Observe that the output consists of the following seven moves:

$$A \to C, \quad A \to B, \quad C \to B, \quad A \to C, \quad B \to A, \quad B \to C, \quad A \to$$

This agrees with the solution in Fig. 6.15.

Summary

The Towers of Hanoi problem illustrates the power of recursion in the solution of various alg problems. This section has shown how to implement recursion by means of stacks when programming language—notably FORTRAN or COBOL—which does not allow recursive pr In fact, even when using a programming language—such as Pascal—which does support re the programmer may want to use the nonrecursive solution, since it may be much less ex than using the recursive solution.

6.10 QUEUES

A *queue* is a linear list of elements in which deletions can take place only at one end, call *front*, and insertions can take place only at the other end, called the *rear*. The terms "fro "rear" are used in describing a linear list only when it is implemented as a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will first element out of the queue. In other words, the order in which elements enter a queue order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists

Queues abound in everyday life. The automobiles waiting to pass through an intersection queue, in which the first car in line is the first car through; the people waiting in line at form a queue, where the first person in line is the first person to be waited on; and so important example of a queue in computer science occurs in a timesharing system, in programs with the same priority form a queue while waiting to be executed. (Another str called a priority queue, is discussed in Sec. 6.13.)

Example 6.10

Figure 6.19(a) is a schematic diagram of a queue with 4 elements; where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are also, respectively, the first and last elements of the list. Suppose an element is deleted from the queue. Then it must be AAA. This yields the queue in Fig. 6.19(b), where BBB is now the front element. Next, suppose EEE is added to the queue and then FFF is added to the queue. Then they must be added at the rear of the queue, as pictured in Fig. 6.19(c). Note that FFF is now the rear element. Now suppose another element is deleted from the queue; then it must be BBB, to yield the queue in Fig. 6.19(d). And so on. Observe that in such a data structure, EEE will be deleted before FFF because it has been placed in the queue before FFF. However, EEE will have to wait until CCC and DDD are deleted.

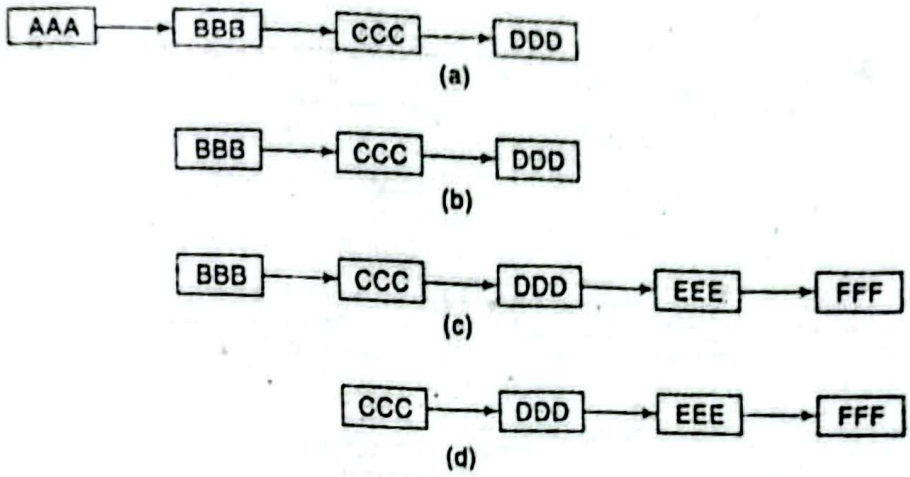


Fig. 6.19

epresentation of Queues

ues may be represented in the computer in various ways, usually by means at one-way lists or ear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear ay QUEUE and two pointer variables: FRONT, containing the location of the front element of queue; and REAR, containing .the location of the rear element of the queue. The condition ONT = NULL will indicate that the queue is empty.

Figure 6.20 shows the way the array in Fig. 6.19 will be stared in memory using an array UEUE with N elements. Figure 6.20 also indicates the way elements will be deleted from the ue and the way new elements will be added to the queue. Observe that whenever an element is eted from the queue, the value of FRONT is increased by 1; this can be implemented by the ignment

FRONT := FRONT + 1

milarly, whenever an element is added to the queue, the value of REAR is increased by 1; this be implemented by the assignment

REAR := REAR + 1

is means that after N insertions, the rear element of the queue will occupy QUEUE[N] or, in er words; eventually the queue will occupy the last part of the array. This occurs even though queue itself may not contain many elements.

Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the t part of the array, i.e., when REAR = N. One way to do this is to simply move the entire queue the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM above. This procedure may be very expensive. The procedure we adopt is to assume that the ay QUEUE is circular, that is, that QUEUE[1] comes after QUEUE[N] in the array. With this umption, we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Specifically, instead increasing REAR to N + 1, we reset REAR = 1 and then assign

QUEUE[REAR] := ITEM

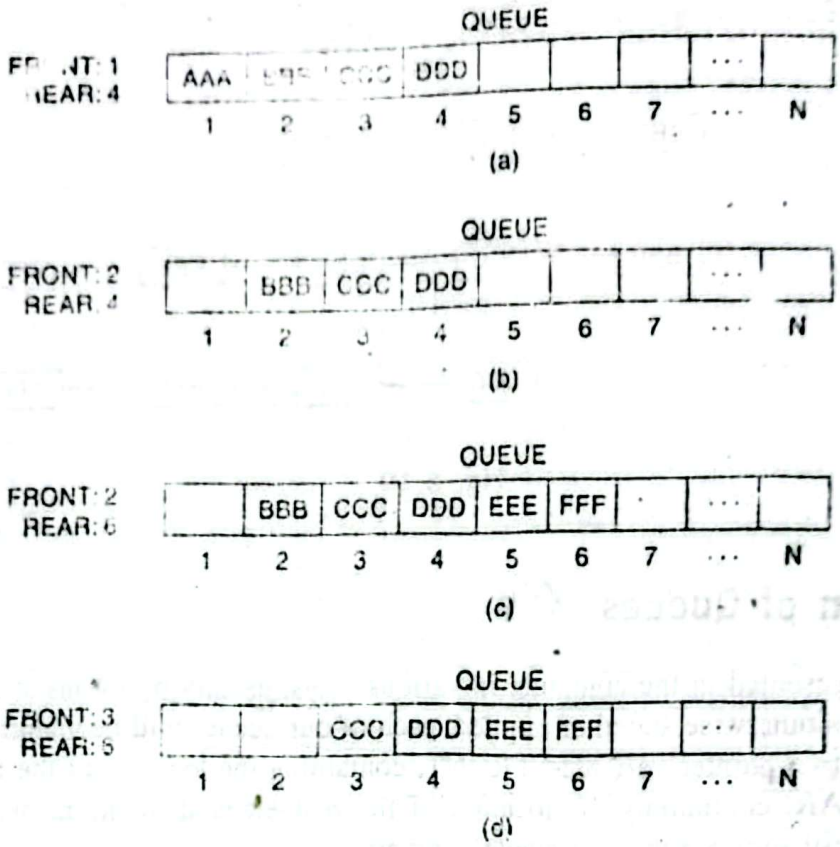


Fig. 6.20 *Array Representation of a Queue*

Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 i increasing FRONT to N + 1. (Some readers may recognize this as modular arithmetic. dis Sec. 2.2.)

Suppose that our queue contains only one element, i.e., suppose that

$$\text{FRONT} = \text{REAR} \neq \text{NULL}$$

and suppose that the element is deleted. Then we assign

$$\text{FRONT} := \text{NULL} \quad \text{and} \quad \text{REAR} := \text{NULL}$$

to indicate that the queue is empty.

Example 6.11

Figure 6.21 shows how a queue may be maintained by a circular array QUEUE with $N = 5$ memory locations. Observe that the queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that it still occupies consecutive locations. Also, as indicated by Fig. 6.21(m), the queue will be empty only when FRONT = REAR and an element is deleted. For this reason, NULL is assigned to FRONT and REAR in Fig. 6.21(m).

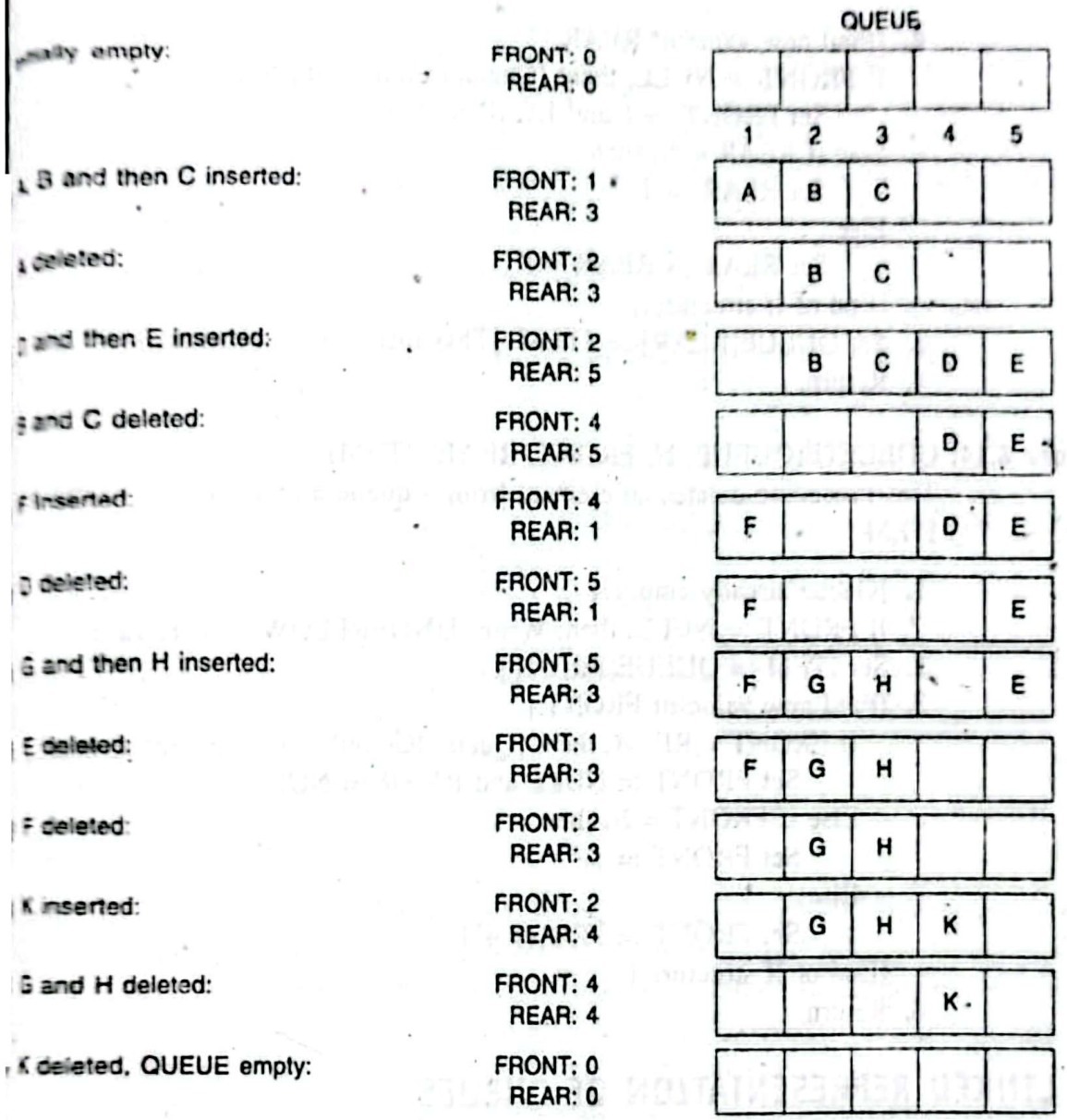


Fig. 6.21

We are now prepared to formally state our procedure QINSERT (Procedure 6.13), which inserts a data ITEM into a queue. The first thing we do in the procedure is to test for overflow, that is, to test whether or not the queue is filled.

Next we give a procedure QDELETE (Procedure 6.14), which deletes the first element from a queue, assigning it to the variable ITEM. The first thing we do is to test for underflow, i.e., to test whether or not the queue is empty.

Procedure 6.13: QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
   Write: OVERFLOW, and Return.

2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
      Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
      Set REAR := 1.
   Else:
      Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.

Procedure 6.14: QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the ITEM.

1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
   If FRONT = REAR, then: [Queue has only one element to s
      Set FRONT := NULL and REAR := NULL.
   Else if FRONT = N, then:
      Set FRONT := 1.
   Else:
      Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.

6.11 LINKED REPRESENTATION OF QUEUES

In this section we discuss the linked representation of a queue. A linked queue is implemented as a linked list with two pointer variables FRONT and REAR pointing to which is in the FRONT and REAR of the queue. The INFO fields of the list hold the d the queue and the LINK fields hold pointers to the neighboring elements in the queue illustrates the linked representation of the queue shown in Fig. 6.16(a).

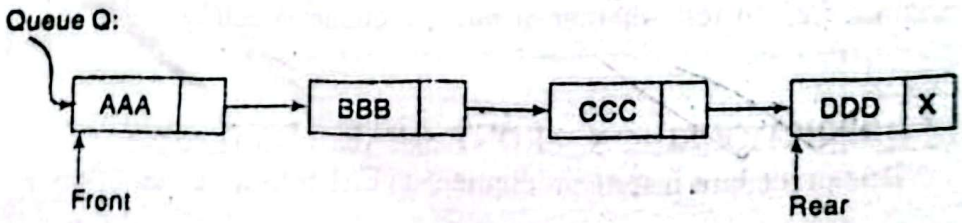
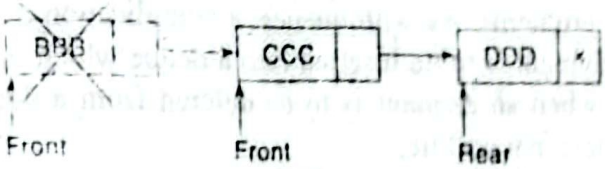


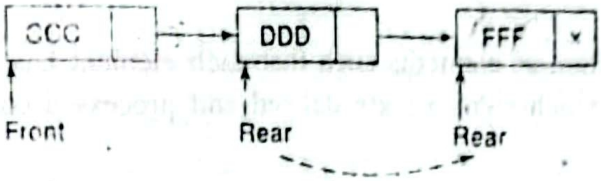
Fig. 6.22

(ii) Delete



(c)

(iii) Insert FFF



(d)

Fig. 6.25

DEQUES

A deque (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name *double-ended queue*.

There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array. Figure 6.26 pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two variations of a deque—namely, an input-restricted deque and an output-restricted deque—which are intermediate between a deque and a queue. Specifically, an *input-restricted deque* is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list; and an *output-restricted deque* is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

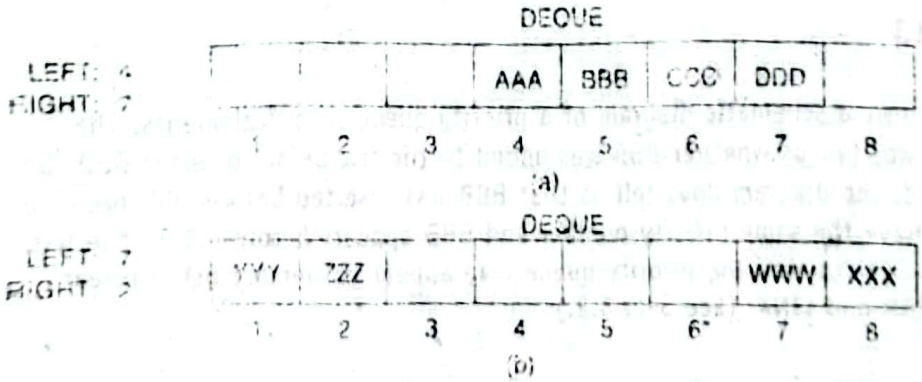


Fig. 6.26

The procedures which insert and delete elements in deques and the variations on those pr are given as supplementary problems. As with queues, a complication may arise (a) when overflow, that is, when an element is to be inserted into a deque which is already full, or there is underflow, that is, when an element is to be deleted from a deque which is em procedures must consider these possibilities.

6.13 PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has been assigned a p and such that the order in which elements are deleted and processed comes from the fo rules:

(1) An element of higher priority is processed before any element of lower priority.
(2) Two elements with the same priority are processed according to the order in wh were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are pr first, and programs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. We discuss two o here: one uses a one-way list, and the other uses multiple queues. The ease or difficulty in r elements to or deleting them from a priority queue clearly depends on the representation th chooses.

One-Way List Representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows

(a) Each node in the list will contain three items of information: an information field IN priority number PRN and a link number LINK.
(b) A node X precedes a node Y in the list (1) when X has higher priority than Y or (2 both have the same priority but X was added to the list before Y. This means that th in the one-way list corresponds to the order of the priority queue.

Priority numbers will operate in the usual way: the lower the priority number, the high priority.

Example 6.13

Figure 6.27 shows a schematic diagram of a priority queue with 7 elements. The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list. Figure 6.28 shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK. (See Sec. 5.2.)

(2) Translation of "Step K. Call P."

(a) Push the current values of the parameters and local variables and the cur return address ADD onto the appropriate stacks.

(b) Reset the parameters using the new argument values, and set ADD := [Step] 1.

(c) Go to Step 1. [The beginning of the procedure P.]

(3) Translation of "Step J. Return."

(a) If ADD = Main, then: Return. [Control is transferred to the main program.]

(b) Set SAVE := ADD.

(c) Restore the top values of the stacks. That is, set the parameters and local variab equal to the top values on the stacks, and set ADD equal to the top value on stack STADD.

(d) Go to Step SAVE.

(Compare this translation algorithm with the algorithm in Sec. 6.9.)

Queues, Deques

6.22 Consider the following queue of characters, where QUEUE is a circular array which is allocated six memory cells:

FRONT = 2, REAR = 4 QUEUE: ___, A, C, D, ___, ___

(For notational convenience, we use "___" to denote an empty memory cell.) Describe the queue as the following operations take place:

(a) F is added to the queue.
(b) two letters are deleted.
(c) K, L and M are added to the queue.
(d) two letters are deleted.
(e) R is added to the queue.
(f) two letters are deleted.
(g) S is added to the queue.
(h) two letters are deleted.
(i) one letter is deleted.
(j) one letter is deleted.

(a) F is added to the rear of the queue, yielding

FRONT = 2, REAR = 5 QUEUE: ___, A, C, D, F, ___

Note that REAR is increased by 1.

(b) The two letters, A and C, are deleted, leaving

FRONT = 4, REAR = 5 QUEUE: ___, ___, ___, D, F, ___

Note that FRONT is increased by 2.

(c) K, L and M are added to the rear of the queue. Since K is placed in the last memory cell of QUEUE, L and M are placed in the first two memory cells. This yields

FRONT = 4, REAR = 2 QUEUE: L, M, ___, D, F, K

Note that REAR is increased by 3 but the arithmetic is modulo 6:

REAR = 5 + 3 = 8 = 2 (mod 6)

(d) The two front letters, D and F are deleted, leaving

FRONT = 6, REAR = 2 QUEUE: L, M, ___, ___, ___, K

(e) R is added to the rear of the queue, yielding

FRONT = 6, REAR = 3 QUEUE: L, M, R, ___, ___, K

(f) The two front letters, K and L, are deleted, leaving

FRONT = 2, REAR = 3 QUEUE: ___, M, R, ___, ___, ___

Note that FRONT is increased by 2 but the arithmetic is modulo 6:

FRONT = 6 + 2 = 8 = 2 (mod 6)

(g) S is added to the rear of the queue, yielding

FRONT = 2, REAR = 4 QUEUE: ___, M, R, S, ___, ___

(h) The two front letters, M and R, are deleted, leaving

FRONT = 4, REAR = 4 QUEUE: ___, ___, ___, S, ___, ___

(i) The front letter S is deleted. Since FRONT = REAR, this means that the queue is empty; hence we assign NULL to FRONT and REAR. Thus

FRONT = 0, REAR = 0 QUEUE: ___, ___, ___, ___, ___, ___

(j) Since FRONT = NULL, no deletion can take place. That is, underflow has occurred.

Suppose each data structure is stored in a circular array with N memory cells.

(a) Find the number NUMB of elements in a queue in terms of FRONT and REAR.
(b) Find the number NUMB of elements in a deque in terms of LEFT and RIGHT.
(c) When will the array be filled?

(a) If FRONT ≤ REAR, then NUMB = REAR – FRONT + 1. For example, consider the following queue with N = 12:

FRONT = 3, REAR = 9 QUEUE: ___, ___, *, *, *, *, *, *, *, ___, ___, ___

Then NUMB = 9 – 3 + 1 = 7, as pictured.

If REAR < FRONT, then FRONT – REAR – 1 is the number of empty cells, so

NUMB = N – (FRONT – REAR – 1) = N + REAR – FRONT + 1

For example, consider the following queue with N = 12:

FRONT = 9, REAR = 4 QUEUE: *, *, *, *, ___, ___, ___, ___, *, *, *, *

Then NUMB = 12 + 4 – 9 + 1 = 8, as pictured.

Using arithmetic modulo N, we need only one formula, as follows:

NUMB = REAR – FRONT + 1 (mod N)

(b) The same result holds for deques except that FRONT is replaced by RIGHT. That is,

NUMB = RIGHT – LEFT + 1 (mod N)

(c) With a queue, the array is full when

(i) FRONT = 1 and REAR = N or (ii) FRONT = REAR + 1

Similarly, with a deque, the array is full when

(i) LEFT = 1 and RIGHT = N or (ii) LEFT = RIGHT + 1

Each of these conditions implies NUMB = N.

6.24 Consider the following deque of characters where DEQUE is a circular array which allocated six memory cells:

LEFT = 2, RIGHT = 4 DEQUE: ___, A, C, D, ___, ___

Describe the deque while the following operations take place.

(a) F is added to the right of the deque.
(b) Two letters on the right are deleted.
(c) K, L and M are added to the left of the deque.
(d) One letter on the left is deleted.
(e) R is added to the left of the deque.
(f) S is added to the right of the deque.
(g) T is added to the right of the deque.

(a) F is added on the right, yielding

LEFT = 2, RIGHT = 5 DEQUE: ___, A, C, D, F, ___

Note that RIGHT is increased by 1.

(b) The two right letters, F and D, are deleted, yielding

LEFT = 2, RIGHT = 3 DEQUE: ___, A, C, ___, ___, ___

Note that RIGHT is decreased by 2.

(c) K, L and M are added on the left. Since K is placed in the first memory cell, L is placed in the last memory cell and M is placed in the next-to-last memory cell. This yields

LEFT = 5, RIGHT = 3 DEQUE: K, A, C, ___, M, L

Note that LEFT is decreased by 3 but the arithmetic is modulo 6:

LEFT = 2 – 3 = –1 = 5 (mod 6)

(d) The left letter, M, is deleted, leaving

LEFT = 6, RIGHT = 3 DEQUE: K, A, C, ___, ___, L

Note that LEFT is increased by 1.

(e) R is added on the left, yielding

LEFT = 5, RIGHT = 3 DEQUE: K, A, C, —, R, L

Note that LEFT is decreased by 1.

(f) S is added on the right; yielding

LEFT = 5, RIGHT = 4 DEQUE: K, A, C, S, R, L

(g) Since LEFT = RIGHT + 1, the array is full, and hence T cannot be added to the deque. That is, overflow has occurred.

6.16 Use Definition 6.3 (of the Ackermann function) to find $A(2, 2)$.

6.17 Let M and N be integers and suppose F(M, N) is recursively defined by

$$F(M, N) = \begin{cases} 1 & \text{if } M = 0 \text{ or } M \geq N \geq 1 \\ F(M-1, N) + F(M-1, N-1) & \text{otherwise} \end{cases}$$

(a) Find (4, 2), F(1, 5) and F(2, 4). (b) When is F(M, N) undefined?

6.18 Let A be an integer array with N elements. Suppose X is an integer function def

$$X(K) = X(A, N, K) = \begin{cases} 0 & \text{if } K = 0 \\ X(K-1) + A(K) & \text{if } 0 < K \leq N \\ X(K-1) & \text{if } K > N \end{cases}$$

Find X(5) for each of the following arrays:

(a) N = 8, A: 3, 7, −2, 5, 6, −4, 2, 7 (b) N = 3, A: 2, 7, −4

What does this function do?

6.19 Show that the recursive solution to the Towers of Hanoi problem in Sec. 6.8 re $= 2^n - 1$ moves for n disks. Show that no other solution uses fewer than $f(n)$ mo

6.20 Suppose S is a string with N characters. Let SUB(S, J, L) denote the substring of S in the position J and having length L. Let A//B denote the concatenation of string Suppose REV(S, N) is recursively defined by

$$REV(S, N) = \begin{cases} S & \text{if } N = 1 \\ SUB(S, N, 1)//REV(SUB(S, 1, N-1) & \text{otherwise} \end{cases}$$

(a) Find REV(S, N) when (i) N = 3, S = *abc* and (ii) N = 5, S = *ababc*. (b) Wha function do?

Queues; Deques

6.21 Consider the following queue where QUEUE is allocated 6 memory cells:

FRONT = 2, REAR = 5 QUEUE: _____, London, Berlin, Rome, Paris, —

Describe the queue, including FRONT and REAR, as the following operations (a) Athens is added, (b) two cities are deleted, (c) Madrid is added, (d) Mosco (e) three cities are deleted and (f) Oslo is added.

6.22 Consider the following deque where DEQUE is allocated 6 memory cells:

LEFT = 2, RIGHT = 5 DEQUE: _____, London, Berlin, Rome, Paris, —

Describe the deque, including LEFT and RIGHT, as the following operations tak

(a) Athens is added on the left.

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.

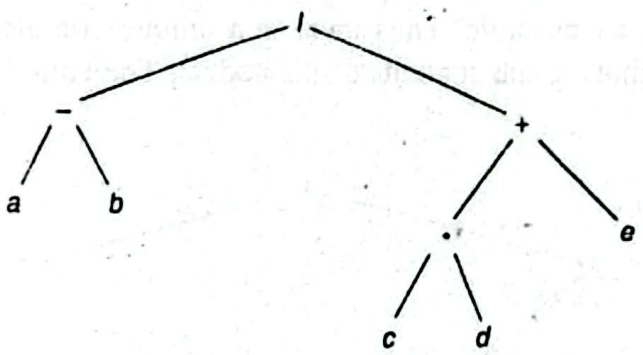


Fig. 7.3 $E = (a - b)/((c * d) + e)$

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

Terminology

Terminology describing family relationships is frequently used to describe relationships between 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, a 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, 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.

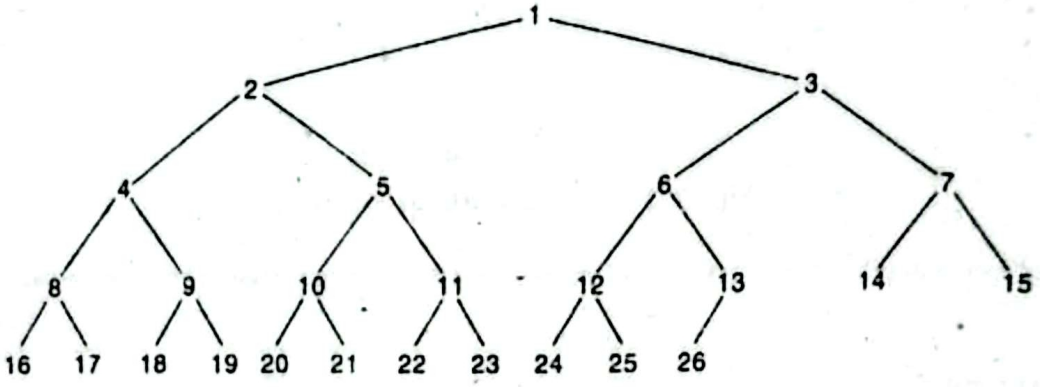


Fig. 7.4 *Complete Tree T_{26}*

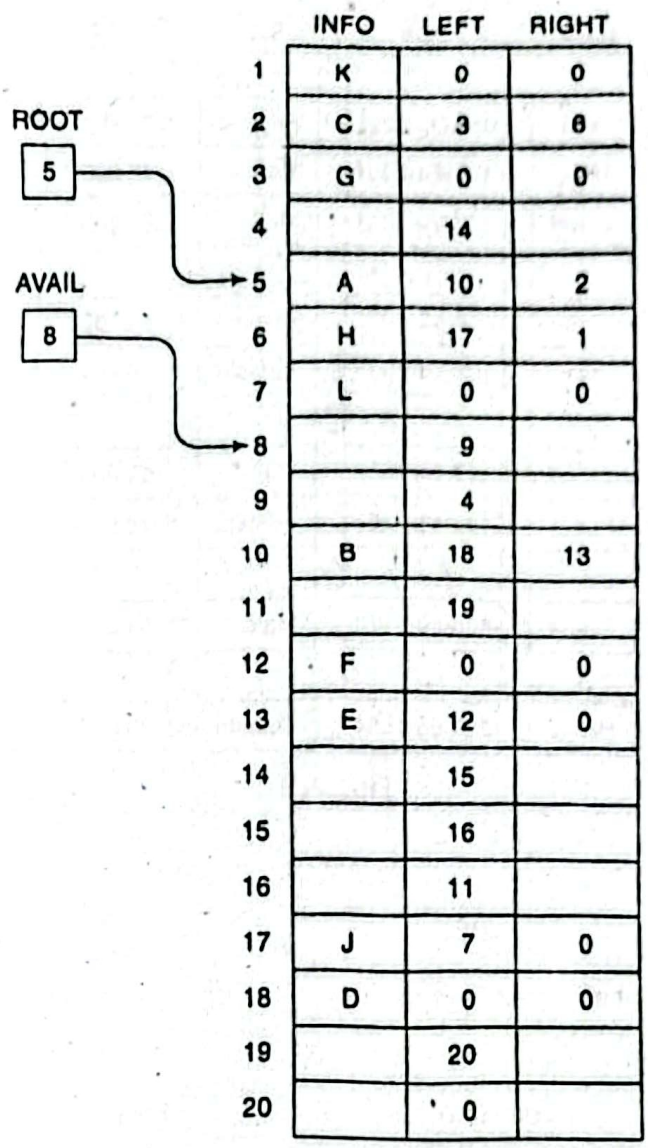
The nodes of the complete binary tree T_{26} in Fig. 7.4 have been purposely labeled by the integers 1, 2, ..., 26, from left to right, generation by generation. With this labeling, one can easily determine the children and parent of any node K in any complete tree T_n. Specifically, the left and right 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 relative y 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.



	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

Suppose we want to draw the tree diagram which corresponds to the binary tree in Fig. 7.8. For national convenience, we label the nodes in the tree diagram only by the key values NAME. We nstruct the tree as follows:

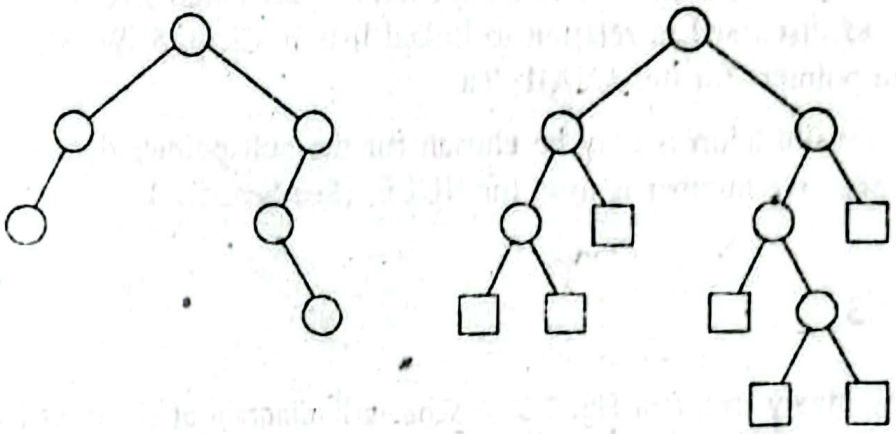
(a) The value ROOT = 14 indicates that Harris is the root of the tree.
(b) LEFT[14] = 9 indicates that Cohen is the left child of Harris, and RIGHT[14] = 7 indicates that Lewis is the right child of Harris.

peating Step (b) for each new node in the diagram, we obtain Fig. 7.9.

quential Representation of Binary Trees

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

term "exte... comes from the following operation. Consider any binary tree as the tree in Fig. 7.5(a). Then T may be "converted" into a 2-tree by replacing each empty by a new node, as pictured in Fig. 7.5(b). Observe that the new tree is, indeed, a 2-tree. thermore, the nodes in the original tree T are now the internal nodes in the extended tree, and 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 hich uses only binary operations. As illustrated in Fig. 7.3, the variables in E will appear as the ternal nodes, and the operations in E will appear as internal nodes.

3 REPRESENTING BINARY TREES IN MEMORY

T be a binary tree. This section discusses two ways of representing T in memory. The first and ual way is called the link representation of T and is analogous to the way linked lists are presented 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 the root R of T and, given any node N of T, one should have direct access to the children of N.

inked Representation of Binary Trees

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

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

Remark 1: Most of our examples will show a single item of information at each node N of a b tree T. In actual practice, an entire record may be stored at the node N. In other words, INFO 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 impl assume that the empty locations in the arrays INFO, LEFT and RIGHT form a linked lis pointer AVAIL, as discussed in relation to linked lists in Chap. 5. We will usually let the array contain the pointers for the AVAIL list.

Remark 3: Any invalid address may be chosen for the null pointer denoted by NULL. In 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.

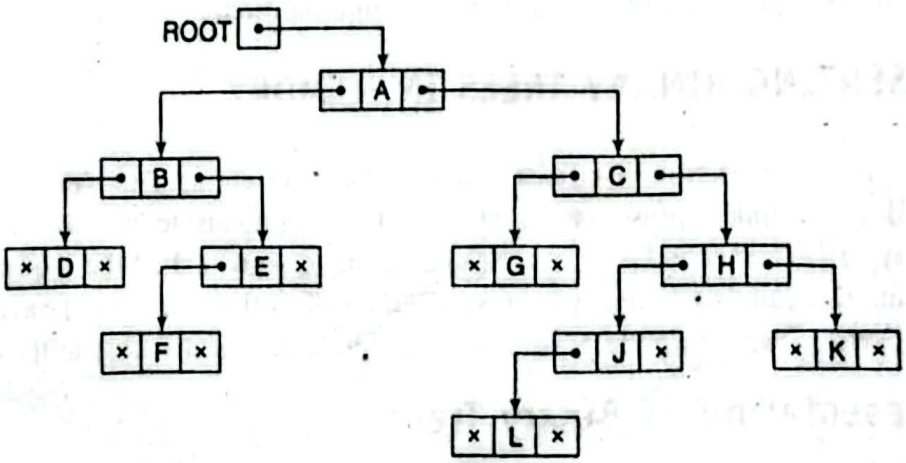


Fig. 7.6

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 7.1 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.

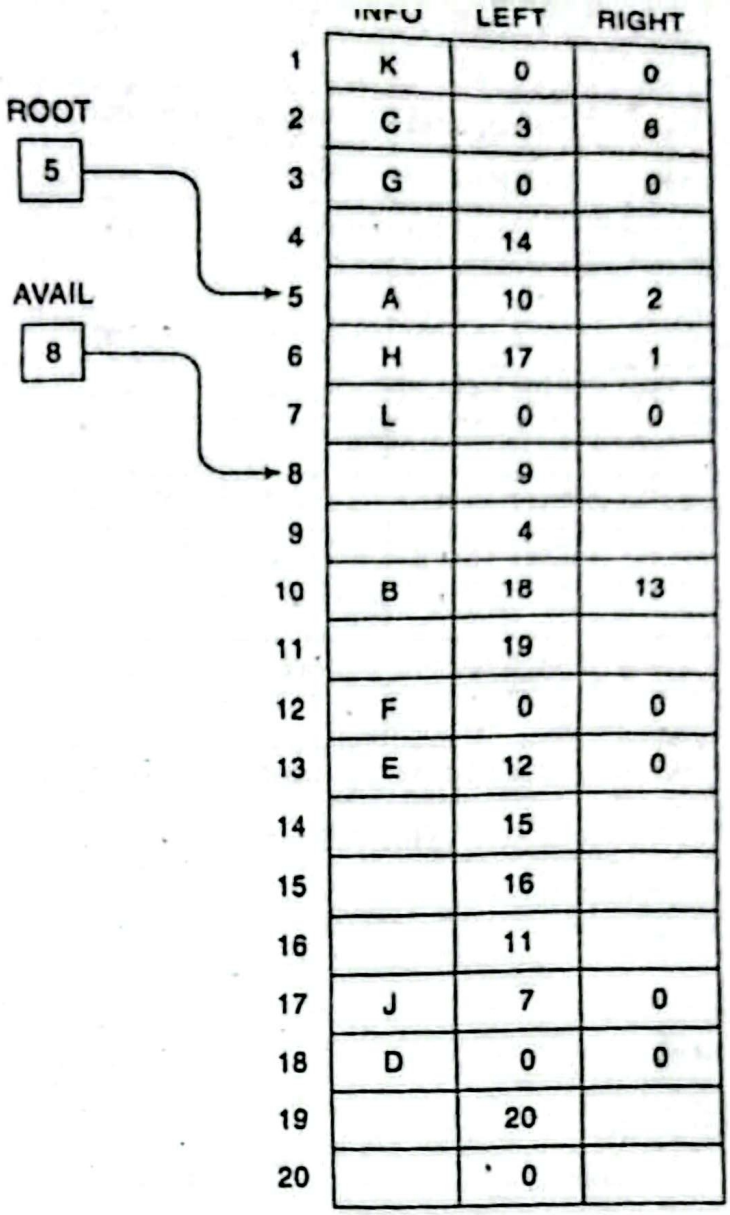


Fig. 7.7

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:

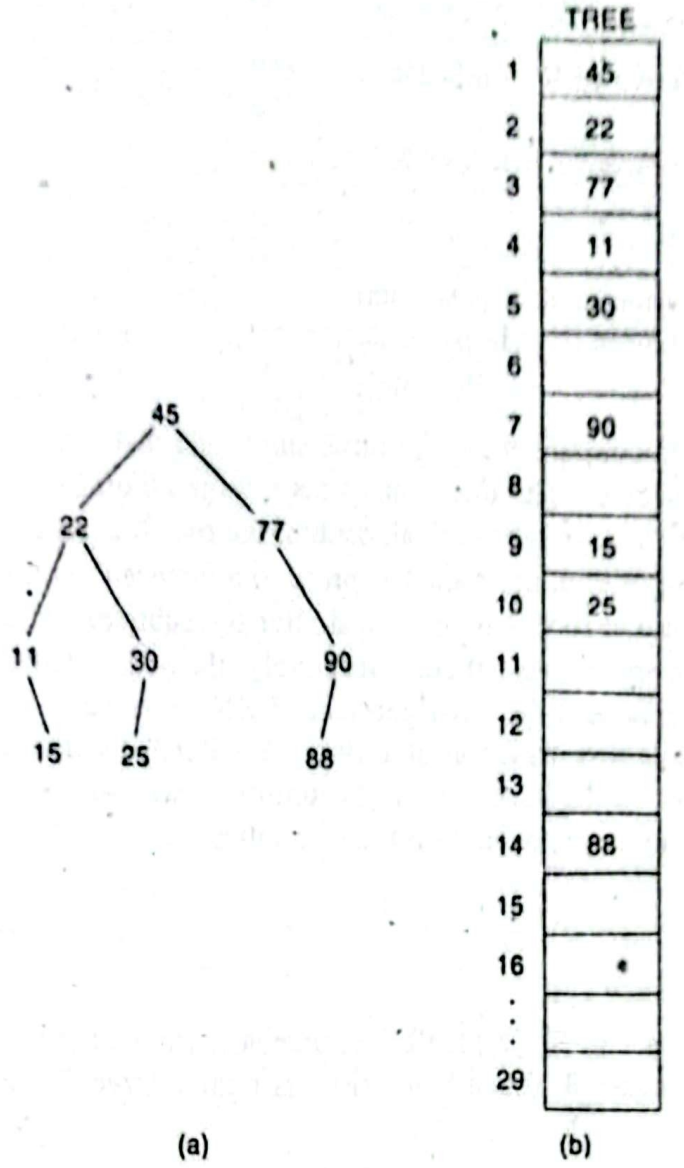


Fig. 7.10

a tree with depth d will require an array with approximately 2^{d+1} elements. Accordingly, this equential representation is usually inefficient unless, as stated above, the binary tree T is complete r nearly complete. For example, the tree T in Fig. 7.1 has 11 nodes and depth 5, which means it ould 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.

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) * + a - b c / - d e - + f g h
(Postorder) a b c - + d e - f g + h - / *

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.

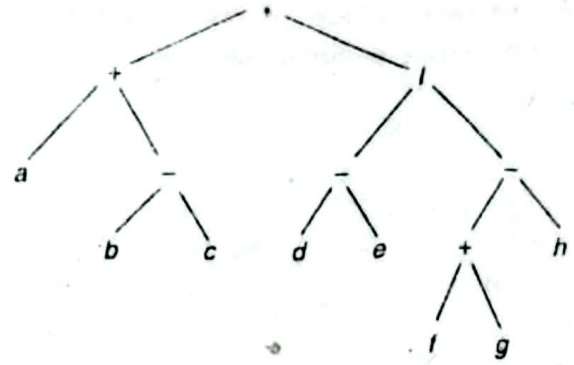


Fig. 7.13

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.

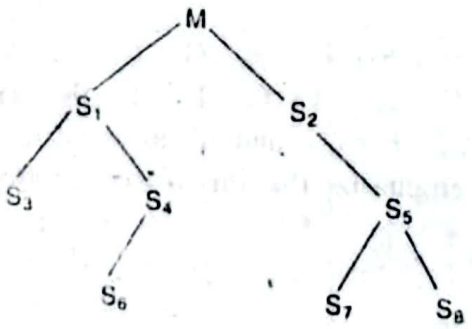


Fig. 7.14

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

TRAVERSAL ALGORITHMS USING STACKS

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

TREE(INFO, LEFT, RIGHT, ROOT)

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

rder Traversal

rorder traversal algorithm uses a variable PTR (pointer) which will contain the location of de N currently being scanned. This is pictured in Fig. 7.15, where L(N) denotes the left child de N and R(N) denotes the right child. The algorithm also uses an array STACK, which will he addresses of nodes for future processing.

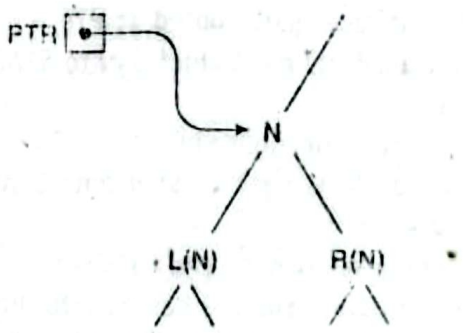


Fig. 7.15

rithm: 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 numbers 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.

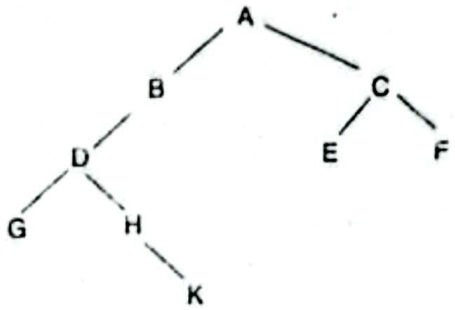


Fig. 7.16

1. Initially push NULL onto STACK:
   STACK: ∅.
   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: ∅, C.
   (ii) Process B. (There is no right child.)
   (iii) Process D and push its right child H onto STACK:
   STACK: ∅, 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: ∅, 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: ∅, 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: ∅, 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: 0.
   Since PTR ≠ NULL, return to Step (a) of the algorithm.
8. Proceed down the left most path rooted at PTR = C as follows:
   (vii) Process C and push its right child F onto STACK:
   STACK: 0, F.
   (viii) Process E. (There is no right child.)
9. [Backtracking.] Pop F from STACK, and set PTR := F. This leaves:
   STACK: 0.
   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.

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

...formal presentation of our preorder traversal algorithm follows:

...rithm 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 := [address] Then repeat the following steps until NULL is popped from STACK.

(a) Proceed down the left-most path rooted at PTR, pushing each node 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.

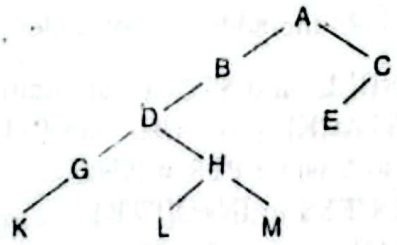


Fig. 7.17

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: 0, 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: 0, 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: 0, 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: 0.
   (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: 0, 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 1, 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.

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].

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

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

dicate an empty tree.

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

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

dicate an empty subtree.

reads; Inorder Threading

sider again the linked representation of a binary tree T. Approximately half of the entries in the nter fields LEFT and RIGHT will contain null elements. This space may be more efficiently ed by replacing the null entries by some other type of information. Specifically, we will replace tain null entries by special pointers which point to nodes higher in the tree. These special nters 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 reads in a diagram of a threaded tree are usually indicated by dotted lines. In computer memory, extra 1-bit TAG field may be used to distinguish threads from ordinary pointers, or, alternaely, threads may be denoted by negative integers when ordinary pointers are denoted by positive tegers.

There are many ways to thread a binary tree T, but each threading will correspond to a particur traversal of T. Also, one may choose a one-way threading or a two-way threading. Unless herwise stated, our threading will correspond to the inorder traversal of T. Accordingly, in the ne-way threading of T, a thread will appear in the right field of a node and will point to the next ode in the inorder traversal of T; and in the two-way threading of T, a thread will also appear in e LEFT field of a node and will point to the preceding node in the inorder traversal of T. rthermore, the left pointer of the first node and the right pointer of the last node (in the inorder aversal of T) will contain the null value when T does not have a header node, but will point to the eader 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 aversal of T. (See Solved Problem 7.13.) On the other hand, there is no threading of T which rresponds to the postorder traversal of T.

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.

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, here we may have to save a node N in two different situations. We distinguish between cases by pushing either N or its negative, −N, onto STACK. (In actual practice, the location pushed onto STACK, so −N has the obvious meaning.) Again, a variable PTR (pointer) 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 := [illegible]. 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 path, push N onto STACK and, if N has a right child R(N), push −R onto STACK.
(b) [Backtracking.] Pop and process positive nodes on STACK. If N 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 go 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: 0.
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: 0, 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: 0, A, −C, B, D.
   Now PTR = −H. Reset PTR = H and return to Step (a).

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 certain operations on T are easier to implement by using the modifications. Some of these tions, 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. Some an extra, special node, called a *header node*, is added to the beginning of T. When this extra is used, the tree pointer variable, which we will call HEAD (instead of ROOT), will point header node, and the left pointer of the header node will point to the root of T. Figure 7.18 a schematic picture of the binary tree in Fig. 7.1 that uses a linked representation with a node. (Compare with Fig. 7.6.)

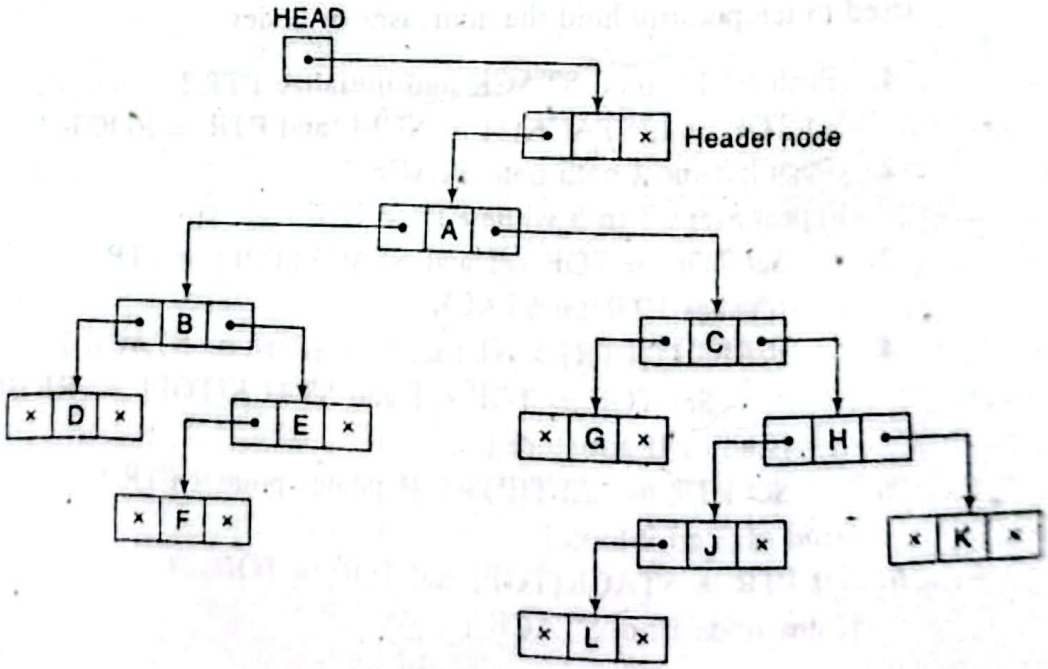
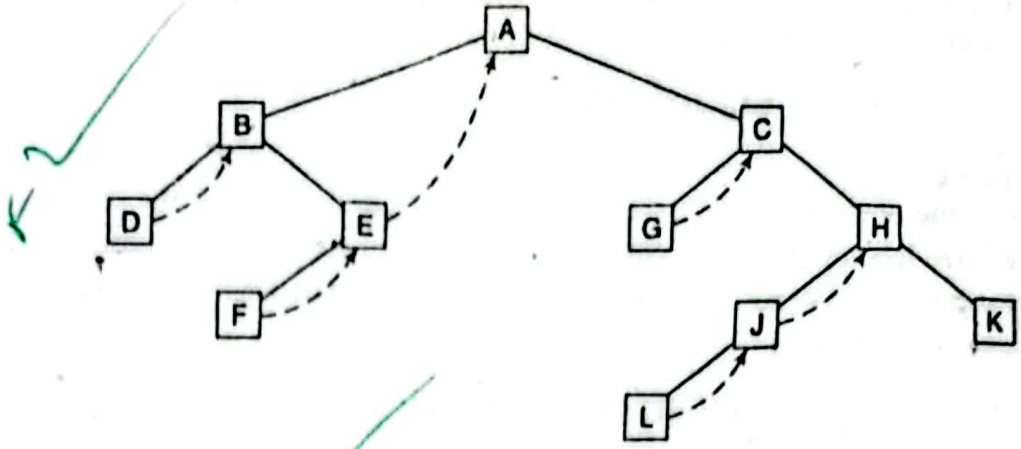
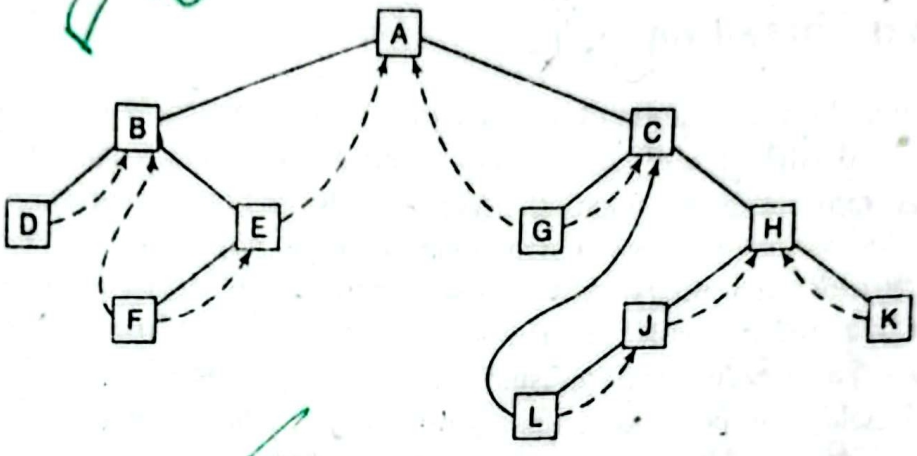


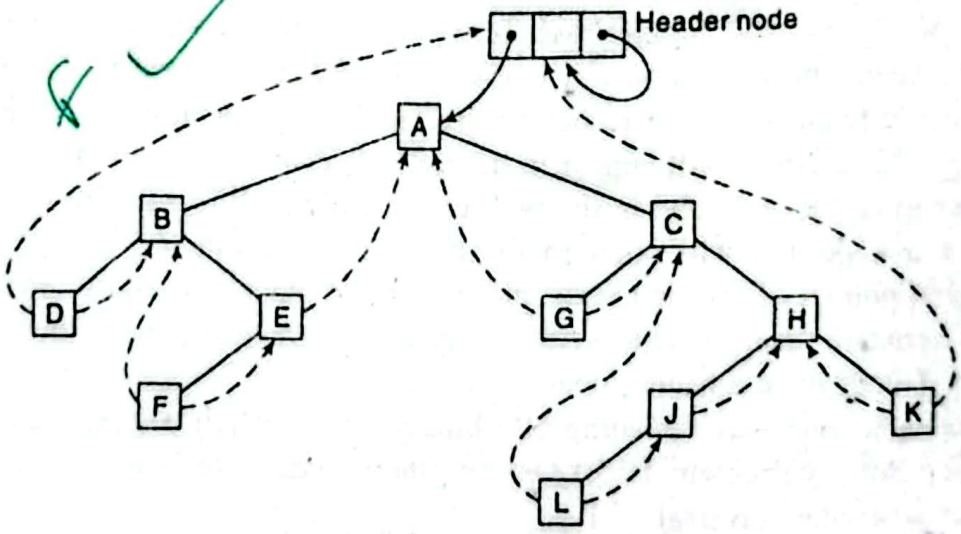
Fig. 7.18



(a) One-way inorder threading



(b) Two-way inorder threading



(c) Two-way threading with header node

Fig. 7.19

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

7.7 BINARY SEARCH TREES

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

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

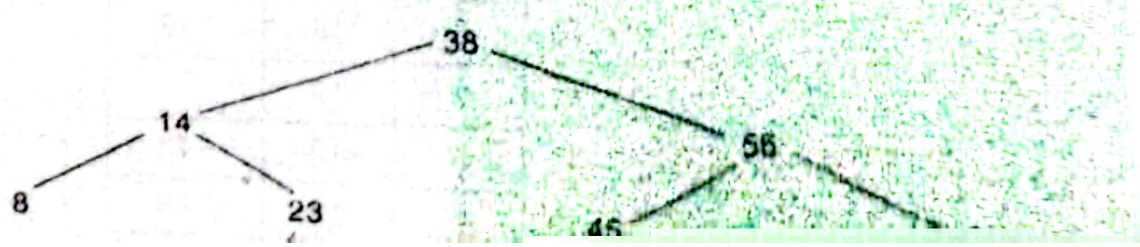
(b) *Linked list.* Here one can easily insert and delete elements, but it is expensive to and find an element, since one must use a linear search with running time $f(n)$

Although each node in a binary search tree may contain an entire record of data, the d 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 t* node N of T has the following property: *The value at* N *is greater than every value subtree* of N *and is less than every value in the right subtree* of N. (It is not difficult to property guarantees that the inorder traversal of T will yield a sorted listing of the elem

Example 7.13

(a) Consider the binary tree T in Fig. 7.21. T is a binary search tree; that is, ever 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 woul 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.



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.

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.

(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:

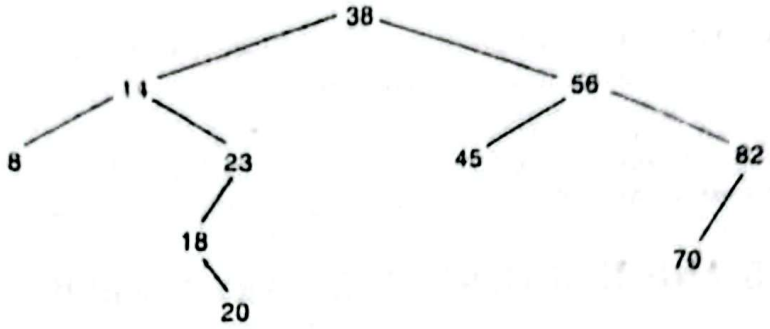


Fig. 7.22 *ITEM = 20 Inserted*

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.

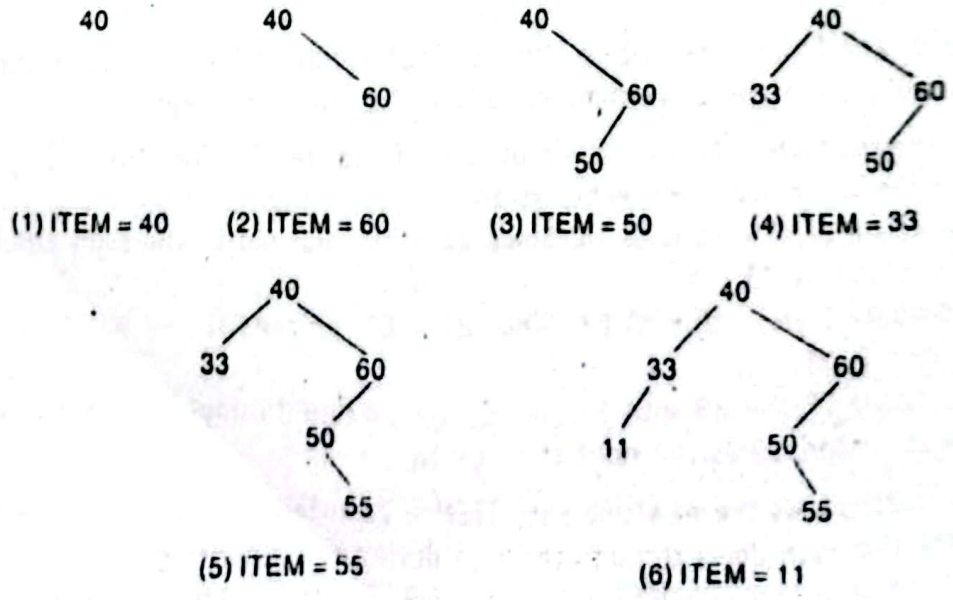


Fig. 7.23

by Algorithm B is approximately $n \log_2 n$, that is, $f(n) = O(n \log_2 n)$. For example, for ... Algorithm B will require approximately 10 000 comparisons rather than the 500 000 co... with Algorithm A. (We note that, for the worst case, the number of comparisons for Al... 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

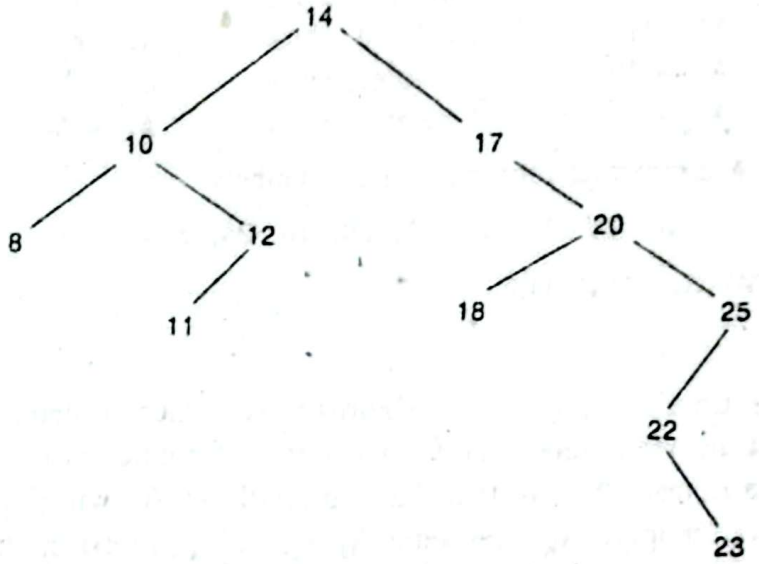


Fig. 7.24

On the other hand, Algorithm A requires

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

comparisons.

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 ... 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 ... contains ITEM and also the location of the parent node P(N). The way N is deleted from the ... 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 ... parent node P(N) by the null pointer.

Example 7.30

Deletion of keys 95, 226, 221 and 70 on a given B-tree of order 5 is shown in Fig. 7.55. The deletion of key 95 is simple and straight since the leaf node has more than the minimum number of elements. To delete 226, the internal node has only the minimum number of elements and hence borrows the immediate successor viz., 300 from the leaf node which has more than the minimum number of elements. Deletion of 221 calls for the hauling of key 440 to the parent node and pulling down of 300 to take the place of the deleted entry in the leaf. Lastly the deletion of 70 is a little more involved in process. Since none of the adjacent leaf nodes can afford lending a key, two of the leaf nodes are combined with the intervening element from the parent to form a new leaf node, viz., [32, 44, 65, 81] leaving 86 alone in the parent node. This is not possible since the parent node is now running low on its minimum number of elements. Hence we once again proceed to combine the adjacent sibling nodes of the specified parent node with a median element of the parent which is the root. This yields the node [86, 110, 120, 440] which is the new root. Observe the reduction in height of the B-tree.

Example 7.31

On the B-tree of order 3 shown in Fig. 7.56, perform the following operations in the order of their appearance:

Insert 75, 57 Delete 35, 65

The B-tree after the operations is shown in Fig. 7.57

17 HEAP; HEAPSORT

is section discusses another tree structure, called a *heap*. The heap is used in an elegant sorting orithm called *heapsort*. Although sorting will be treated mainly in Chapter 9, we give the apsort algorithm here and compare its complexity with that of the bubble sort and quicksort orithms, 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 is maintained in memory by a linear array TREE using the sequential representation of H, not a ked representation.) Then H is called a *heap*, or a *maxheap*, if each node N of H has the llowing property: *The value at* N *is greater than or equal to the value at each of the children of* Accordingly, the value at N is greater than or equal to the value at any of the descendants of N. *minheap* is defined analogously: The value at N is less than or equal to the value at any of the ildren of N.)

B-tree of order 3.

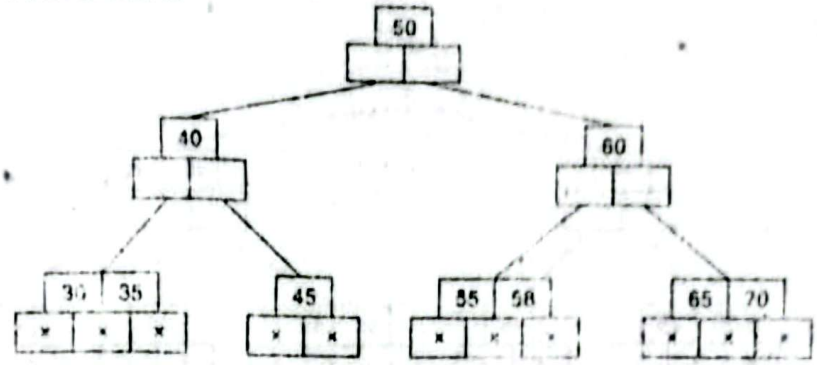


Fig. 7.56

Example 7.32

Consider the complete tree H in Fig. 7.58(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 7.58(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.

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 7.33

Consider the heap H in Fig. 7.58. 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. 7.59(a). We now find the appropriate place of 70 in the heap as follows:

mal statement of our insertion procedure follows:

dure 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[I] := ITEM.
8. Return.

rve that ITEM is not assigned to an element of the array TREE until the appropriate place for M is found. Step 7 takes care of the special case that ITEM rises to the root TREE[1].
ppose an array A with N elements is given. By repeatedly applying Procedure 7.9 to A, that y executing

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

I = 1, 2, ..., N – 1, we can build a heap H out of the array A.

leting the Root of a Heap

pose H is a heap with N elements, and suppose we want to delete the root R of H. This is mplished 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.

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

(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 7.59(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 and that nothing else has been disturbed. This is easy to see, and we leave this verification to the reader.

Example 7.34

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 7.60(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.

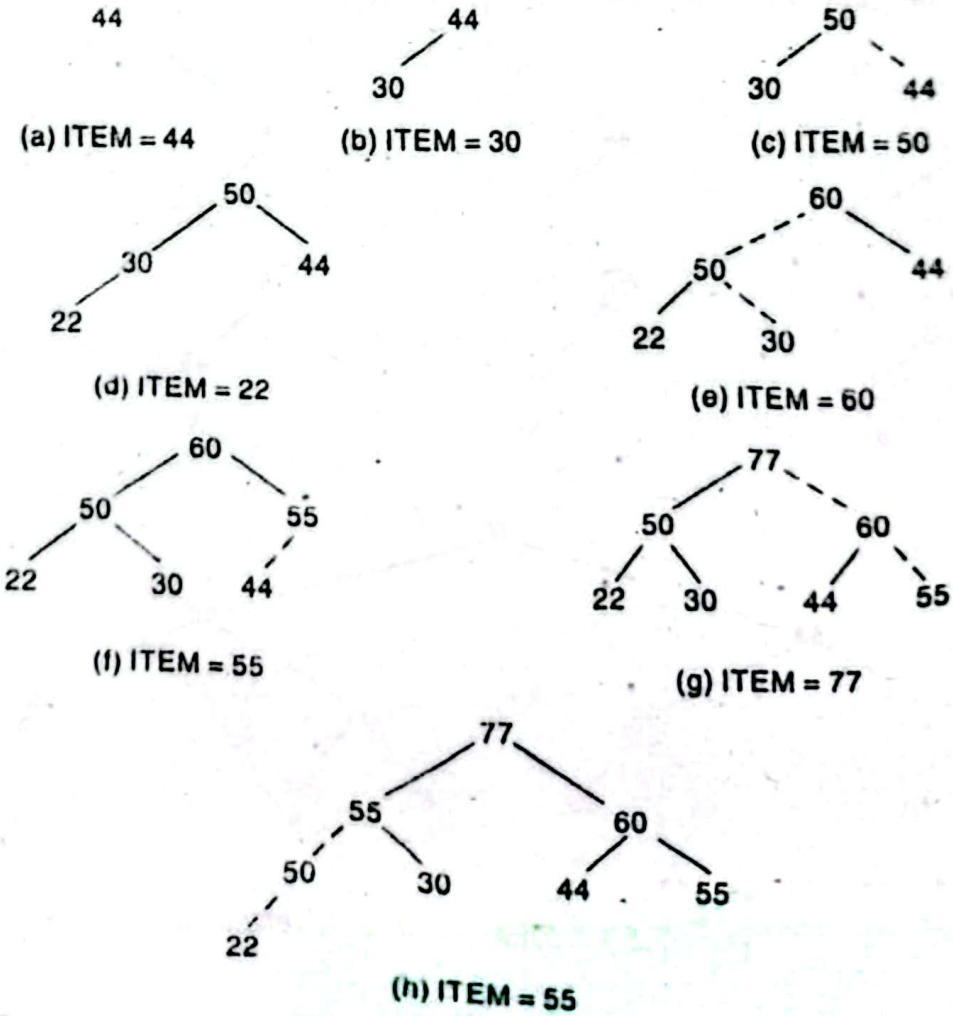
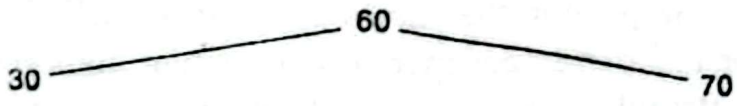


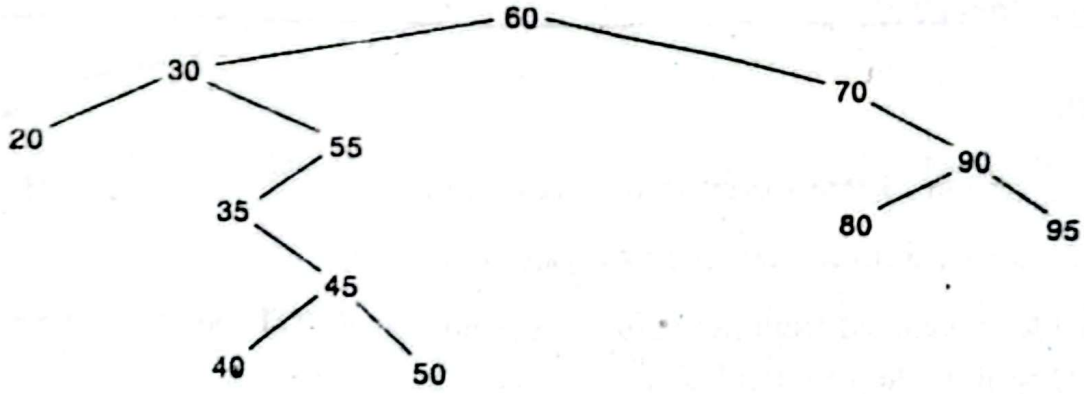
Fig. 7.60 Build H

5	2	30	1	13
AVAIL	3	40	0	0
9	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. 7.72



(a)



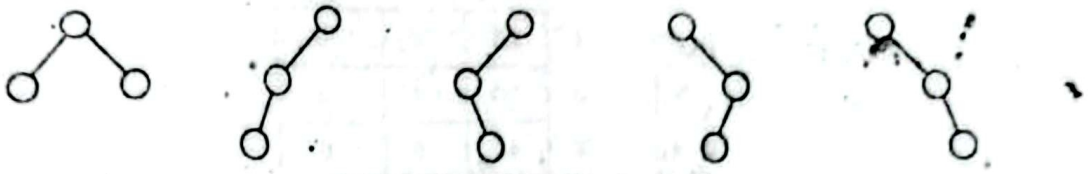
(b)

Fig. 7.73

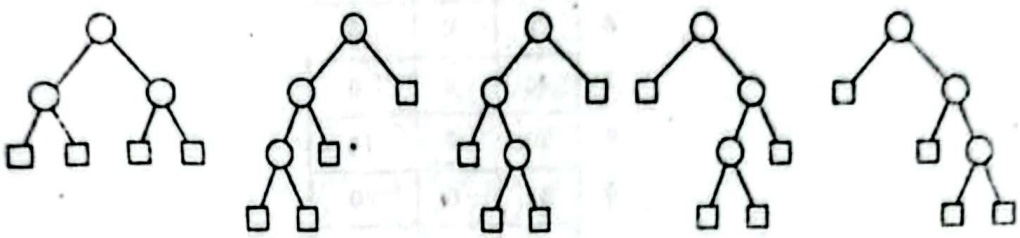
7.2 A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the following sequences of nodes:

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

Draw the tree T.



(a) Binary trees with 3 nodes



(b) Extended binary trees with 4 external nodes

Fig. 7.76

Search Trees; Heaps

Consider the binary search tree T in Fig. 7.73(b), which is stored in memory as in Fig. 7.72. Suppose ITEM = 33 is added to the tree T. (a) Find the new tree T. (b) Which changes occur in Fig. 7.72?

(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. 7.77. The shaded edges indicate the path down through the tree during the insertion algorithm.

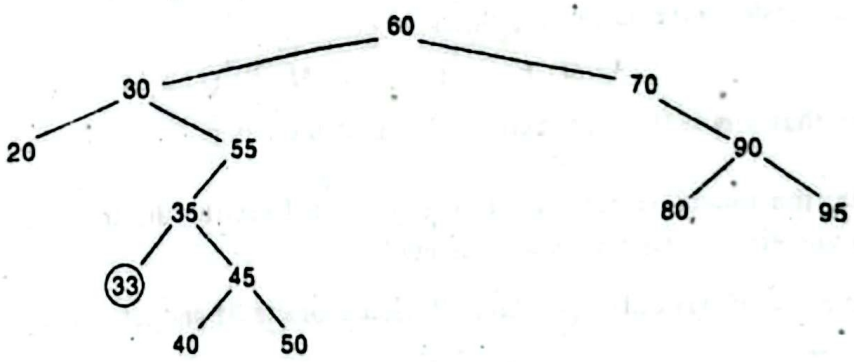


Fig. 7.77

(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 7.78 shows the updated tree T in memory. The shading indicates the changes from the original picture.

ROOT 5

AVAIL 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. 7.78

7.8 Suppose the following list of letters is inserted in order into an empty binary se

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

(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. 7.79.

(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.

7.9 Consider the binary search tree T in Fig. 7.79. Describe the tree after (a) the

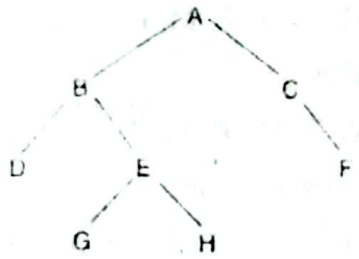
SUPPLEMENTARY PROBLEMS

Binary Trees

7.1 Consider the tree T in Fig. 7.95(a).

(a) Fill in the values for ROOT, LEFT and RIGHT in Fig. 7.95(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 descendents of node B.



(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. 7.95

7.2 List the nodes of the tree T in Fig. 7.95(a) in (a) preorder, (b) inorder and (c) postorder.

7.3 Draw the diagram of the tree T in Fig. 7.96.

7.4 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.

7.5 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 data 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.

Chapter Eight

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 $e = [u, v]$

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:

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

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.

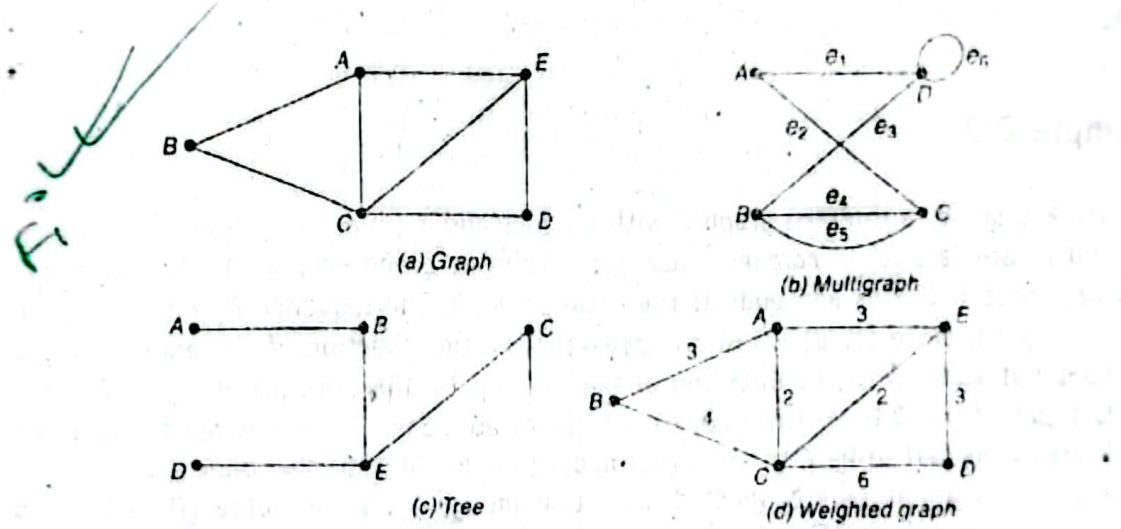


Fig. 8.1

(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

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 Chapter 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.

TRAVERSING A GRAPH

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

ring the execution of our algorithms, each node N of G will be in one of three states, called us 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.

ow discuss the two searches separately.

dth-First Search

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

orithm 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.

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

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.

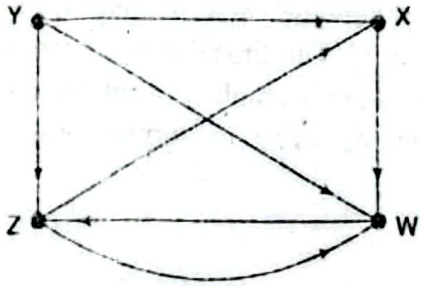


Fig. 8.3

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 Miscellaneous Problem 8.3 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:

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

ngly, in particular, there is a path of length 2 from v_4 to v_1, there are two paths of length 3 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

)

ose we now define the matrix B_r as follows:

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

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

Matrix

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

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

e 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 ust 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 ngth $m - 1$ or less, or such a cycle must have length m or less. This means that there is a ij entry in the matrix B_m, defined at the end of the preceding subsection. Accordingly, we e following relationship between the path matrix P and the adjacency matrix A.

sition 8.3

e the adjacency matrix and let $P = (p_{ij})$ be the path matrix of a digraph G. Then $P_{ij} = 1$ if y if there is a nonzero number in the ij entry of the matrix

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

sider the graph G with $m = 4$ nodes in Fig. 8.3. Adding the matrices A, A^2, A^3 and A^4, we the following matrix B_4, and, replacing the nonzero entries in B_4 by 1, we obtain the path P of the graph G:

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

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

ll that a directed graph G is said to be *strongly connected* if, for any pair of nodes u and v here are both a path from u to v and a path from v to u. Accordingly, G is strongly connected only if the path matrix P of G has no zero entries. Thus the graph G in Fig. 8.3 is not y connected.

(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 : Ø, 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 : Ø, 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 : Ø, 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 : Ø, 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

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

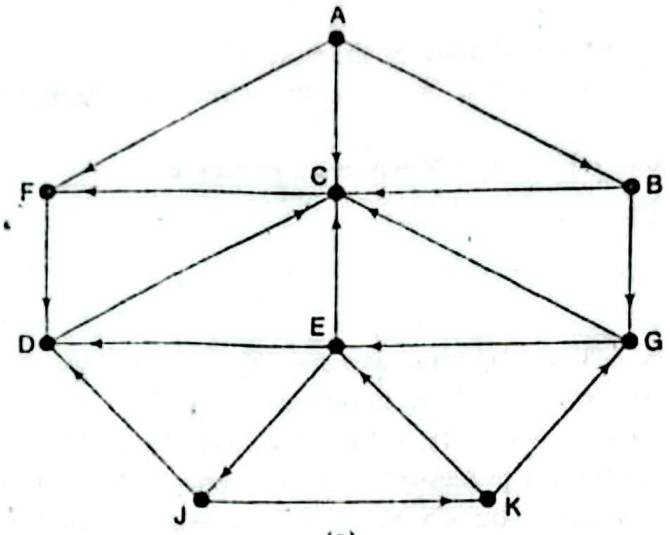
is the required path P.

epth-First Search

he 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.

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

The transitive *closure* of a graph G is defined to be the graph G' such that G' has 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 the complete graph.

Remark: The adjacency matrix A and the path matrix P of a graph G may be viewed as matrices (Boolean) matrices, where 0 represents "false" and 1 represents "true." Thus, the logical operations $\wedge$(AND) and $\vee$(OR) may be applied to the entries of A and P. The values of $\wedge$ and $\vee$ are given 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 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

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

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 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 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 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: