qquad u < v$$

Furthermore, one can show that a partially ordered set S, regarded as a graph, has no cycles.

EXAMPLE 8.9

Let S be the graph in Fig. 8-15. Observe that S has no cycles. Thus S may be regarded as a partially ordered set. Note that $G < C$, since there is a path from G to C. Similarly, $B < F$ and $B < C$. On the other hand, $B \not< A$, since there is no path from B to A. Also, $A \not< B$.



(*a*)

Adjacency Lists	
A:	C
B:	D, F
C:	
D:	C
E:	C
F:	
G:	A, F

(*b*)

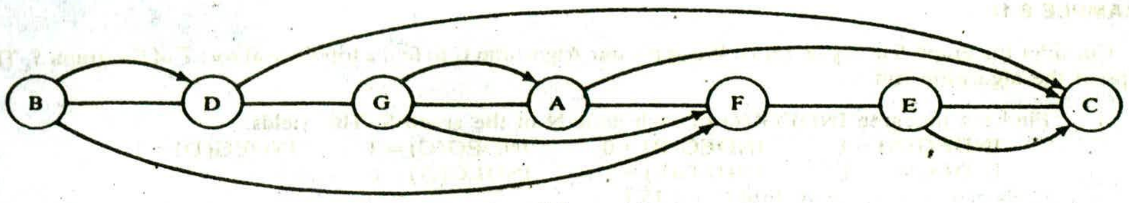
Fig. 8-15

Topological Sorting

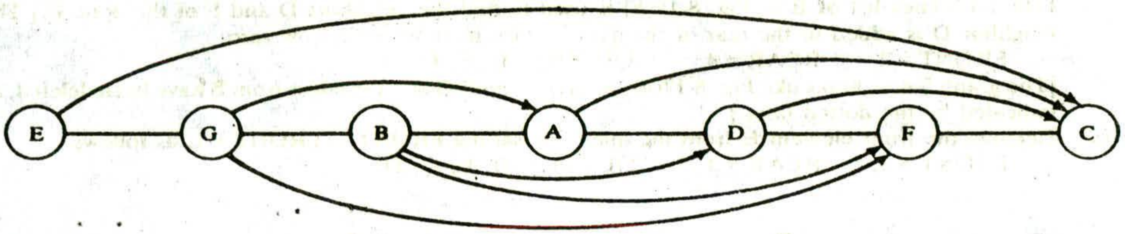
Let S be a directed graph without cycles (or a partially ordered set). A topological sort T of S is a linear ordering of the nodes of S which preserves the original partial ordering of S. That is: *If $u < v$ in S (i.e., if there is a path from u to v in S), then u comes before v in the linear ordering T.* Figure 8-16 shows two different topological sorts of the graph S in Fig. 8-15. We have included the edges in Fig. 8-16 to indicate that they agree with the direction of the linear ordering.

The following is the main theoretical result in this section.

Proposition 8.4: Let S be a finite directed graph without cycles or a finite partially ordered set. Then there exists a topological sort T of the set S.



(*a*)



(*b*)

Fig. 8-16 Two topological sorts.

Note that the proposition states only that a topological sort exists. We now give an algorithm which will find such a topological sort.

The main idea behind our algorithm to find a topological sort *T* of a graph *S* without cycles is that any node N with zero indegree, i.e., without any predecessors, may be chosen as the first element in the sort *T*. Accordingly, our algorithm will repeat the following two steps until the graph *S* is empty:

(1) Finding a node N with zero indegree

(2) Deleting N and its edges from the graph *S*

The order in which the nodes are deleted from the graph *S* will use an auxiliary array QUEUE which will temporarily hold all the nodes with zero indegree. The algorithm also uses a field INDEG such that INDEG(N) will contain the current indegree of the node N. The algorithm follows.

Algorithm C: This algorithm finds a topological sort T of a graph S without cycles.

```
1. Find the indegree INDEG(N) of each node N of S. (This can be done by traversing
   each adjacency list as in Prob. 8.15.)
2. Put in a queue all the nodes with zero indegree.
3. Repeat Steps 4 and 5 until the queue is empty.
4.     Remove the front node N of the queue (by setting FRONT := FRONT + 1).
5.     Repeat the following for each neighbor M of the node N:
           (a) Set INDEG(M) := INDEG(M) - 1.
               [This deletes the edge from N to M.]
           (b) If INDEG(M) = 0, then: Add M to the rear of the queue.
       [End of loop.]
   [End of Step 3 loop.]
6. Exit.
```

EXAMPLE 8.10

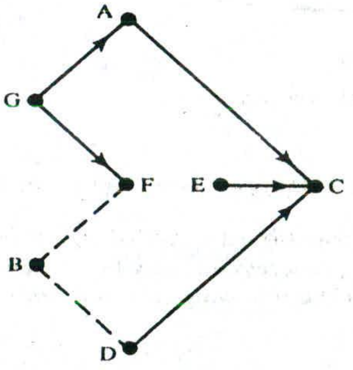
Consider the graph *S* in Fig. 8-15(*a*). We apply our Algorithm C to find a topological sort *T* of the graph *S*. The steps of the algorithm follow.

1. Find the indegree INDEG(N) of each node N of the graph S. This yields:
   INDEG(A) = 1 INDEG(B) = 0 INDEG(C) = 3 INDEG(D) = 1
   INDEG(E) = 0 INDEG(F) = 2 INDEG(G) = 0
   [This can be done as in Problem 8.15.]
2. Initially add to the queue each node with zero indegree as follows:
   FRONT = 1, REAR = 3, QUEUE: B, E, G

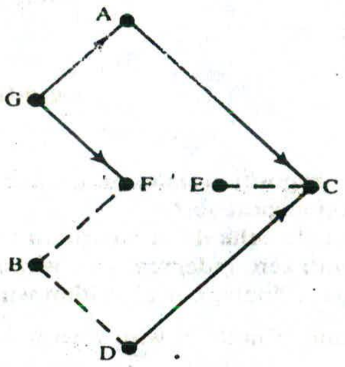
3*a*. Remove the front element B from the queue by setting FRONT := FRONT + 1, as follows:
   FRONT = 2, REAR = 3 QUEUE: B, E, G

3*b*. Decrease by 1 the indegree of each neighbor of B, as follows:
   INDEG(D) = 1 − 1 = 0 and INDEG(F) = 2 − 1 = 1
   [The adjacency list of B in Fig. 8-15(*b*) is used to find the neighbors D and F of the node B.] The neighbor D is added to the rear of the queue, since its indegree is now zero:
   FRONT = 2, REAR = 4 QUEUE: B, E, G, D
   [The graph S now looks like Fig. 8-17(*a*), where the node B and the edges from B have been deleted, as indicated by the dotted lines.]

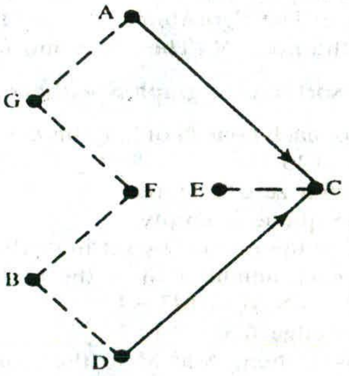
4*a*. Remove the front element E from the queue by setting FRONT := FRONT + 1, as follows:
   FRONT = 3, REAR = 4 QUEUE: B, E, G, D



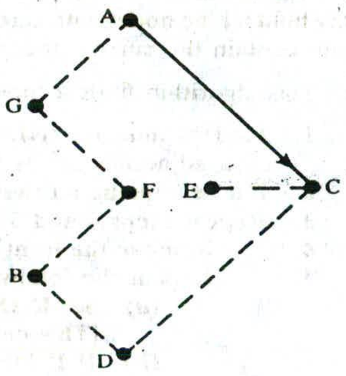
(*a*) B deleted.



(*b*) E deleted.



(*c*) G deleted.



(*d*) D deleted.

Fig. 8-17

4b. Decrease by 1 the indegree of each neighbor of E, as follows:
INDEG(C) = 3 − 1 = 2
[Since the indegree is nonzero, QUEUE is not changed. The graph S now looks like Fig. 8-17(*b*), where the node E and its edge have been deleted.]

5a. Remove the front element G from the queue by setting FRONT := FRONT + 1, as follows:
FRONT = 4, REAR = 4 QUEUE: B, E, G, D

5b. Decrease by 1 the indegree of each neighbor of G, as follows:
INDEG(A) = 1 − 1 = 0 and INDEG(F) = 1 − 1 = 0
Both A and F are added to the rear of the queue, as follows:
FRONT = 4, REAR = 6 QUEUE: B, E, G, D, A, F
[The graph S now looks like Fig. 8-17(*c*), where G and its two edges have been deleted.]

6a. Remove the front element D from the queue by setting FRONT := FRONT + 1, as follows:
FRONT = 5, REAR = 6 QUEUE: B, E, G, D, A, F

6b. Decrease by 1 the indegree of each neighbor of D, as follows:
INDEG(C) = 2 − 1 = 1
[Since the indegree is nonzero, QUEUE is not changed. The graph S now looks like Fig. 8-17(*d*), where D and its edge have been deleted.]

7a. Remove the front element A from the queue by setting FRONT := FRONT + 1, as follows:
FRONT = 6, REAR = 6 QUEUE: B, E, G, D, A, F

7b. Decrease by 1 the indegree of each neighbor of A, as follows:
INDEG(C) = 1 − 1 = 0
Add C to the rear of the queue, since its indegree is now zero:
FRONT = 6, REAR = 7 QUEUE: B, E, G, D, A, F, C

8a. Remove the front element F from the queue by setting FRONT := FRONT + 1, as follows:
FRONT = 7, REAR = 7 QUEUE: B, E, G, D, A, F, C

8b. The node F has no neighbors, so no change takes place.

9a. Remove the front element C from the queue by setting FRONT := FRONT + 1, as follows:
FRONT = 8, REAR = 7 QUEUE: B, E, G, D, A, F, C

9b. The node C has no neighbors, so no other changes take place.

The queue now has no front element, so the algorithm is completed. The elements in the array QUEUE give the required topological sort *T* of *S* as follows:

T: B, E, G, D, A, F, C

The algorithm could have stopped in Step 7*b*, where REAR is equal to the number of nodes in the graph *S*.

Solved Problems

GRAPH TERMINOLOGY

8.1 Consider the (undirected) graph G in Fig. 8-18. (a) Describe G formally in terms of its set V of nodes and its set E of edges. (b) Find the degree of each node.

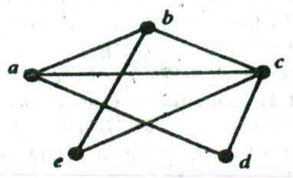


Fig. 8-18

(a) There are 5 nodes, a, b, c, d and e; hence $V = \{a, b, c, d, e\}$. There are 7 pairs $[x, y]$ of nodes such that node x is connected with node y; hence

$$E = \{[a, b], [a, c], [a, d], [b, c], [b, e], [c, d], [c, e]\}$$

(b) The degree of a node is equal to the number of edges to which it belongs; for example, $\deg(a) = 3$, since a belongs to three edges, $[a, b]$, $[a, c]$ and $[a, d]$. Similarly, $\deg(b) = 3$, $\deg(c) = 4$, $\deg(d) = 2$ and $\deg(e) = 2$.

8.2 Consider the multigraphs in Fig. 8-19. Which of them are (a) connected; (b) loop-free (i.e., without loops); (c) graphs?

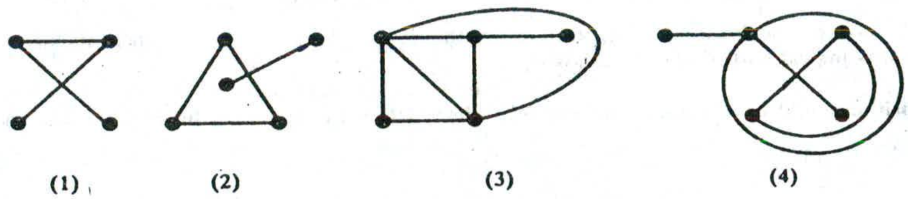


Fig. 8-19

(a) Only multigraphs 1 and 3 are connected.

(b) Only multigraph 4 has a loop (i.e., an edge with the same endpoints).

(c) Only multigraphs 1 and 2 are graphs. Multigraph 3 has multiple edges, and multigraph 4 has multiple edges and a loop.

8.3 Consider the connected graph G in Fig. 8-20. (a) Find all simple paths from node A to node F. (b) Find the distance between A and F. (c) Find the diameter of G. (The diameter of G is the maximum distance existing between any two of its nodes.)

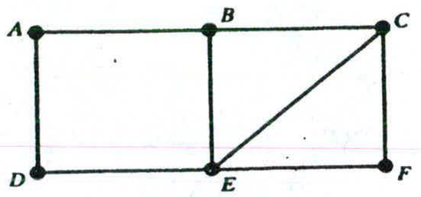


Fig. 8-20

(*a*) A simple path from A to F is a path such that no node and hence no edge is repeated. There are seven such simple paths:

$$(A, B, C, F) \qquad (A, B, E, F) \qquad (A, D, E, F) \qquad (A, D, E, C, F)$$
$$(A, B, C, E, F) \qquad (A, B, E, C, F) \qquad (A, D, E, B, C, F)$$

(*b*) The distance from A to F equals 3, since there is a simple path, (A, B, C, F), from A to F of length 3 and there is no shorter path from A to F.

(*c*) The distance between A and F equals 3, and the distance between any two nodes does not exceed 3; hence the diameter of the graph G equals 3.

8.4 Consider the (directed) graph G in Fig. 8-21. (*a*) Find all the simple paths from X to Z. (*b*) Find all the simple paths from Y to Z. (*c*) Find indeg(Y) and outdeg(Y). (*d*) Are there any sources or sinks?

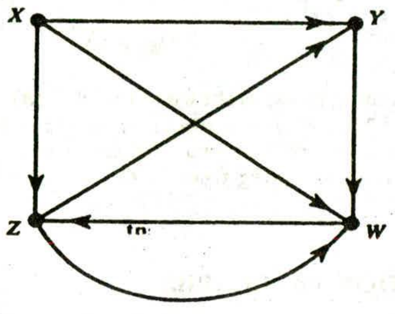


Fig. 8-21

(*a*) There are three simple paths from X to Z: (X, Z), (X, W, Z) and (X, Y, W, Z).

(*b*) There is only one simple path from Y to Z: (Y, W, Z).

(*c*) Since two edges enter Y (i.e., end at Y), we have indeg(Y) = 2. Since only one edge leaves Y (i.e., begins at Y), outdeg(Y) = 1.

(*d*) X is a source, since no edge enters X (i.e., indeg(X) = 0) but some edges leave X (i.e., outdeg(X) > 0). There are no sinks, since each node has a nonzero outdegree (i.e., each node is the initial point of some edge).

8.5 Draw all (nonsimilar) trees with exactly 6 nodes. (A graph G is *similar* to a graph G' if there is a one-to-one correspondence between the set V of nodes of G and the set V' of nodes of G' such that (u, v) is an edge in G if and only if the corresponding pair (u', v') of nodes is an edge in G'.)

There are six such trees, which are exhibited in Fig. 8-22. The first tree has diameter 5, the next two diameter 4, the next two diameter 3 and the last one diameter 2. Any other tree with 6 nodes will be similar to one of these trees.

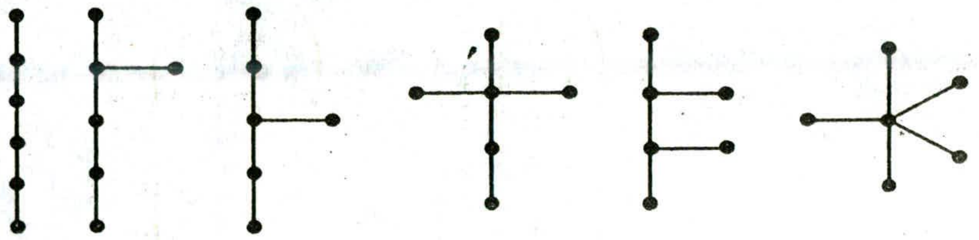


Fig. 8-22

8.6 Find all spanning trees of the graph G shown in Fig. 8-23(a). (A tree T is called a *spanning tree* of a connected graph G if T has the same nodes as G and all the edges of T are contained among the edges of G.)

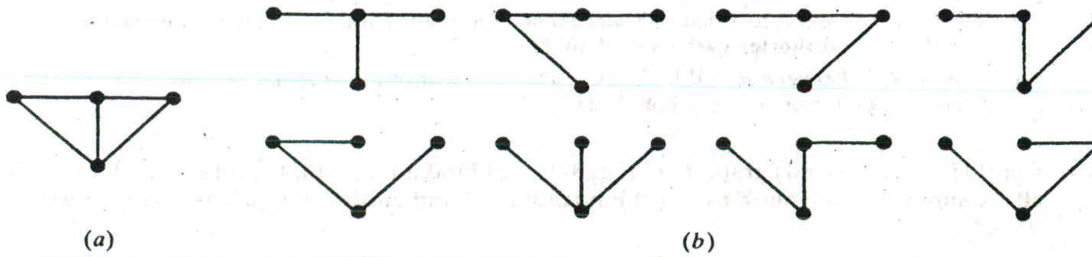


Fig. 8-23

There are eight such spanning trees, as shown in Fig. 8-23(b). Since G has 4 nodes, each spanning tree T must have $4 - 1 = 3$ edges. Thus each spanning tree can be obtained by deleting 2 of the 5 edges of G. This can be done in 10 ways, except that two of them lead to disconnected graphs. Hence the eight spanning trees shown are all the spanning trees of G.

SEQUENTIAL REPRESENTATION OF GRAPHS

8.7 Consider the graph G in Fig. 8-21. Suppose the nodes are stored in memory in an array DATA as follows:

DATA: X, Y, Z, W

(a) Find the adjacency matrix A of the graph G.
(b) Find the path matrix P of G using powers of the adjacency matrix A.
(c) Is G strongly connected?

(a) The nodes are normally ordered according to the way they appear in memory; that is, we assume $v_1 = \mathrm{X}$, $v_2 = \mathrm{Y}$, $v_3 = \mathrm{Z}$ and $v_4 = \mathrm{W}$. The adjacency matrix A of G follows:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Here $a_{ij} = 1$ if there is a node from v_i to v_j; otherwise, $a_{ij} = 0$.

(b) Since G has 4 nodes, compute A^2, A^3, A^4 and $B_4 = A + A^2 + A^3 + A^4$:

$$A^2 = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 2 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 1 \end{pmatrix} \qquad B_4 = \begin{pmatrix} 0 & 5 & 6 & 8 \\ 0 & 1 & 2 & 3 \\ 0 & 3 & 3 & 5 \\ 0 & 2 & 3 & 5 \end{pmatrix}$$

The path matrix P is now obtained by setting $p_{ij} = 1$ wherever there is a nonzero entry in the matrix B_4. Thus

$$P = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

(c) The path matrix shows that there is no path from v_2 to v_1. In fact, there is no path from any node to v_1. Thus G is not strongly connected.

8.8 Consider the graph G in Fig. 8-21 and its adjacency matrix A obtained in Prob. 8.7. Find the path matrix P of G using Warshall's algorithm rather than the powers of A.

Compute the matrices P_0, P_1, P_2, P_3 and P_4 where initially $P_0 = A$ and

$$P_k[i, j] = P_{k-1}[i, j] \vee (P_{k-1}[i, j] \wedge P_{k-1}[k, j])$$

That is,

$$P_k[i, j] = 1 \quad \text{if} \quad P_{k-1}[i, j] = 1 \quad \text{or both} \quad P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

Then:

$$P_1 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$P_3 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \qquad P_4 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Observe that $P_0 = P_1 = P_2 = A$. The changes in P_3 occur for the following reasons:

$$P_3(4, 2) = 1 \quad \text{because} \quad P_2(4, 3) = 1 \quad \text{and} \quad P_2(3, 2) = 1$$
$$P_3(4, 4) = 1 \quad \text{because} \quad P_2(4, 3) = 1 \quad \text{and} \quad P_2(3, 4) = 1$$

The changes in P_4 occur similarly. The last matrix, P_4, is the required path matrix P of the graph G.

8.9 Consider the (undirected) weighted graph G in Fig. 8-24. Suppose the nodes are stored in memory in an array DATA as follows:

DATA: A, B, C, X, Y

Find the weight matrix $W = (w_{ij})$ of the graph G.

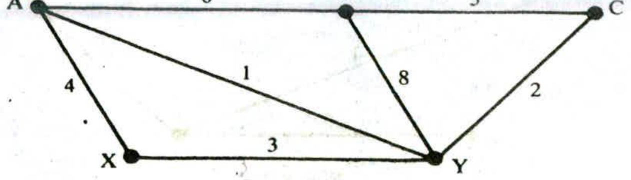


Fig. 8-24

Assuming $v_1 = \text{A}$, $v_2 = \text{B}$, $v_3 = \text{C}$, $v_4 = \text{X}$ and $v_5 = \text{Y}$, we arrive at the following weight matrix W of G:

$$W = \begin{pmatrix} 0 & 6 & 0 & 4 & 1 \\ 6 & 0 & 5 & 0 & 8 \\ 0 & 5 & 0 & 0 & 2 \\ 4 & 0 & 0 & 0 & 3 \\ 1 & 8 & 2 & 3 & 0 \end{pmatrix}$$

Here w denotes the weight of the edge from v_i to v_j. Since G is undirected, W is a symmetric matrix, that is, w

8.10 Suppose G is a graph (undirected) which is cycle-free, that is, without cycles. Let $P = (p_{ij})$ be the path matrix of G.

(*a*) When can an edge $[v_i, v_j]$ be added to G so that G is still cycle-free?

(*b*) How does the path matrix P change when an edge $[v_i, v_j]$ is added to G?

(*a*) The edge $[v_i, v_j]$ will form a cycle when it is added to G if and only if there already is a path between v_i and v_j. Hence the edge may be added to G when $p_{ij} = 0$.

(*b*) First set $p_{ij} = 1$, since the edge is a path from v_i to v_j. Also, set $p_{st} = 1$ if $p_{si} = 1$ and $p_{jt} = 1$. In other words, if there are both a path P_1 from v_s to v_i and a path P_2 from v_j to v_t, then P_1, $[v_i, v_j]$, P_2 will form a path from v_s to v_t.

8.11 A minimum spanning tree T of a weighted graph G is a spanning tree of G (see Prob. 8.6) which has the minimum weight among all the spanning trees of G.

(*a*) Describe an algorithm to find a minimum spanning tree T of a weighted graph G.

(*b*) Find a minimum spanning tree T of the graph in Fig. 8-24.

(*a*) **Algorithm P8.11:** This algorithm finds a minimum spanning tree T of a weighted graph G.

```
1. Order all the edges of G according to increasing weights.
2. Initialize T to be a graph consisting of the same nodes as G and no edges.
3. Repeat the following M − 1 times, where M is the number of nodes in G:
       Add to T an edge E of G with minimum weight such that E does not
       form a cycle in T.
   [End of loop.]
4. Exit.
```

Step 3 may be implemented using the results of Prob. 8.10. Problem 8.10(*a*) tells us which edge *e* may be added to T so that no cycle is formed—i.e., so that T is still cycle-free—and Prob. 8.10(*b*) tells us how to keep track of the path matrix P of T as each edge *e* is added to T.

(*b*) Apply Algorithm P8.11 to obtain the minimum spanning tree T in Fig. 8-25. Although [A, X] has less weight than [B, C], we cannot add [A, X] to T, since it would form a cycle with [A, Y] and [Y, X].

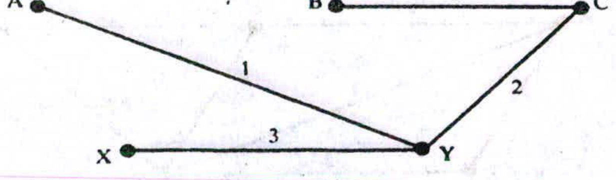


Fig. 8-25

8.12 Suppose a weighted graph G is maintained in memory by a node array DATA and a weight matrix W as follows:

DATA: X, Y, S, T

$$W = \begin{pmatrix} 0 & 0 & 3 & 0 \\ 5 & 0 & 1 & 7 \\ 2 & 0 & 0 & 4 \\ 0 & 6 & 8 & 0 \end{pmatrix}$$

Draw a picture of G.

The picture appears in Fig. 8-26. The nodes are labeled by the entries in DATA. Also, if $w_{ij} \neq 0$, then there is an edge from v_i to v_j with weight w_{ij}. (We assume $v_1 = X$, $v_2 = Y$, $v_3 = S$ and $v_4 = T$, the order in which the nodes appear in the array DATA.)

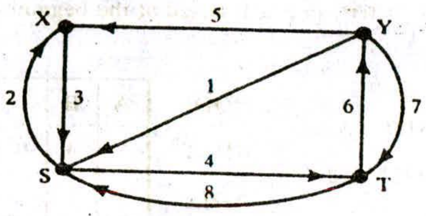


Fig. 8-26

LINKED REPRESENTATION OF GRAPHS

8.13 A graph G is stored in memory as follows:

	1	2	3	4	5	6	7	8
NODE	A	B		E		D	C	
NEXT	7	4	0	6	8	0	2	3
ADJ	1	2		5		7	9	

START = 1, AVAILN = 5

	1	2	3	4	5	6	7	8	9	10
DEST	2	6	4		6	7	4		4	6
LINK	10	3	6	0	0	0	0	4	0	0

AVAILE = 8

Draw the graph G.

First find the neighbors of each NODE[K] by traversing its adjacency list, which has the pointer ADJ[K]. This yields:

A: 2(B) and 6(D)	C: 4(E)
B: 6(D), 4(E) and 7(C)	D: 4(E)
	E: 6(D)

Then draw the diagram as in Fig. 8-27.

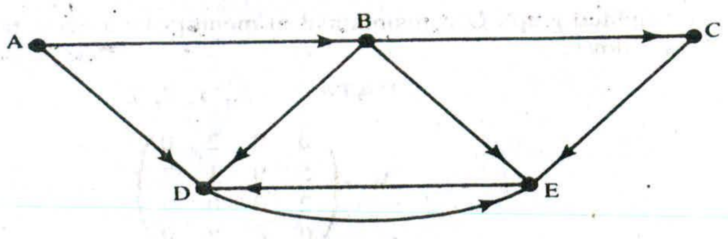


Fig. 8-27

8.14 **Find the changes in the linked representation of the graph G in Prob. 8.13 if the following operations occur: (*a*) Node F is added to G. (*b*) Edge (B, E) is deleted from G. (*c*) Edge (A, F) is added to G. Draw the resultant graph G.**

(*a*) The node list is not sorted, so F is inserted at the beginning of the list, using the first available free node as follows:

START = 5

AVAILN = 8

	1	2	3	4	5	6	7	8
NODE	A	B		E	F	D	C	
NEXT	7	4	0	6	1	0	2	3
ADJ	1	2		5	0	7	9	

Observe that the edge list does not change.

(*b*) Delete LOC = 4 of node E from the adjacency list of node B as follows:

AVAILE = 3

	1	2	3	4	5	6	7	8	9	10
DEST	2	6			6	7	4		4	6
LINK	10	6	8	0	0	0	0	4	0	0

Observe that the node list does not change.

(*c*) The location LOC = 5 of the node F is inserted at the beginning of the adjacency list of the node A, using the first available free edge. The changes are as follows:

ADJ[1] = 3

AVAILE = 8

	1	2	3	4	5	6	7	8	9	10
DEST	2	6	5		6	7	4		4	6
LINK	10	6	1	0	0	0	0	4	0	0

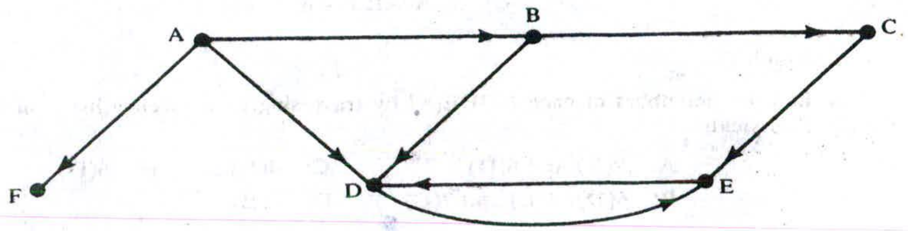


Fig. 8-28

The only change in the node list is the ADJ[1] = 3. (Observe that the shading indicates the changes in the lists.) The updated graph G appears in Fig. 8-28.

8.15 Suppose a graph G is maintained in memory in the form

GRAPH(NODE, NEXT, ADJ, START, DEST, LINK)

Write a procedure which finds the indegree INDEG and the outdegree OUTDEG of each node of G.

First we traverse the node list, using the pointer PTR in order to initialize the arrays INDEG and OUTDEG to zero. Then we traverse the node list, using the pointer PTRA, and for each value of PTRA, we traverse the list of neighbors of NODE[PTRA], using the pointer PTRB. Each time an edge is encountered, PTRA gives the location of its initial node and DEST[PTRB] gives the location of its terminal node. Accordingly, each edge updates the arrays INDEG and OUTDEG as follows:

OUTDEG[PTRA] := OUTDEG[PTRA] + 1

and

INDEG[DEST[PTRB]] := INDEG[DEST[PTRB]] + 1

The formal procedure follows.

```
Procedure P8.15: DEGREE(NODE, NEXT, ADJ, START, DEST, LINK, INDEG, OUTDEG)
                 This procedure finds the indegree INDEG and outdegree OUTDEG of each
                 node in the graph G in memory.

  1. [Initialize arrays INDEG and OUTDEG.]
     (a) Set PTR := START.
     (b) Repeat while PTR ≠ NULL: [Traverses node list.]
             (i)  Set INDEG[PTR] := 0 and OUTDEG[PTR] := 0.
             (ii) Set PTR := NEXT[PTR].
         [End of loop.]
  2. Set PTRA := START.
  3. Repeat Steps 4 to 6 while PTRA ≠ NULL: [Traverses node list.]
  4.     Set PTRB := ADJ[PTRA].
  5.     Repeat while PTRB ≠ NULL: [Traverses list of neighbors.]
             (a) Set OUTDEG[PTRA] := OUTDEG[PTRA] + 1 and
                 INDEG[DEST[PTRB]] := INDEG[DEST[PTRB]] + 1.
             (b) Set PTRB := LINK[PTRB].
         [End of inner loop using pointer PTRB.]
  6.     Set PTRA := NEXT[PTRA].
     [End of Step 3 outer loop using the pointer PTRA.]
  7. Return.
```

8.16 Suppose G is a finite undirected graph. Then G consists of a finite number of disjoint connected components. Describe an algorithm which finds the number NCOMP of connected components of G. Furthermore, the algorithm should assign a component number COMP(N) to every node N in the same connected component of G such that the component numbers range from 1 to NCOMP.

The general idea of the algorithm is to use a breadth-first or depth-first search to find all nodes N reachable from a starting node A and to assign them the same component number. The algorithm follows.

Algorithm P8.16: Finds the connected components of an undirected graph G.

```
1. Initially set COMP(N) := 0 for every node N in G, and initially set L := 0.
2. Find a node A such that COMP(A) = 0. If no such node A exists, then:
       Set NCOMP := L, and Exit.
   Else:
       Set L := L + 1 and set COMP(A) := L.
3. Find all nodes N in G which are reachable from A (using a breadth-first search or a
   depth-first search) and set COMP(N) = L for each such node N.
4. Return to Step 2.
```

MISCELLANEOUS PROBLEMS

8.17 Suppose G is an undirected graph with m nodes $v_1, v_2, \ldots, v_m$ and n edges $e_1, e_2, \ldots, e_n$. The *incidence matrix* of G is the $m \times n$ matrix $M = (m_{ij})$ where

$$m_{ij} = \begin{cases} 1 & \text{if node } v_i \text{ belongs to edge } e_j \\ 0 & \text{otherwise} \end{cases}$$

Find the incidence matrix M of the graph G in Fig. 8-29.

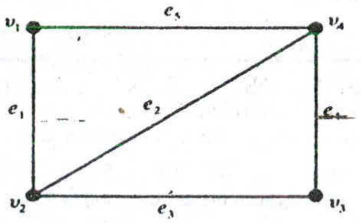


Fig. 8-29

Since G has 4 nodes and 5 edges, M is a 4×5 matrix. Set $m_{ij} = 1$ if v_i belongs to e_j. This yields the following matrix M:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

8.18 Suppose u and v are distinct nodes in an undirected graph G. Prove:

(*a*) If there is a path P from u to v, then there is a simple path Q from u to v.

(*b*) If there are two distinct paths P_1 and P_2 from u to v, then G contains a cycle.

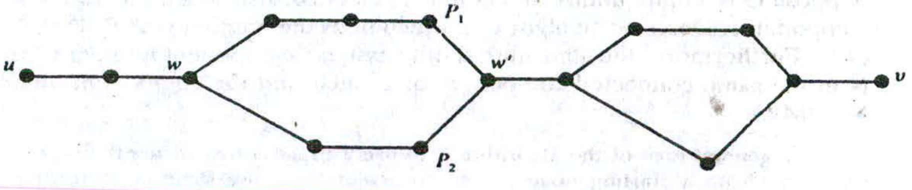


Fig. 8-30

(*a*) Suppose $P = (v_0, v_1, \ldots, v_n)$ where $u = v_0$ and $v = v_n$. If $v_i = v_j$, then

$$P' = (v_0, \ldots, v_i, v_{j+1}, \ldots, v_n)$$

is a path from u to v which is shorter than P. Repeating this process, we finally obtain a path Q from u to v whose nodes are distinct. Thus Q is a simple path from u to v.

(*b*) Let w be a node in P_1 and P_2 such that the next nodes in P_1 and P_2 are distinct. Let w' be the first node following w which lies on both P_1 and P_2. (See Fig. 8-30.) Then the subpaths of P_1 and P_2 between w and w' have no nodes in common except w and w'; hence these two subpaths form a cycle.

8.19 **Prove Proposition 8.2: Let A be the adjacency matrix of a graph G. Then $a_K(i, j)$, the ij entry in the matrix A^K, gives the number of paths of length K from v_i to v_j.**

The proof is by induction on K. Note first that a path of length 1 from v_i to v_j is precisely an edge (v_i, v_j). By definition of the adjacency matrix A, $a_1(i, j) = a_{ij}$ gives the number of edges from v_i to v_j. Hence the proposition is true for $K = 1$.

Supose $K > 1$. (Assume G has m nodes.) Since $A^K = A^{K-1}A$,

$$a_K(i, j) = \sum_{s=1}^{m} a_{K-1}(i, s)a_1(s, j)$$

By induction, $a_{K-1}(i, s)$ gives the number of paths of length $K - 1$ from v_i to v_s, and $a_1(s, j)$ gives the number of paths of length 1 from v_s to v_j. Thus $a_{K-1}(i, s)a_1(s, j)$ gives the number of paths of length K from v_i to v_j where v_s is the next-to-last node. Thus all the paths of length K from v_i to v_j can be obtained by summing up the $a_{K-1}(i, s)a_1(s, j)$ for all s. That is, $a_K(i, j)$ is the number of paths of length K from v_i to v_j. Thus the proposition is proved.

8.20 **Suppose G is a finite undirected graph without cycles. Prove each of the following:**

(*a*) If G has at least one edge, then G has a node v with degree 1.

(*b*) If G is connected—so that G is a tree—and if G has m nodes, then G has $m - 1$ edges.

(*c*) If G has m nodes and $m - 1$ edges, then G is a tree.

(*a*) Let $P = (v_0, v_1, \ldots, v_n)$ be a simple path of maximum length. Suppose $\deg(v_0) \neq 1$, and assume $[u, v_0]$ is an edge and $u \neq v_1$. If $u = v_i$ for $i > 1$, then $C = (v_i, v_0, \ldots, v_i)$ is a cycle. If $u \neq v_i$, then $P' = (u, v_0, \ldots, v_n)$ is a simple path with length greater than P. Each case leads to a contradiction. Hence $\deg(v_0) = 1$.

(*b*) The proof is by induction on m. Suppose $m = 1$. Then G consists of an isolated node and G has $m - 1 = 0$ edges. Hence the result is true for $m = 1$. Suppose $m > 1$. Then has a node v such that $\deg(v) = 1$. Delete v and its only edge $[v, v']$ from the graph G to obtain the graph G'. Then G is still connected and G is a tree with $m - 1$ nodes. By induction, G' has $m - 2$ edges. Hence G has $m - 1$ edges. Thus the result is true.

(*c*) Let $T_1, T_2, \ldots, T_s$ denote the connected components of G. Then each T_i is a tree. Hence each T_i has one more node than edges. Hence G has s more nodes than edges. But G has only one more node than edges. Hence $s = 1$ and G is a tree.

Supplementary Problems

GRAPH TERMINOLOGY

8.21 Consider the undirected graph G in Fig. 8-31. Find (*a*) all simple paths from node A to node H, (*b*) the diameter of G and (*c*) the degree of each node.

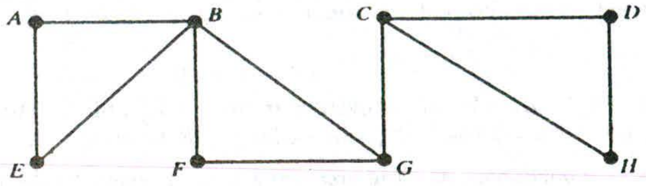


Fig. 8-31

8.22 Which of the multigraphs in Fig. 8-32 are (*a*) connected, (*b*) loop-free (i.e., without loops) and (*c*) graphs?

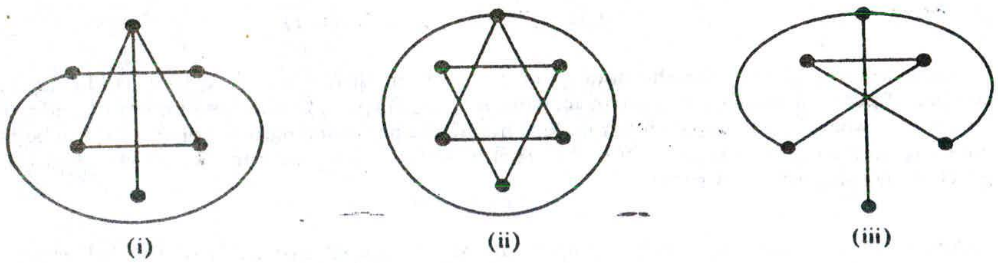


Fig. 8-32

8.23 Consider the directed graph G in Fig. 8-33. (*a*) Find the indegree and outdegree of each node. (*b*) Find the number of simple paths from v_1 to v_4. (*c*) Are there any sources or sinks?

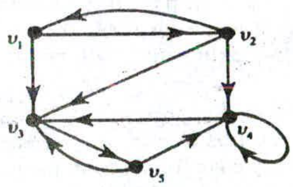


Fig. 8-33

8.24 Draw all (nonsimilar) trees with 5 or fewer nodes. (There are eight such trees.)

8.25 Find the number of spanning trees of the graph G in Fig. 8-34.

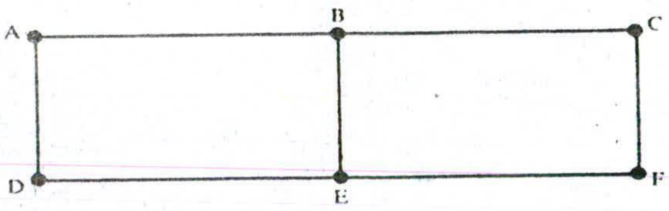


Fig. 8-34

SEQUENTIAL REPRESENTATION OF GRAPHS; WEIGHTED GRAPHS

8.26 Consider the graph G in Fig. 8-35. Suppose the nodes are stored in memory in an array DATA as follows:

DATA: X, Y, Z, S, T

(*a*) Find the adjacency matrix A of G. (*b*) Find the path matrix P or G. (*c*) Is G strongly connected?

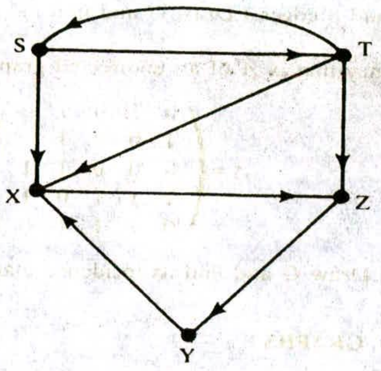


Fig. 8-35

8.27 Consider the weighted graph G in Fig. 8-36. Suppose the nodes are stored in an array DATA as follows:

DATA: X, Y, S, T

(*a*) Find the weight matrix W of G. (*b*) Find the matrix Q of shortest paths using Warshall's Algorithm 8.2.

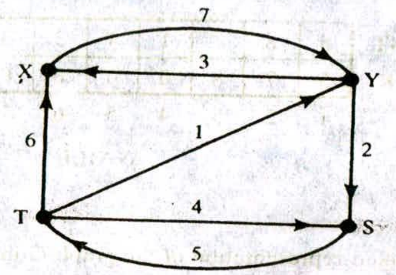


Fig. 8-36

8.28 Find a minimum spanning tree of the graph G in Fig. 8-37.

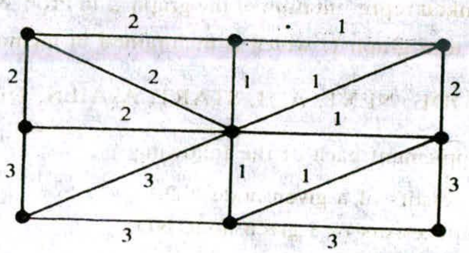


Fig. 8-37

8.29 The following is the incidence matrix M of an undirected graph G:

$$M = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

(Note that G has 5 nodes and 8 edges.) Draw G and find its adjacency matrix A.

8.30 The following is the adjacency matrix A of an undirected graph G:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(Note that G has 5 nodes.) Draw G and find its incidence matrix M.

LINKED REPRESENTATION OF GRAPHS

8.31 Suppose a graph G is stored in memory as follows:

	1	2	3	4	5	6	7	8
NODE	A		C	E		D		B
NEXT	4	0	8	0	7	3	2	1
ADJ	6		1	10		2		9

START = 6, AVAILN = 5

	1	2	3	4	5	6	7	8	9	10
DEST	8	8		1	4	3	3		6	3
LINK	5	7	8	0	0	0	0	0	4	0

AVAILE = 3

Draw the graph G.

8.32 Find the changes in the linked representation of the graph G in Prob. 8.31 if edge (C, E) is deleted and edge (D, E) is inserted.

8.33 Find the changes in the linked representation of the graph G in Prob. 8.31 if a node F and the edges (E, F) and (F, D) are inserted into

8.34 Find the changes in the linked representation of the graph G in Prob. 8.31 if the node B is deleted from G.

Problems 8.35 to 8.38 refer to a graph G which is maintained in memory by a linked representation:

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

8.35 Write a procedure to supplement each of the following:

(*a*) Print the list of successors of a given node ND.

(*b*) Print the list of predecessors of a given node ND.

8.36 Write a procedure which determines whether or not G is an undirected graph.

8.37 Write a procedure which finds the number M of nodes of G and then finds the $M \times M$ adjacency matrix A of G. (The nodes are ordered according to their order in the node list of G.)

8.38 Write a procedure which determines whether there are any sources or sinks in G.

Problems 8.39 to 8.40 refer to a weighted graph G which is stored in memory using a linked representation as follows:

GRAPH(NODE, NEXT, ADJ, START, AVAILN, WEIGHT, DEST, LINK, AVAILE)

8.39 Write a procedure which finds the shortest path from a given node NA to a given node NB.

8.40 Write a procedure which finds the longest simple path from a given node NA to a given node NB.

Programming Problems

8.41 Suppose a graph G is input by means of an integer M, representing the nodes 1, 2, . . . , M, and a list of N ordered pairs of the integers, representing the edges of G. Write a procedure for each of the following:

(*a*) To find the M × M adjacency matrix A of the graph G.

(*b*) To use the adjacency matrix A and Warshall's algorithm to find the path matrix P of the graph G.

Test the above using the following data:

(i) M = 5; N = 8; (3, 4), (5, 3), (2, 4), (1, 5), (3, 2), (4, 2), (3, 1), (5, 1).

(ii) M = 6; N = 10; (1, 6), (2, 1), (2, 3), (3, 5), (4, 5), (4, 2), (2, 6), (5, 3), (4, 3), (6, 4)

8.42 Suppose a weighted graph G is input by means of an integer M, representing the nodes 1, 2, . . . , M, and a list of N ordered triplets (a_i, b_i, w_i) of integers such that the pair (a_i, b_i) is an edge of G and w_i is its weight. Write a procedure for each of the following:

(*a*) To find the M × M weight matrix W of the graph G.

(*b*) To use the weight matrix W and Warshall's Algorithm 8.2 to find the matrix Q of shortest paths between the nodes.

Test the above using the following data:

(i) M = 4; N = 7; (1, 2, 5), (2, 4, 2), (3, 2, 3), (1, 1, 7), (4, 1, 4), (4, 3, 1). (Compare with Example 8.4.)

(ii) M = 5; N = 8; (3, 5, 3), (4, 1, 2), (5, 2, 2), (1, 5, 5), (1, 3, 1), (2, 4, 1), (3, 4, 4), (5, 4, 4).

8.43 Suppose an empty graph G is stored in memory using the linked representation

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

Assume NODE has space for 8 nodes and DEST has space for 12 edges. Write a program which executes the following operations on G:

(*a*) Inputs nodes A, B, C and D

(*b*) Inputs edges (A, B), (A, C), (C, B), (D, A), (B, D) and (C, D)

(*c*) Inputs nodes E and F

	CITY	LEFT	RIGHT	ADJ
1	Atlanta	0	2	12
2	Boston	0	0	1
3	Houston	0	0	14
4	New York	3	8	4
5		6		
6		0		
7	Washington	0	0	10
8	Philadelphia	0	7	6
9	Denver	10	4	8
10	Chicago	1	0	2

START = 9, AVAILN = 5

	NUMBER	PRICE	ORIG	DEST	LINK
1	201	80	2	10	3
2	202	80	10	2	0
3	301	50	2	4	0
4	302	50	4	2	5
5	303	40	4	8	7
6	304	40	8	4	9
7	305	120	4	9	0
8	306	120	9	4	13
9	401	40	8	7	0
10	402	40	7	8	11
11	403	80	7	1	0
12	404	80	1	7	16
13	501	80	9	3	15
14	502	80	3	9	0
15	503	140	9	1	0
16	504	140	1	9	0
17					18
18					19
19					20
20					0

NUM = 16, AVAILE = 17

Fig. 8-38

(*d*) Inputs edges (B, E), (F, E), (D, F) and (F, B)

(*e*) Deletes edges (D, A) and (B, D)

(*f*) Deletes node A

Problems 8.44 to 8.48 refer to the data in Fig. 8-38, where the cities are stored as a binary search tree.

8.44 Write a procedure with input CITYA and CITYB which finds the flight number and cost of the flight from city A to city B, if a flight exists. Test the procedure using (*a*) CITYA = Chicago, CITYB = Boston; (*b*) CITYA = Washington, CITYB = Denver; and (*c*) CITYA = New York, CITYB = Philadelphia.

8.45 Write a procedure with input CITYA and CITYB which finds the way to fly from city A to city B with a minimum number of stops, and also finds its cost. Test the procedure using (*a*) CITYA = Boston, CITYB = Houston; (*b*) CITYA = Denver, CITYB = Washington; and (*c*) CITYA = New York, CITYB = Atlanta.

8.46 Write a procedure with input CITYA and CITYB which finds the cheapest way to fly from city A to city B and also finds the cost. Test the procedure using the data in Prob. 8.45. (Compare the results.)

8.47 Write a procedure which deletes a record from the file given the flight number NUMB. Test the program using (*a*) NUMB = 503 and NUMB = 504 and (*b*) NUMB = 303 and NUMB = 304.

8.48 Write a procedure which inputs a record of the form

(NUMBNEW, PRICENEW, ORIGNEW, DESTNEW)

Test the procedure using the following data:

(*a*)	NUMBNEW = 505,	PRICENEW = 80,	ORIGNEW = Chicago,	DESTNEW = Denver
	NUMBNEW = 506,	PRICENEW = 80,	ORIGNEW = Denver,	DESTNEW = Chicago
(*b*)	NUMBNEW = 601,	PRICENEW = 70,	ORIGNEW = Atlanta,	DESTNEW = Miami
	NUMBNEW = 602,	PRICENEW = 70,	ORIGNEW = Miami,	DESTNEW = Atlanta

(Note that a new city may have to be inserted into the binary search tree of cities.)

8.49 Translate the topological sort algorithm into a program which sorts a graph G. Assume G is input by its set V of nodes and its set E of edges. Test the program using the nodes A, B, C, D, X, Y, Z, S and T and the edges

(*a*) (A, Z), (S, Z), (X, D), (B, T), (C, B), (Y, X), (Z, X), (S, C) and (Z, B)

(*b*) (A, Z), (D, Y), (A, X), (Y, B), (S, Y), (C, T), (X, S), (B, A), (C, S) and (X, T)

(*c*) (A, C), (B, Z), (Y, A), (Z, X), (D, Z), (A, S), (B, T), (Z, Y), (T, Y) and (X, A)

8.50 Write a program which finds the number of connected components of an unordered graph G and also assigns a component number to each of its nodes. Assume G is input by its set V of nodes and its set E of (undirected) edges. Test the program using the nodes A, B, C, D, X, Y, Z, S and T and the edges:

(*a*) [A, X], [B, T], [Y, C], [S, Z], [D, T], [A, S], [Z, A], [D, B] and [X, S]

(*b*) [Z, C], [D, B], [A, X], [S, C], [D, T], [X, S], [Y, B], [T, B] and [S, Z]

Chapter 9

Sorting and Searching

9.1 INTRODUCTION

Sorting and searching are fundamental operations in computer science. *Sorting* refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data. *Searching* refers to the operation of finding the location of a given item in a collection of items.

There are many sorting and searching algorithms. Some of them, such as heapsort and binary search, have already been discussed throughout the text. The particular algorithm one chooses depends on the properties of the data and the operations one may perform on the data. Accordingly, we will want to know the complexity of each algorithm; that is, we will want to know the running time $f(n)$ of each algorithm as a function of the number n of input items. Sometimes, we will also discuss the space requirements of our algorithms.

Sorting and searching frequently apply to a file of records, so we recall some standard terminology. Each record in a file F can contain many fields, but there may be one particular field whose values uniquely determine the records in the file. Such a field K is called a *primary key*, and the values $k_1, k_2, \ldots$ in such a field are called *keys* or *key values*. Sorting the file F usually refers to sorting F with respect to a particular primary key, and searching in F refers to searching for the record with a given key value.

This chapter will first investigate sorting algorithms and then investigate searching algorithms. Some texts treat searching before sorting.

9.2 SORTING

Let A be a list of n elements $A_1, A_2, \ldots, A_n$ in memory. *Sorting A* refers to the operation of rearranging the contents of A so that they are increasing in order (numerically or lexicographically), that is, so that

$$A_1 \le A_2 \le A_3 \le \cdots \le A_n$$

Since A has n elements, there are $n!$ ways that the contents can appear in A. These ways correspond precisely to the $n!$ permutations of $1, 2, \ldots, n$. Accordingly, each sorting algorithm must take care of these $n!$ possibilities.

EXAMPLE 9.1

Suppose an array DATA contains 8 elements as follows:

DATA: 77, 33, 44, 11, 88, 22, 66, 55

After sorting, DATA must appear in memory as follows:

DATA: 11, 22, 33, 44, 55, 66, 77, 88

Since DATA consists of 8 elements, there are $8! = 40\,320$ ways that the numbers $11, 22, \ldots, 88$ can appear in DATA.

Complexity of Sorting Algorithms

The complexity of a sorting algorithm measures the running time as a function of the number n of items to be sorted. We note that each sorting algorithm S will be made up of the following operations, where $A_1, A_2, \ldots, A_n$ contain the items to be sorted and B is an auxiliary location:

(*a*) Comparisons, which test whether $A_i < A_j$ or test whether $A_i < B$

(*b*) Interchanges, which switch the contents of A_i and A_j or of A_i and B

(*c*) Assignments, which set $B := A_i$ and then set $A_j := B$ or $A_j := A_i$

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

There are two main cases whose complexity we will consider; the worst case and the average case. In studying the average case, we make the probabilistic assumption that all the $n!$ permutations of the given n items are equally likely. (The reader is referred to Sec. 2.5 for a more detailed discussion of complexity.)

Previously, we have studied the bubble sort (Sec. 4.6), quicksort (Sec. 6.5) and heapsort (Sec. 7.10). The approximate number of comparisons and the order of complexity of these algorithms are summarized in the following table:

Algorithm	Worst Case	Average Case
Bubble Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$
Quicksort	$\frac{n(n+3)}{2} = O(n^2)$	$1.4n \log n = O(n \log n)$
Heapsort	$3n \log n = O(n \log n)$	$3n \log n = O(n \log n)$

Note first that the bubble sort is a very slow way of sorting; its main advantage is the simplicity of the algorithm. Observe that the average-case complexity ($n \log n$) of heapsort is the same as that of quicksort, but its worst-case complexity ($n \log n$) seems quicker than quicksort (n^2). However, empirical evidence seems to indicate that quicksort is superior to heapsort except on rare occasions.

Lower Bounds

The reader may ask whether there is an algorithm which can sort n items in time of order less than $O(n \log n)$. The answer is no. The reason is indicated below.

Suppose S is an algorithm which sorts n items $a_1, a_2, \ldots, a_n$. We assume there is a *decision tree* T corresponding to the algorithm S such that T is an extended binary search tree where the external nodes correspond to the $n!$ ways that n items can appear in memory and where the internal nodes correspond to the different comparisons that may take place during the execution of the algorithm S. Then the number of comparisons in the worst case for the algorithm S is equal to the length of the longest path in the decision tree T or, in other words, the depth D of the tree, T. Moreover, the average number of comparisons for the algorithm S is equal to the average external path length $\bar{E}$ of the tree T.

Figure 9-1 shows a decision tree T for sorting $n = 3$ items. Observe that T has $n! = 3! = 6$ external nodes. The values of D and $\bar{E}$ for the tree follow:

$$D = 3 \quad \text{and} \quad \bar{E} = \frac{1}{6}(2 + 3 + 3 + 3 + 3 + 2) = 2.667$$

Consequently, the corresponding algorithm S requires at most (worst case) $D = 3$ comparisons and, on the average, $\bar{E} = 2.667$ comparisons to sort the $n = 3$ items.

Accordingly, studying the worst-case and average-case complexity of a sorting algorithm S is reduced to studying the values of D and $\bar{E}$ in the corresponding decision tree T. First, however, we recall some facts about extended binary trees (Sec. 7.11). Suppose T is an extended binary tree with N external nodes, depth D and external path length $E(T)$. Any such tree cannot have more than 2^D external nodes, and so

$$2^D \geq N \quad \text{or equivalently} \quad D \geq \log N$$

Furthermore, T will have a minimum external path length $E(L)$ among all such trees with N nodes when T is a complete tree. In such a case,

$$E(L) = N \log N + O(N) \geq N \log N$$

The $N \log N$ comes from the fact that there are N paths with length $\log N$ or $\log N + 1$, and the $O(N)$ comes from the fact that there are at most N nodes on the deepest level. Dividing $E(L)$ by the number N of external paths gives the average external path length $\bar{E}$. Thus, for any extended binary tree T with N external nodes,

$$\bar{E} = \frac{E(L)}{N} \geq \frac{N \log N}{N} = \log N$$

Now suppose T is the decision tree corresponding to a sorting algorithm S which sorts n items. Then T has $n!$ external nodes. Substituting $n!$ for N in the above formulas yields

$$D \geq \log n! \approx n \log n \quad \text{and} \quad \bar{E} \geq \log n! \approx n \log n$$

The condition $\log n! \approx n \log n$ comes from Stirling's formula, that

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

Thus $n \log n$ is a lower bound for both the worst case and the average case. In other words, $O(n \log n)$ *is the best possible for any sorting algorithm which sorts n items.*

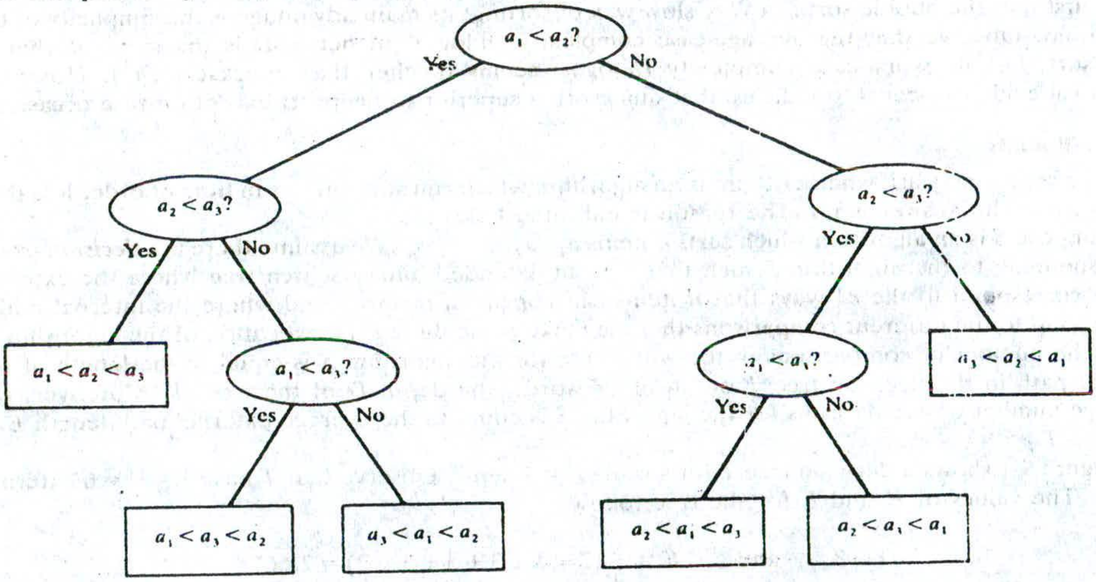


Fig. 9-1 Decision tree T for sorting $n = 3$ items.

Sorting Files; Sorting Pointers

Suppose a file F of records $R_1, R_2, \ldots, R_n$ is stored in memory. "Sorting F" refers to sorting F with respect to some field K with corresponding values $k_1, k_2, \ldots, k_n$. That is, the records are ordered so that

$$k_1 \leq k_2 \leq \cdots \leq k_n$$

The field K is called the *sort key*. (Recall that K is called a *primary key* if its values uniquely determine the records in F.) Sorting the file with respect to another key will order the records in another way.

EXAMPLE 9.2

Suppose the personnel file of a company contains the following data on each of its employees:

Name Social Security Number Sex Monthly Salary

Sorting the file with respect to the Name key will yield a different order of the records than sorting the file with respect to the Social Security Number key. The company may want to sort the file according to the Salary field even though the field may not uniquely determine the employees. Sorting the file with respect to the Sex key will likely be useless; it simply separates the employees into two subfiles, one with the male employees and one with the female employees.

Sorting a file F by reordering the records in memory may be very expensive when the records are very long. Moreover, the records may be in secondary memory, where it is even more time-consuming to move records into different locations. Accordingly, one may prefer to form an auxiliary array POINT containing pointers to the records in memory and then sort the array POINT with respect to a field KEY rather than sorting the records themselves. That is, we sort POINT so that

$$\text{KEY[POINT[1]]} \le \text{KEY[POINT[2]]} \le \cdots \le \text{KEY[POINT[N]]}$$

Note that choosing a different field KEY will yield a different order of the array POINT.

EXAMPLE 9.3

Figure 9-2(a) shows a personnel file of a company in memory. Figure 9-2(b) shows three arrays, POINT, PTRNAME and PTRSSN. The array POINT contains the locations of the records in memory, PTRNAME shows the pointers sorted according to the NAME field, that is,

$$\text{NAME[PTRNAME[1]]} < \text{NAME[PTRNAME[2]]} < \cdots < \text{NAME[PTRNAME[9]]}$$

	NAME	SSN	SEX	SALARY
1				
2	Davis	192-38-7282	Female	22 800
3	Kelly	165-64-3351	Male	19 000
4	Green	175-56-2251	Male	27 200
5				
6	Brown	178-52-1065	Female	14 700
7	Lewis	181-58-9939	Female	16 400
8				
9	Cohen	177-44-4557	Male	19 000
10	Rubin	135-46-6262	Female	15 500
11				
12	Evans	168-56-8113	Male	34 200
13				
14	Harris	208-56-1654	Female	22 800

(a)

	POINT	PTRNAME	PTRSSN
1	2	6	10
2	3	9	3
3	4	2	12
4	6	12	4
5	7	4	9
6	9	14	6
7	10	3	7
8	12	7	2
9	14	10	14

(b)

Fig. 9-2

and PTRSSN shows the pointers sorted according to the SSN field, that is,

$$SSN[PTRSSN[1]] < SSN[PTRSSN[2]] < \cdots < SSN[PTRSSN[9]]$$

Given the name (EMP) of an employee, one can easily find the location of NAME in memory using the array PTRNAME and the binary search algorithm. Similarly, given the social security number NUMB of an employee, one can easily find the location of the employee's record in memory by using the array PTRSSN and the binary search algorithm. Observe, also, that it is not even necessary for the records to appear in successive memory locations. Thus inserting and deleting records can easily be done.

9.3 INSERTION SORT

Suppose an array A with n elements A[1], A[2], . . . , A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2], . . . , A[K−1]. That is:

Pass 1. A[1] by itself is trivially sorted.
Pass 2. A[2] is inserted either before or after A[1] so that: A[1], A[2] is sorted.
Pass 3. A[3] is inserted into its proper place in A[1], A[2], that is, before A[1], between A[1] and A[2], or after A[2], so that: A[1], A[2], A[3] is sorted.
Pass 4. A[4] is inserted into its proper place in A[1], A[2], A[3] so that:
A[1], A[2], A[3], A[4] is sorted.
. . . .
Pass N. A[N] is inserted into its proper place in A[1], A[2], . . . , A[N − 1] so that:
A[1], A[2], . . . , A[N] is sorted.

This sorting algorithm is frequently used when n is small. For example, this algorithm is very popular with bridge players when they are first sorting their cards.

There remains only the problem of deciding how to insert A[K] in its proper place in the sorted subarray A[1], A[2], . . . , A[K −1]. This can be accomplished by comparing A[K] with A[K−1], comparing A[K] with A[K−2], comparing A[K] with A[K−3], and so on, until first meeting an element A[J] such that A[J] ≤ A[K]. Then each of the elements A[K−1], A[K−2], . . . , A[J+1] is moved forward one location, and A[K] is then inserted in the J+1st position in the array.

The algorithm is simplified if there always is an element A[J] such that A[J] ≤ A[K]; otherwise we must constantly check to see if we are comparing A[K] with A[1]. This condition can be accomplished by introducing a sentinel element A[0] = −∞ (or a very small number).

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1:	−∞	$\boxed{77}$	33	44	11	88	22	66	55
K = 2:	−∞	77	$\boxed{33}$	44	11	88	22	66	55
K = 3:	−∞	33	77	$\boxed{44}$	11	88	22	66	55
K = 4:	−∞	33	44	77	$\boxed{11}$	88	22	66	55
K = 5:	−∞	11	33	44	77	$\boxed{88}$	22	66	55
K = 6:	−∞	11	33	44	77	88	$\boxed{22}$	66	55
K = 7:	−∞	11	22	33	44	77	88	$\boxed{66}$	55
K = 8:	∞	11	22	33	44	66	77	88	$\boxed{55}$
Sorted:	−∞	11	22	33	44	55	66	77	88

Fig. 9-3 Insertion sort for $n = 8$ items.

EXAMPLE 9.4

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Figure 9-3 illustrates the insertion sort algorithm. The circled element indicates the A[K] in each pass of the algorithm, and the arrow indicates the proper place for inserting A[K].

The formal statement of our insertion sort algorithm follows.

```
Algorithm 9.1: (Insertion Sort) INSERTION(A, N).
               This algorithm sorts the array A with N elements.
               1. Set A[0] := −∞. [Initializes sentinel element.]
               2. Repeat Steps 3 to 5 for K = 2, 3, . . . . , N:
               3.      Set TEMP := A[K] and PTR := K − 1.
               4.      Repeat while TEMP < A[PTR]:
                           (a) Set A[PTR + 1] := A[PTR]. [Moves element forward.]
                           (b) Set PTR := PTR − 1.
                       [End of loop.]
               5.      Set A[PTR + 1] := TEMP. [Inserts element in proper place.]
                  [End of Step 2 loop.]
               6. Return.
```

Observe that there is an inner loop which is essentially controlled by the variable PTR, and there is an outer loop which uses K as an index.

Complexity of Insertion Sort

The number $f(n)$ of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K − 1 of comparisons. Hence

$$f(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Furthermore, one can show that, on the average, there will be approximately (K − 1)/2 comparisons in the inner loop. Accordingly, for the average case,

$$f(n) = \frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-1}{2} = \frac{n(n-1)}{4} = O(n^2)$$

Thus the insertion sort algorithm is a very slow algorithm when n is very large.

The above results are summarized in the following table:

Algorithm	Worst Case	Average Case
Insertion Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$

Remark: Time may be saved by performing a binary search, rather than a linear search, to find the location in which to insert A[K] in the subarray A[1], A[2], . . . , A[K − 1]. This requires, on the average, log K comparisons rather than (K − 1)/2 comparisons. However, one still needs to move (K − 1)/2 elements forward. Thus the order of complexity is not changed. Furthermore, insertion sort is usually used only when n in small, and in such a case, the linear search is about as efficient as the binary search.

9.4 SELECTION SORT

Suppose an array A with *n* elements A[1], A[2], . . . , A[N] is in memory. The selection sort algorithm for sorting A works as follows. First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on. More precisely:

Pass 1.	Find the location LOC of the smallest in the list of N elements A[1], A[2], . . . , A[N], and then interchange A[LOC] and A[1]. Then: A[1] is sorted.
Pass 2.	Find the location LOC of the smallest in the sublist of N − 1 elements A[2], A[3], . . . , A[N], and then interchange A[LOC] and A[2]. Then: A[1], A[2] is sorted, since $A[1] \le A[2]$.
Pass 3.	Find the location LOC of the smallest in the sublist of N − 2 elements A[3], A[4], . . . , A[N], and then interchange A[LOC] and A[3]. Then: A[1], A[2], . . . , A[3] is sorted, since $A[2] \le A[3]$.
. . .	. . . .
. . .	. . . .
Pass N − 1.	Find the location LOC of the smaller of the elements A[N − 1], A[N], and then interchange A[LOC] and A[N − 1]. Then: A[1], A[2], . . . , A[N] is sorted, since $A[N-1] \le A[N]$.

Thus A is sorted after N − 1 passes.

EXAMPLE 9.5

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Applying the selection sort algorithm to A yields the data in Fig. 9-4. Observe that LOC gives the location of the smallest among A[K], A[K + 1], . . . , A[N] during Pass K. The circled elements indicate the elements which are to be interchanged.

There remains only the problem of finding, during the Kth pass, the location LOC of the smallest among the elements A[K], A[K + 1], . . . , A[N]. This may be accomplished by using a variable MIN to hold the current smallest value while scanning the subarray from A[K] to A[N]. Specifically, first set MIN := A[K] and LOC := K, and then traverse the list, comparing MIN with each other element A[J] as follows:

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1, LOC = 4	$\textcircled{77}$	33	44	$\textcircled{11}$	88	22	66	55
K = 2, LOC = 6	11	$\textcircled{33}$	44	77	88	$\textcircled{22}$	66	55
K = 3, LOC = 6	11	22	$\textcircled{44}$	77	88	$\textcircled{33}$	66	55
K = 4, LOC = 6	11	22	33	$\textcircled{77}$	88	$\textcircled{44}$	66	55
K = 5, LOC = 8	11	22	33	44	$\textcircled{88}$	77	66	$\textcircled{55}$
K = 6, LOC = 7	11	22	33	44	55	$\textcircled{77}$	$\textcircled{66}$	88
K = 7, LOC = 7	11	22	33	44	55	66	$\textcircled{77}$	88
Sorted:	11	22	33	44	55	66	77	88

Fig. 9-4 Selection sort for $n = 8$ items.

(*a*) If MIN ≤ A[J], then simply move to the next element.

(*b*) If MIN > A[J], then update MIN and LOC by setting MIN := A[J] and LOC := J.

After comparing MIN with the last element A[N], MIN will contain the smallest among the elements A[K], A[K + 1], . . . , A[N] and LOC will contain its location.

The above process will be stated separately as a procedure.

Procedure 9.2: MIN(A, K, N, LOC)
An array A is in memory. This procedure finds the location LOC of the smallest element among A[K], A[K + 1], . . . , A[N].

```
1. Set MIN := A[K] and LOC := K. [Initializes pointers.]
2. Repeat for J = K + 1, K + 2, . . . , N:
       If MIN > A[J], then: Set MIN := A[J] and LOC := A[J] and LOC := J.
   [End of loop.]
3. Return.
```

The selection sort algorithm can now be easily stated:

Algorithm 9.3: (Selection Sort) SELECTION(A, N)
This algorithm sorts the array A with N elements.

```
1. Repeat Steps 2 and 3 for K = 1, 2, . . . , N − 1:
2.     Call MIN(A, K, N, LOC).
3.     [Interchange A[K] and A[LOC].]
       Set TEMP := A[K], A[K] := A[LOC] and A[LOC] := TEMP.
   [End of Step 1 loop.]
4. Exit.
```

Complexity of the Selection Sort Algorithm

First note that the number $f(n)$ of comparisons in the selection sort algorithm is independent of the original order of the elements. Observe that MIN(A, K, N, LOC) requires $n - K$ comparisons. That is, there are $n - 1$ comparisons during Pass 1 to find the smallest element, there are $n - 2$ comparisons during Pass 2 to find the second smallest element, and so on. Accordingly,

$$f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

The above result is summarized in the following table:

Algorithm	Worst Case	Average Case
Selection Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$

Remark: The number of interchanges and assignments does depend on the original order of the elements in the array A, but the sum of these operations does not exceed a factor of n^2.

9.5 MERGING

Suppose A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with $n = r + s$ elements is called *merging*. One simple way to merge is to place the elements of B after the elements of A and then use some sorting algorithm on the entire list. This method does not take advantage of the fact that A and B are individually sorted. A much more efficient algorithm is Algorithm 9.4 in this section. First, however, we indicate the general idea of the algorithm by means of two examples.

Suppose one is given two sorted decks of cards. The decks are merged as in Fig. 9-5. That is, at each step, the two front cards are compared and the smaller one is placed in the combined deck. When one of the decks is empty, all of the remaining cards in the other deck are put at the end of the combined deck. Similarly, suppose we have two lines of students sorted by increasing heights, and suppose we want to merge them into a single sorted line. The new line is formed by choosing, at each step, the shorter of the two students who are at the head of their respective lines. When one of the lines has no more students, the remaining students line up at the end of the combined line.

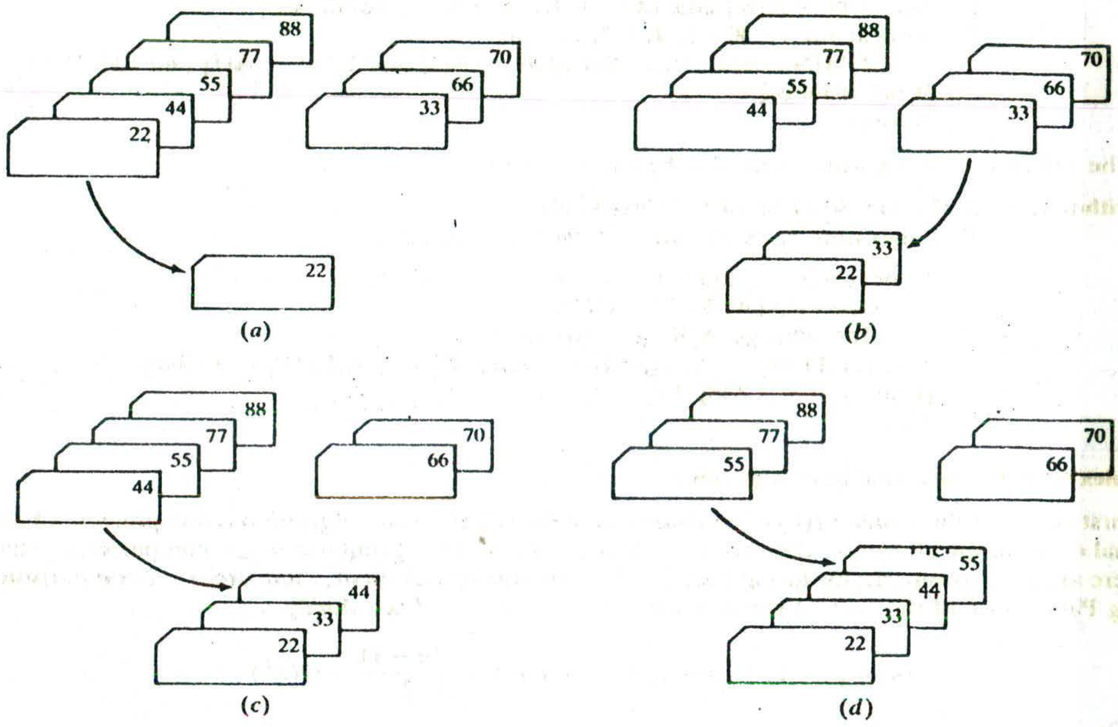


Fig. 9-5

The above discussion will now be translated into a formal algorithm which merges a sorted r-element array A and a sorted s-element array B into a sorted array C, with $n = r + s$ elements. First of all, we must always keep track of the locations of the smallest element of A and the smallest element of B which have not yet been placed in C. Let NA and NB denote these locations, respectively. Also, let PTR denote the location in C to be filled. Thus, initially, we set NA := 1, NB := 1 and PTR := 1. At each step of the algorithm, we compare

$$A[NA] \qquad \text{and} \qquad B[NB]$$

and assign the smaller element to C[PTR]. Then we increment PTR by setting PTR:= PTR + 1, and we either increment NA by setting NA:= NA + 1 or increment NB by setting NB:= NB + 1, according to whether the new element in C has come from A or from B. Furthermore, if NA > r, then the remaining elements of B are assigned to C; or if NB > s, then the remaining elements of A are assigned to C.

The formal statement of the algorithm follows.

```
Algorithm 9.4: MERGING(A, R, B, S, C)
    Let A and B be sorted arrays with R and S elements, respectively. This algorithm
    merges A and B into an array C with N = R + S elements.
    1. [Initialize.] Set NA := 1, NB := 1 and PTR := 1.
    2. [Compare.] Repeat while NA ≤ R and NB ≤ S:
           If A[NA] < B[NB], then:
               (a) [Assign element from A to C.] Set C[PTR] := A[NA].
               (b) [Update pointers.] Set PTR := PTR + 1 and NA := NA + 1.
           Else:
               (a) [Assign element from B to C.] Set C[PTR] := B[NB].
               (b) [Update pointers.] Set PTR := PTR + 1 and NB := NB + 1.
           [End of If structure.]
       [End of loop.]
    3. [Assign remaining elements to C.]
       If NA > R, then:
           Repeat for K = 0, 1, 2, . . . , S − NB:
               Set C[PTR + K] := B[NB + K].
           [End of loop.]
       Else:
           Repeat for K = 0, 1, 2, . . . , R − NA:
               Set C[PTR + K] := A[NA + K].
           [End of loop.]
       [End of If structure.]
    4. Exit.
```

Complexity of the Merging Algorithm

The input consists of the total number $n = r + s$ of elements in A and B. Each comparison assigns an element to the array C, which eventually has n elements. Accordingly, the number $f(n)$ of comparisons cannot exceed n:

$$f(n) \leq n = O(n)$$

In other words, the merging algorithm can be run in linear time.

Nonregular Matrices

Suppose A, B and C are matrices, but not necessarily regular matrices. Assume A is sorted, with r elements and lower bound LBA; B is sorted, with s elements and lower bound LBB; and C has lower bound LBC. Then UBA = LBA + r − 1 and UBB = LBB + s − 1 are, respectively, the upper bounds of A and B. Merging A and B now may be accomplished by modifying the above algorithm as follows.

```
Procedure 9.5: MERGE(A, R, LBA, S, LBB, C, LBC)
    This procedure merges the sorted arrays A and B into the array C.
    1. Set NA := LBA, NB := LBB, PTR := LBC, UBA := LBA + R − 1,
       UBB := LBB + S − 1.
    2. Same as Algorithm 9.4 except R is replaced by UBA and S by UBB.
    3. Same as Algorithm 9.4 except R is replaced by UBA and S by UBB.
    4. Return.
```

Observe that this procedure is called MERGE, whereas Algorithm 9.4 is called MERGING. The reason for stating this special case is that this procedure will be used in the next section, on merge-sort.

Binary Search and Insertion Algorithm

Suppose the number r of elements in a sorted array A is much smaller than the number s of elements in a sorted array B. One can merge A with B as follows. For each element A[K] of A, use a binary search on B to find the proper location to insert A[K] into B. Each such search requires at most $\log s$ comparisons; hence this binary search and insertion algorithm to merge A and B requires at most $r \log s$ comparisons. We emphasize that this algorithm is more efficient than the usual merging Algorithm 9.4 only when $r << s$, that is, when r is much less than s.

EXAMPLE 9.6

Suppose A has 5 elements and suppose B has 100 elements. Then merging A and B by Algorithm 9.4 uses approximately 100 comparisons. On the other hand, only approximately $\log 100 = 7$ comparisons are needed to find the proper place to insert an element of A into B using a binary search. Hence only approximately $5 \cdot 7 = 35$ comparisons are need to merge A and B using the binary search and insertion algorithm.

The binary search and insertion algorithm does not take into account the fact that A is sorted. Accordingly, the algorithm may be improved in two ways as follows. (Here we assume that A has 5 elements and B has 100 elements.)

(1) *Reducing the target set*. Suppose after the first search we find that A[1] is to be inserted after B[16]. Then we need only use a binary search on B[17], . . . , B[100] to find the proper location to insert A[2]. And so on.

(2) *Tabbing*. The expected location for inserting A[1] in B is near B[20] (that is, B[s/r]), not near B[50]. Hence we first use a linear search on B[20], B[40], B[60], B[80] and B[100] to find B[K] such that A[1] ≤ B[K], and then we use a binary search on B[K − 20], B[K − 19], . . . , B[K]. (This is analogous to using the tabs in a dictionary which indicate the location of all words with the same first letter.)

The details of the revised algorithm are left to the reader.

9.6 MERGE-SORT

Suppose an array A with n elements A[1], A[2], . . . , A[N] is in memory. The merge-sort algorithm which sorts A will first be described by means of a specific example.

EXAMPLE 9.7

Suppose the array A contains 14 elements as follows:

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30

Each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs:

$$\underbrace{33,\ 66}\quad \underbrace{22,\ 40}\quad \underbrace{55,\ 88}\quad \underbrace{11,\ 60}\quad \underbrace{20,\ 80}\quad \underbrace{44,\ 50}\quad \underbrace{30,\ 77}$$

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets:

$$\underbrace{22,\ 33,\ 40,\ 66}\quad \underbrace{11,\ 55,\ 60,\ 88}\quad \underbrace{20,\ 44,\ 50,\ 80}\quad \underbrace{30,\ 77}$$

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

$$\underbrace{11,\ 22,\ 33,\ 40,\ 55,\ 60,\ 66,\ 88}\quad \underbrace{20,\ 30,\ 44,\ 50,\ 77,\ 80}$$

Pass 4. Merge the two sorted subarrays to obtain the single sorted array

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88

The original array A is now sorted.

The above merge-sort algorithm for sorting an array A has the following important property. After Pass K, the array A will be partitioned into sorted subarrays where each subarray, except possibly the last, will contain exactly $L = 2^K$ elements. Hence the algorithm requires at most log *n* passes to sort an *n*-element array A.

The above informal description of merge-sort will now be translated into a formal algorithm which will be divided into two parts. The first part will be a procedure MERGEPASS, which uses Procedure 9.5 to execute a single pass of the algorithm; and the second part will repeatedly apply MERGEPASS until A is sorted.

The MERGEPASS procedure applies to an *n*-element array A which consists of a sequence of sorted subarrays. Moreover, each subarray consists of L elements except that the last subarray may have fewer than L elements. Dividing *n* by 2 ∗ L, we obtain the quotient Q, which tells the number of pairs of L-element sorted subarrays; that is,

$$Q = \text{INT}(N/(2*L))$$

(We use INT(X) to denote the integer value of X.) Setting $S = 2*L*Q$, we get the total number S of elements in the Q pairs of subarrays. Hence $R = N - S$ denotes the number of remaining elements. The procedure first merges the initial Q pairs of L-element subarrays. Then the procedure takes care of the case where there is an odd number of subarrays (when $R \le L$) or where the last subarray has fewer than L elements.

The formal statement of MERGEPASS and the merge-sort algorithm follow:

Procedure 9.6: MERGEPASS(A, N, L, B)

The N-element array A is composed of sorted subarrays where each subarray has L elements except possibly the last subarray, which may have fewer than L elements. The procedure merges the pairs of subarrays of A and assigns them to the array B.

```
1.  Set Q := INT(N/(2*L)), S := 2*L*Q and R := N − S.
2.  [Use Procedure 9.5 to merge the Q pairs of subarrays.]
    Repeat for J = 1, 2, . . . , Q:
        (a)  Set LB := 1 + (2*J − 2)*L. [Finds lower bound of first array.]
        (b)  Call MERGE(A, L, LB, A, L, LB + L, B, LB).
    [End of loop.]
3.  [Only one subarray left?]
    If R ≤ L, then:
        Repeat for J = 1, 2, . . . , R:
            Set B(S + J) := A(S + J).
        [End of loop.]
    Else:
        Call MERGE(A, L, S + 1, A, R, L + S + 1, B, S + 1).
    [End of If structure.]
4.  Return.
```

Algorithm 9.7: MERGESORT(A, N)

This algorithm sorts the N-element array A using an auxiliary array B.

```
1.  Set L := 1. [Initializes the number of elements in the subarrays.]
2.  Repeat Steps 3 to 6 while L < N:
3.      Call MERGEPASS(A, N, L, B).
4.      Call MERGEPASS(B, N, 2 * L, A).
5.      Set L := 4 * L.
    [End of Step 2 loop.]
6.  Exit.
```

Since we want the sorted array to finally appear in the original array A, we must execute the procedure MERGEPASS an even number of times.

Complexity of the Merge-Sort Algorithm

Let $f(n)$ denote the number of comparisons needed to sort an n-element array A using the merge-sort algorithm. Recall that the algorithm requires at most $\log n$ passes. Moreover, each pass merges a total of n elements, and by the discussion on the complexity of merging, each pass will require at most n comparisons. Accordingly, for both the worst case and average case,

$$f(n) \le n \log n$$

Observe that this algorithm has the same order as heapsort and the same average order as quicksort. The main drawback of merge-sort is that it requires an auxiliary array with n elements. Each of the other sorting algorithms we have studied requires only a finite number of extra locations, which is independent of n.

The above results are summarized in the following table:

Algorithm	Worst Case	Average Case	Extra Memory
Merge-Sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$

9.7 RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labeled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresonds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the units digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit. We illustrate with an example.

EXAMPLE 9.8

Suppose 9 cards are punched as follows:

348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases, as pictured in Fig. 9-6:

(*a*) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(*b*) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(*a*) First pass.

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(*b*) Second pass.

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(*c*) Third pass

Fig. 9-6

(*c*) In the third and final pass, the hundreds digits are sorted into pockets.

When the cards are collected after the third pass, the numbers are in the following order:

$$128,\ 143,\ 321,\ 348,\ 361,\ 366,\ 423,\ 538,\ 543$$

Thus the cards are now sorted.

The number C of comparisons needed to sort nine such 3-digit numbers is bounded as follows:

$$C \le 9*3*10$$

The 9 comes from the nine cards, the 3 comes from the three digits in each number, and the 10 comes from radix $d = 10$ digits.

Complexity of Radix Sort

Suppose a list A of n items $A_1, A_2, \ldots, A_n$ is given. Let d denote the radix (e.g., $d = 10$ for decimal digits, $d = 26$ for letters and $d = 2$ for bits), and suppose each item A_i is represented by means of s of the digits:

$$A_i = d_{i1}d_{i2}\cdots d_{is}$$

The radix sort algorithm will require s passes, the number of digits in each item. Pass K will compare each d_{iK} with each of the d digits. Hence the number $C(n)$ of comparisons for the algorithm is bounded as follows:

$$C(n) \le d*s*n$$

Although d is independent of n, the number s does depend on n. In the worst case, $s = n$, so $C(n) = O(n^2)$. In the best case, $s = \log_d n$, so $C(n) = O(n \log n)$. In other words, radix sort performs well only when the number s of digits in the representation of the A_i's is small.

Another drawback of radix sort is that one may need $d*n$ memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require $2*n$ memory locations.

9.8 SEARCHING AND DATA MODIFICATION

Suppose S is a collection of data maintained in memory by a table using some type of data structure. Searching is the operation which finds the location LOC in memory of some given ITEM of information or sends some message that ITEM does not belong to S. The search is said to be successful or unsuccessful according to whether ITEM does or does not belong to S. The searching algorithm that is used depends mainly on the type of data structure that is used to maintain S in memory.

Data modification refers to the operations of inserting, deleting and updating. Here data modification will mainly refer to inserting and deleting. These operations are closely related to searching, since usually one must search for the location of the ITEM to be deleted or one must search for the proper place to insert ITEM in the table. The insertion or deletion also requires a certain amount of execution time, which also depends mainly on the type of data structure that is used.

Generally speaking, there is a tradeoff between data structures with fast searching algorithms and data structures with fast modification algorithms. This situation is illustrated below, where we summarize the searching and data modification of three of the data structures previously studied in the text.

(1) *Sorted array*. Here one can use a binary search to find the location LOC of a given ITEM in time $O(\log n)$. On the other hand, inserting and deleting are very slow, since, on the average, $n/2 = O(n)$ elements must be moved for a given insertion or deletion. Thus a sorted array would likely be used when there is a great deal of searching but only very little data modification.

(2) *Linked list.* Here one can only perform a linear search to find the location LOC of a given ITEM, and the search may be very, very slow, possibly requiring time $O(n)$. On the other hand, inserting and deleting requires only a few pointers to be changed. Thus a linked list would be used when there is a great deal of data modification, as in word (string) processing.

(3) *Binary search tree.* This data structure combines the advantages of the sorted array and the linked list. That is, searching is reduced to searching only a certain path P in the tree T, which, on the average, requires only $O(\log n)$ comparisons. Furthermore, the tree T is maintained in memory by a linked representation, so only certain pointers need be changed after the location of the insertion or deletion is found. The main drawback of the binary search tree is that the tree may be very unbalanced, so that the length of a path P may be $O(n)$ rather than $O(\log n)$. This will reduce the searching to approximately a linear search.

Remark: The above worst-case scenario of a binary search tree may be eliminated by using a height-balanced binary search tree that is rebalanced after each insertion or deletion. The algorithms for such rebalancing are rather complicated and lie beyond the scope of this text.

Searching Files, Searching Pointers

Suppose a file F of records $R_1, R_2, \ldots, R_N$ is stored in memory. Searching F usually refers to finding the location LOC in memory of the record with a given key value relative to a primary key field K. One way to simplify the searching is to use an auxiliary sorted array of pointers, as discussed in Sec. 9.2. Then a binary search can be used to quickly find the location LOC of the record with the given key. In the case where there is a great deal of inserting and deleting of records in the file, one might want to use an auxiliary binary search tree rather than an auxiliary sorted array. In any case, the searching of the file F is reduced to the searching of a collection S of items, as discussed above.

9.9 HASHING

The search time of each algorithm discussed so far depends on the number n of elements in the collection S of data. This section discusses a searching technique, called *hashing* or *hash addressing*, which is essentially independent of the number n.

The terminology which we use in our presentation of hashing will be oriented toward file management. First of all, we assume that there is a file F of n records with a set K of keys which uniquely determine the records in F. Secondly, we assume that F is maintained in memory by a table T of m memory locations and that L is the set of memory addresses of the locations in T. For notational convenience, we assume that the keys in K and the addresses in L are (decimal) integers. (Analogous methods will work with binary integers or with keys which are character strings, such as names, since there are standard ways of representing strings by integers.)

The subject of hashing will be introduced by the following example.

EXAMPLE 9.9

Suppose a company with 68 employees assigns a 4-digit employee number to each employee which is used as the primary key in the company's employee file. We can, in fact, use the employee number as the address of the record in memory. The search will require no comparisons at all. Unfortunately, this technique will require space for 10 000 memory locations, whereas space for fewer than 30 such locations would actually be used. Clearly, this tradeoff of space for time is not worth the expense.

The general idea of using the key to determine the address of a record is an excellent idea, but it must be modified so that a great deal of space is not wasted. This modification takes the form of a function H from the set K of keys into the set L of memory addresses. Such a function,

$$H: K \to L$$

is called a *hash function* or *hashing function*. Unfortunately, such a function H may not yield distinct values: it is possible that two different keys k_1 and k_2 will yield the same hash address. This situation is called *collision*, and some method must be used to resolve it. Accordingly, the topic of hashing is divided into two parts: (1) hash functions and (2) collision resolutions. We discuss these two parts separately.

Hash Functions

The two principal criteria used in selecting a hash function $H: K \rightarrow L$ are as follows. First of all, the function H should be very easy and quick to compute. Second the function H should, as far as possible, uniformly distribute the hash addresses throughout the set L so that there are a minimum number of collisions. Naturally, there is no guarantee that the second condition can be completely fulfilled without actually knowing beforehand the keys and addresses. However, certain general techniques do help. One technique is to "chop" a key k into pieces and combine the pieces in some way to form the hash address $H(k)$. (The term "hashing" comes from this technique of "chopping" a key into pieces.)

We next illustrate some popular hash functions. We emphasize that each of these hash functions can be easily and quickly evaluated by the computer.

(*a*) *Division method*. Choose a number m larger than the number n of keys in K. (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions.) The hash function H is defined by

$$H(k) = k \pmod{m} \qquad \text{or} \qquad H(k) = k \pmod{m} + 1$$

Here $k \pmod{m}$ denotes the remainder when k is divided by m. The second formula is used when we want the hash addresses to range from 1 to m rather than from 0 to $m-1$.

(*b*) *Midsquare method*. The key k is squared. Then the hash function H is defined by

$$H(k) = l$$

where l is obtained by deleting digits from both ends of k^2. We emphasize that the same positions of k^2 must be used for all of the keys.

(*c*) *Folding method*. The key k is partitioned into a number of parts, $k_1, \ldots, k_r$, where each part, except possibly the last, has the same number of digits as the required address. Then the parts are added together, ignoring the last carry. That is,

$$H(k) = k_1 + k_2 + \cdots + k_r$$

where the leading-digit carries, if any, are ignored. Sometimes, for extra "milling," the even-numbered parts, $k_2, k_4, \ldots$, are each reversed before the addition.

EXAMPLE 9.10

Consider the company in Example 9.9, each of whose 68 employees is assigned a unique 4-digit employee number. Suppose L consists of 100 two-digit addresses: 00, 01, 02, . . . , 99. We apply the above hash functions to each of the following employee numbers:

$$3205, \qquad 7148, \qquad 2345$$

(*a*) *Division method*. Choose a prime number m close to 99, such as $m = 97$. Then

$$H(3205) = 4, \qquad H(7148) = 67, \qquad H(2345) = 17$$

That is, dividing 3205 by 97 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, and dividing 2345 by 97 gives a remainder of 17. In the case that the memory addresses begin with 01 rather than 00, we choose that the function $H(k) = k \pmod{m} + 1$ to obtain:

$$H(3205) = 4 + 1 = 5, \qquad H(7148) = 67 + 1 = 68, \qquad H(2345) = 17 + \quad 18$$

(*b*) *Midsquare method.* The following calculations are performed:

k:	3205	7148	2345
k^2:	10 272 025	51 093 904	5 499 025
$H(k)$:	72	93	99

Observe that the fourth and fifth digits, counting from the right, are chosen for the hash address.

(*c*) *Folding method.* Chopping the key k into two parts and adding yields the following hash addresses:

$$H(3205) = 32 + 05 = 37, \qquad H(7148) = 71 + 48 = 19, \qquad H(2345) = 23 + 45 = 68$$

Observe that the leading digit 1 in $H(7148)$ is ignored. Alternatively, one may want to reverse the second part before adding, thus producing the following hash addresses:

$$H(3205) = 32 + 50 = 82, \qquad H(7148) = 71 + 84 = 155, \qquad H(2345) = 23 + 54 = 77$$

Collision Resolution

Suppose we want to add a new record R with key k to our file F, but suppose the memory location address $H(k)$ is already occupied. This situation is called *collision*. This subsection discusses two general ways of resolving collisions. The particular procedure that one chooses depends on many factors. One important factor is the ratio of the number n of keys in K (which is the number of records in F) to the number m of hash addresses in L. This ratio, $\lambda = n/m$, is called the *load factor*.

First we show that collisions are almost impossible to avoid. Specifically, suppose a student class has 24 students and suppose the table has space for 365 records. One random hash function is to choose the student's birthday as the hash address. Although the load factor $\lambda = 24/365 \approx 7\%$ is very small, it can be shown that there is a better than fifty-fifty chance that two of the students have the same birthday.

The efficiency of a hash function with a collision resolution procedure is measured by the average number of *probes* (key comparisons) needed to find the location of the record with a given key k. The efficiency depends mainly on the load factor λ. Specifically, we are interested in the following two quantities:

$$S(\lambda) = \text{average number of probes for a successful search}$$
$$U(\lambda) = \text{average number of probes for an unsuccessful search}$$

These quantities will be discussed for our collision procedures.

Open Addressing: Linear Probing and Modifications

Suppose that a new record R with key k is to be added to the memory table T, but that the memory location with hash address $H(k) = h$ is already filled. One natural way to resolve the collision is to assign R to the first available location following $T[h]$. (We assume that the table T with m locations is circular, so that $T[1]$ comes after $T[m]$.) Accordingly, with such a collision procedure, we will search for the record R in the table T by linearly searching the locations $T[h]$, $T[h+1]$, $T[h+2]$, ... until finding R or meeting an empty location, which indicates an unsuccessful search.

The above collision resolution is called *linear probing*. The average numbers of probes for a successful search and for an unsuccessful search are known to be the following respective quantities:

$$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \quad \text{and} \quad U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

(Here $\lambda = n/m$ is the load factor.)

EXAMPLE 9.11

Suppose the table T has 11 memory locations, $T[1]$, $T[2]$, ..., $T[11]$, and suppose the file F consists of 8 records, A, B, C, D, E, X, Y and Z, with the following hash addresses:

Record:	A,	B,	C,	D,	E,	X,	Y,	Z
$H(k)$:	4,	8,	2,	11,	4,	11,	5,	1

Suppose the 8 records are entered into the table T in the above order. Then the file F will appear in memory as follows:

Table T:	X,	C,	Z,	A,	E,	Y,	__,	B,	__,	__,	D
Address:	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11

Although Y is the only record with hash address $H(k) = 5$, the record is not assigned to $T[5]$, since $T[5]$ has already been filled by E because of a previous collision at $T[4]$. Similarly, Z does not appear in $T[1]$.

The average number S of probes for a successful search follows:

$$S = \frac{1+1+1+1+2+2+2+3}{8} = \frac{13}{8} \approx 1.6$$

The average number U of probes for an unsuccessful search follows:

$$U = \frac{7+6+5+4+3+2+1+2+1+1+8}{11} = \frac{40}{11} \approx 3.6$$

The first sum adds the number of probes to find each of the 8 records, and the second sum adds the number of probes to find an empty location for each of the 11 locations.

One main disadvantage of linear probing is that records tend to *cluster*, that is, appear next to one another, when the load factor is greater than 50 percent. Such a clustering substantially increases the average search time for a record. Two techniques to minimize clustering are as follows:

(1) *Quadratic probing*. Suppose a record R with key k has the hash address $H(k) = h$. Then, instead of searching the locations with addresses $h, h+1, h+2, \ldots$, we linearly search the locations with addresses

$$h,\ h+1,\ h+4,\ h+9,\ h+16,\ \ldots,\ h+i^2,\ \ldots$$

If the number m of locations in the table T is a prime number, then the above sequence will access half of the locations in T.

(2) *Double hashing*. Here a second hash function H' is used for resolving a collision, as follows. Suppose a record R with key k has the hash addresses $H(k) = h$ and $H'(k) = h' \neq m$. Then we linearly search the locations with addresses

$$h,\ h+h',\ h+2h',\ h+3h',\ \ldots$$

If m is a prime number, then the above sequence will access all the locations in the table T.

Remark: One major disadvantage in any type of open addressing procedure is in the implementation of deletion. Specifically, suppose a record R is deleted from the location $T[r]$. Afterwards, suppose we meet $T[r]$ while searching for another record R'. This does not necessarily mean that the search is unsuccessful. Thus, when deleting the record R, we must label the location $T[r]$ to indicate that it previously did contain a record. Accordingly, open addressing may seldom be used when a file F is constantly changing.

Chaining

Chaining involves maintaining two tables in memory. First of all, as before, there is a table T in memory which contains the records in F, except that T now has an additional field LINK which is used so that all records in T with the same hash address h may be linked together to form a linked list. Second, there is a hash address table LIST which contains pointers to the linked lists in T.

Suppose a new record R with key k is added to the file F. We place R in the first available location in the table T and then add R to the linked list with pointer LIST[$H(k)$]. If the linked lists of records are not sorted, than R is simply inserted at the beginning of its linked list. Searching for a record or deleting a record is nothing more than searching for a node or deleting a node from a linked list, as discussed in Chap. 5.

The average number of probes, using chaining, for a successful search and for an unsuccessful search are known to be the following approximate values:

$$S(\lambda) \approx 1 + \frac{1}{2}\lambda \qquad \text{and} \qquad U(\lambda) \approx e^{-\lambda} + \lambda$$

Here the load factor $\lambda = n/m$ may be greater than 1, since the number m of hash addresses in L (not the number of locations in T) may be less than the number n of records in F.

EXAMPLE 9.12

Consider again the data in Example 9.11, where the 8 records have the following hash addresses:

Record:	A,	B,	C,	D,	E,	X,	Y,	Z
$H(k)$:	4,	8,	2,	11,	4,	11,	5,	1

Using chaining, the records will appear in memory as pictured in Fig. 9-7. Observe that the location of a record R in table T is not related to its hash address. A record is simply put in the first node in the AVAIL list of table T. In fact, table T need not have the same number of elements as the hash address table.

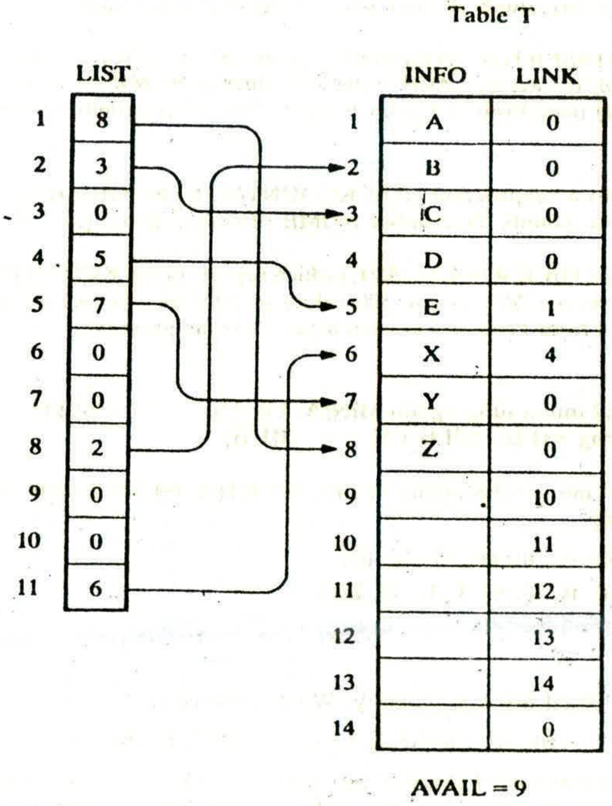


Fig. 9-7

The main disadvantage to chaining is that one needs $3m$ memory cells for the data. Specifically, there are m cells for the information field INFO, there are m cells for the link field LINK, and there are m cells for the pointer array LIST. Suppose each record requires only 1 word for its information field. Then it may be more useful to use open addressing with a table with $3m$ locations, which has the load factor $\lambda \leq 1/3$, than to use chaining to resolve collisions.

Supplementary Problems

SORTING

9.1 Write a subprogram RANDOM(DATA, N, K) which assigns N random integers between 1 and K to the array DATA.

9.2 Translate insertion sort into a subprogram INSERTSORT(A, N) which sorts the array A with N elements. Test the program using:

(*a*) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

(*b*) D, A, T, A, S, T, R, U, C, T, U, R, E, S

9.3 Translate insertion sort into a subprogram INSERTCOUNT(A, N, NUMB) which sorts the array A with N elements and which also counts the number NUMB of comparisons.

9.4 Write a program TESTINSERT(N, AVE) which repeats 500 times the procedure INSERTCOUNT(A, N, NUMB) and which finds the average AVE of the 500 values of NUMB. (Theoretically, AVE $\approx N^2/4$.) Use RANDOM(A, N, 5*N) from Prob. 9.1 as each input. Test the program using N = 100 (so, theoretically, AVE $\approx N^2/4 = 2500$).

9.5 Translate quicksort into a subprogram QUICKCOUNT(A, N, NUMB) which sorts the array A with N elements and which also counts the number NUMB of comparisons. (See Sec. 6.5.)

9.6 Write a program TESTQUICKSORT(N, AVE) which repeats QUICKCOUNT(A, N, NUMB) 500 times and which finds the average AVE of the 500 values of NUMB. (Theoretically, AVE $\approx N \log_2 N$.) Use RANDOM(A, N, 5*N) from Prob. 9.1 as each input. Test the program using N = 100 (so, theoretically, AVE ≈ 700).

9.7 Translate Procedure 9.2 into a subprogram MIN(A, LB, UB, LOC) which finds the location LOC of the smallest elements among A[LB], A[LB + 1], . . . , A[UB].

9.8 Translate selection sort into a subprogram SELECTSORT(A, N) which sorts the array with N elements. Test the program using:

(*a*) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

(*b*) D, A, T, A, S, T, R, U, C, T, U, R, E, S

SEARCHING, HASHING

9.9 Suppose an unsorted linked list is in memory. Write a procedure

SEARCH(INFO, LINK, START, ITEM, LOC)

which (*a*) finds the location LOC of ITEM in the list or sets LOC := NULL for an unsuccessful search and (*b*) when the search is successful, interchanges ITEM with the element in front of it. (Such a list is said to be *self-organizing*. It has the property that elements which are frequently accessed tend to move to the beginning of the list.)

9.10 Consider the following 4-digit employee numbers (see Example 9.10):

9614, 5882, 6713, 4409, 1825

Find the 2-digit hash address of each number using (*a*) the division method, with $m = 97$; (*b*) the midsquare method; (*c*) the folding method without reversing; and (*d*) the folding method with reversing.

9.11 Consider the data in Example 9.11. Suppose the 8 records are entered into the table *T* in the reverse order Z, Y, X, E, D, C, B, A. (*a*) Show how the file *F* appears in memory. (*b*) Find the average number *S* of probes for a successful search and the average number *U* of probes for an unsuccessful search. (Compare with the corresponding results in Example 9.11.)

9.12 Consider the data in Example 9.12 and Fig. 9-7. Suppose the following additional records are added to the file:

(P, 2), (Q, 7), (R, 4), (S, 9)

(Here the left entry is the record and the right entry is the hash address.) (*a*) Find the updated tables T and LIST. (*b*) Find the average number *S* of probes for a successful search and the average number *U* of probes for an unsuccesful search.

9.13 Write a subprogram MID(KEY, HASH) which uses the midsquare method to find the 2-digit hash address HASH of a 4-digit employee number key.

9.14 Write a subprogram FOLD(KEY, HASH) which uses the folding method with reversing to find t'.. 2-digit hash address HASH of a 4-digit employee number key.