Chapter Seven

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-

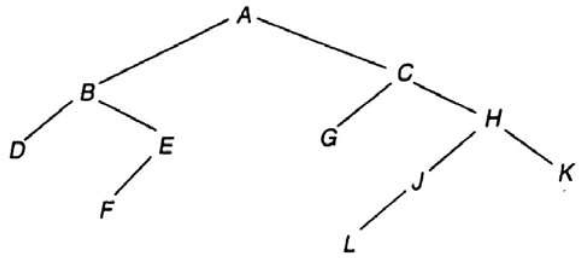


Fig. 7.1

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*.

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.

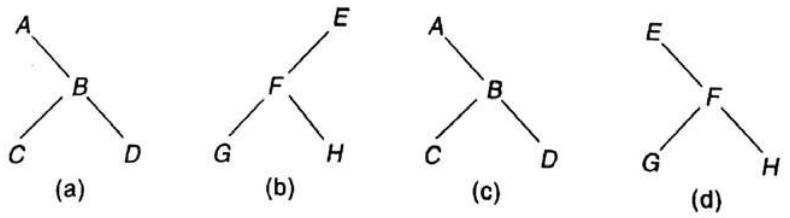


Fig. 7.2

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.

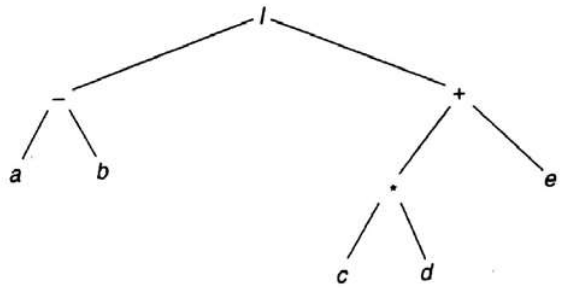


Fig. 7.3 $E = (a - b)/((c * d) + e)$

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 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.

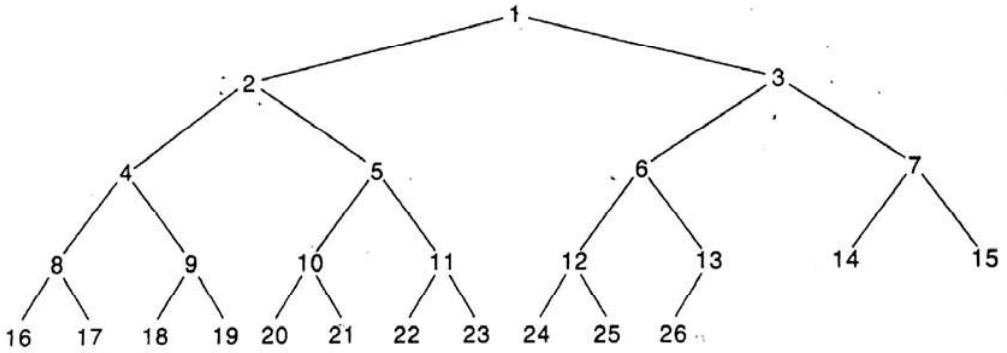


Fig. 7.4 *Complete Tree T_{26}*

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

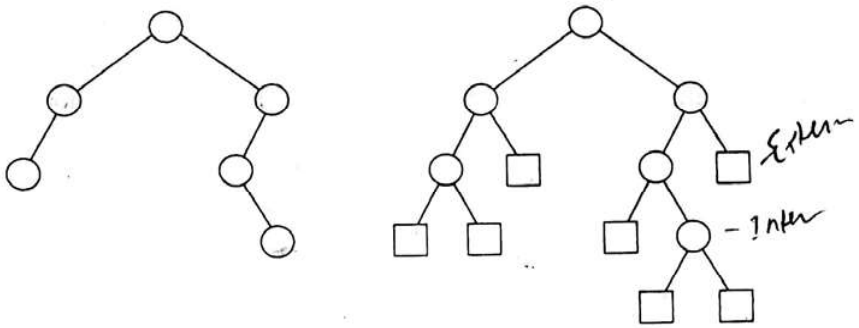
$$D_n = \lfloor \log_2 n + 1 \rfloor$$

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.

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.

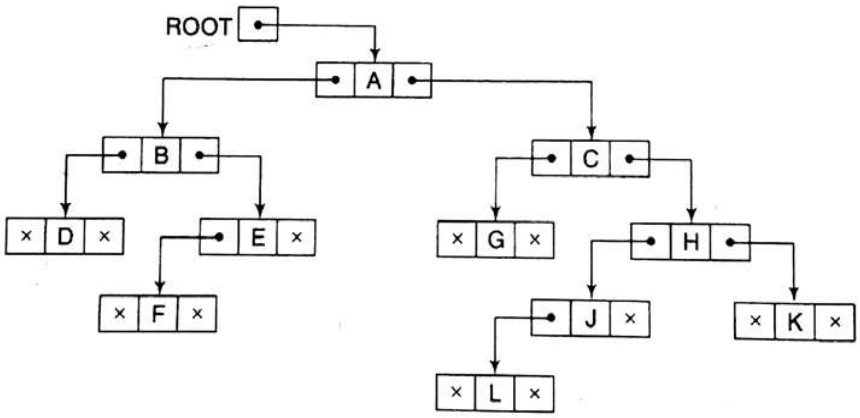


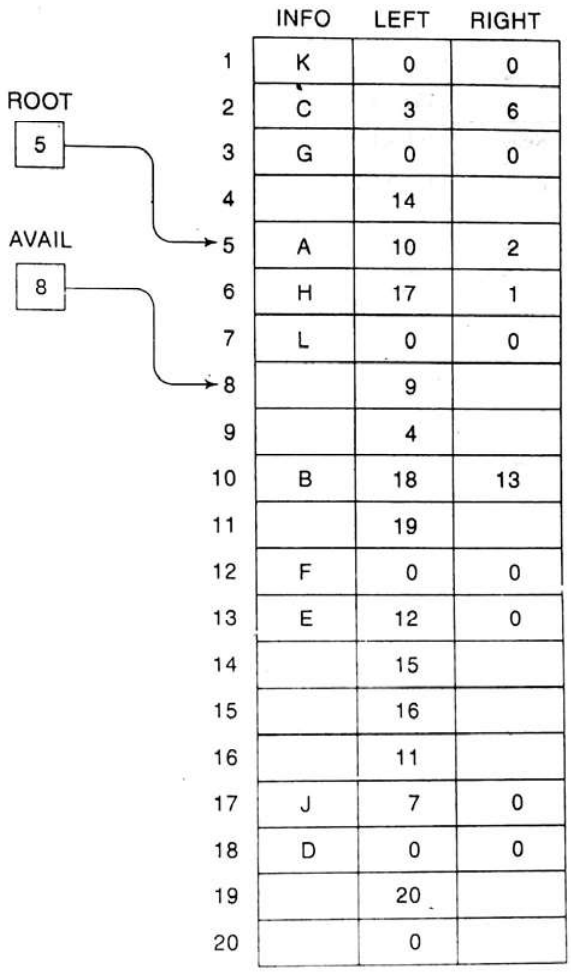
Fig. 7.6

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 7.8 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.



	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 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:

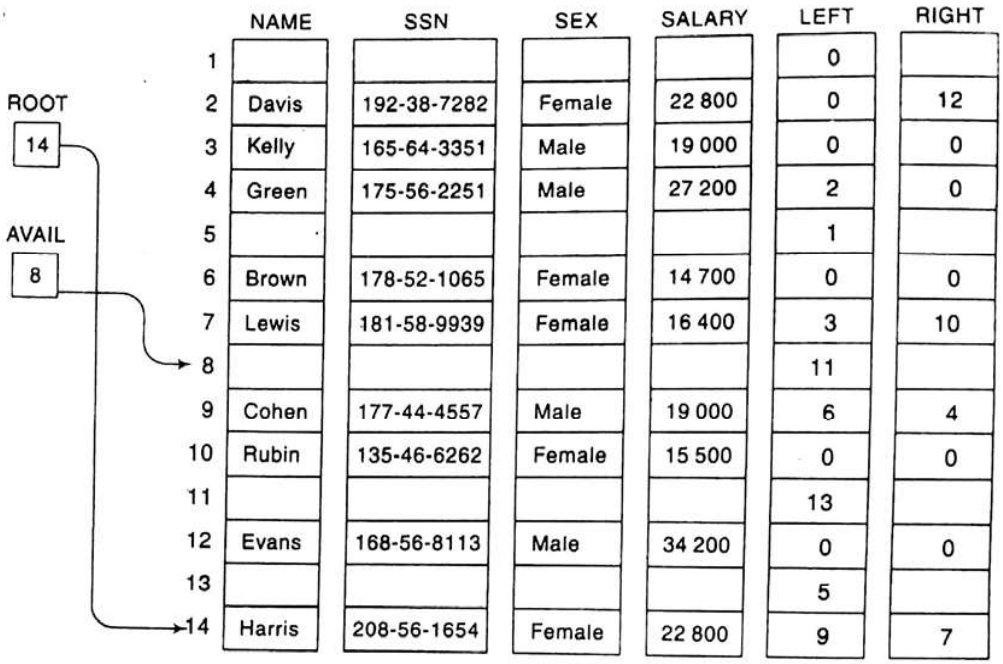


Fig. 7.8

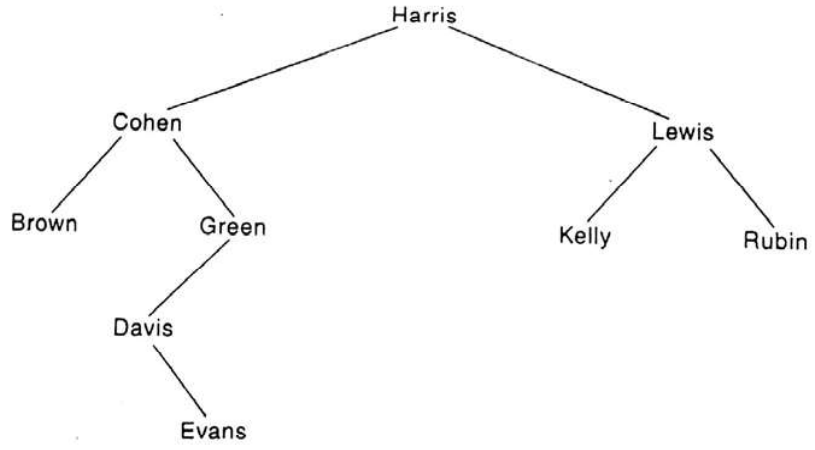


Fig. 7.9

(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

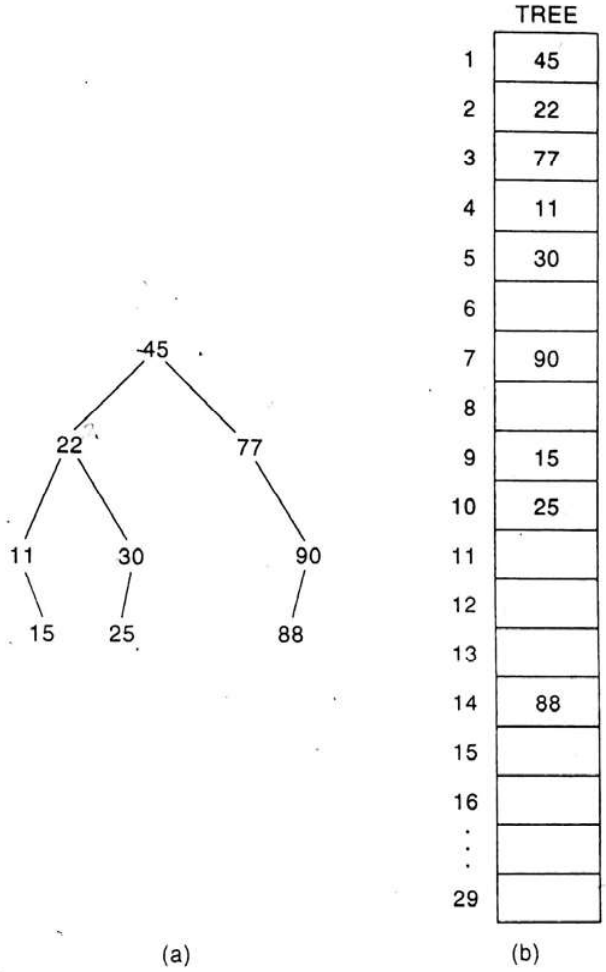


Fig. 7.10

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.

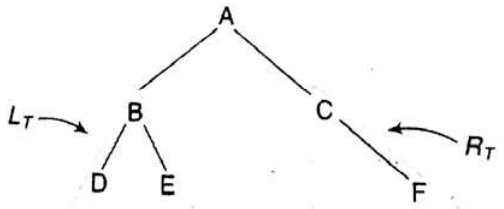


Fig. 7.11

(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.

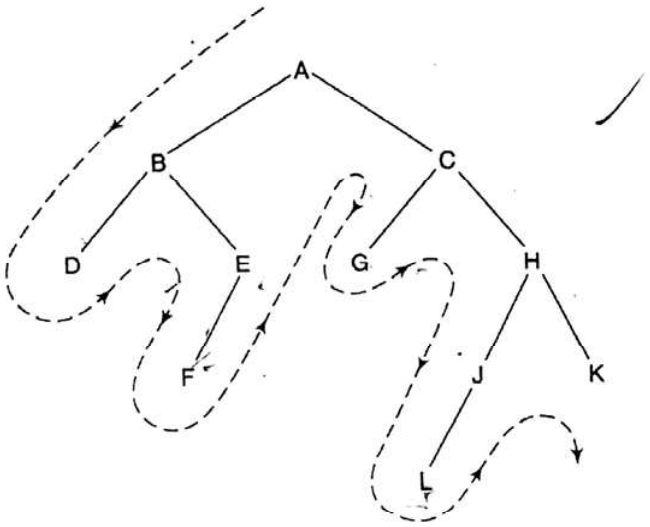


Fig. 7.12

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) * + 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.

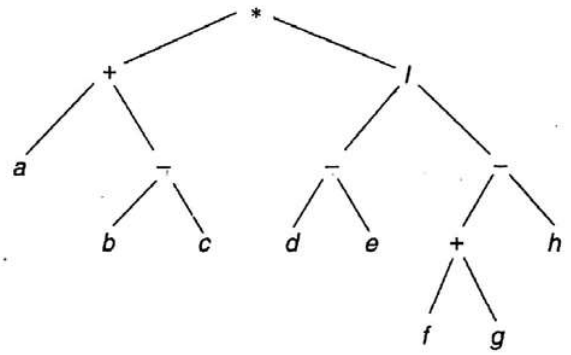


Fig. 7.13

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:

$$S_3,\quad S_6,\quad S_4,\quad S_1,\quad S_7,\quad S_8,\quad S_5,\quad S_2,\quad M$$

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 $S_1, S_2, \ldots, S_8$ come before the root M.

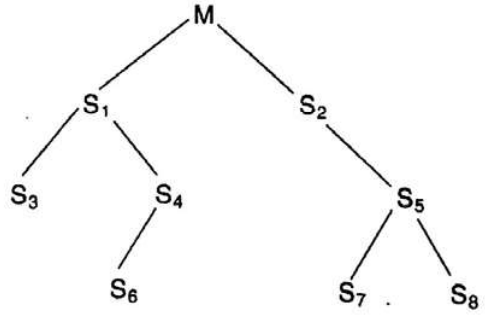


Fig. 7.14

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.

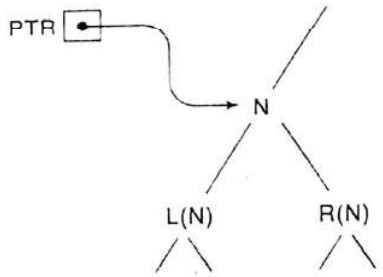


Fig. 7.15

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.

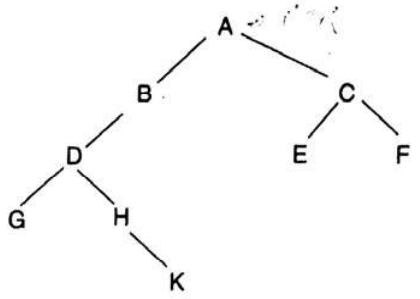


Fig. 7.16

1. Initially push NULL onto STACK:
   STACK: ∅.
   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: ∅.
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: ∅, F.
(viii) Process E. (There is no right child.)
9. [Backtracking.] Pop F from STACK, and set PTR := F. This leaves:
STACK: ∅.
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.

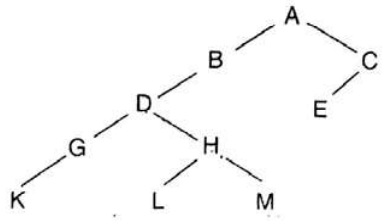


Fig. 7.17

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: Ø, 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: Ø, 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: Ø, 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: Ø.
   (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: Ø, 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.

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: ∅.
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: ∅, 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: ∅, 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.)

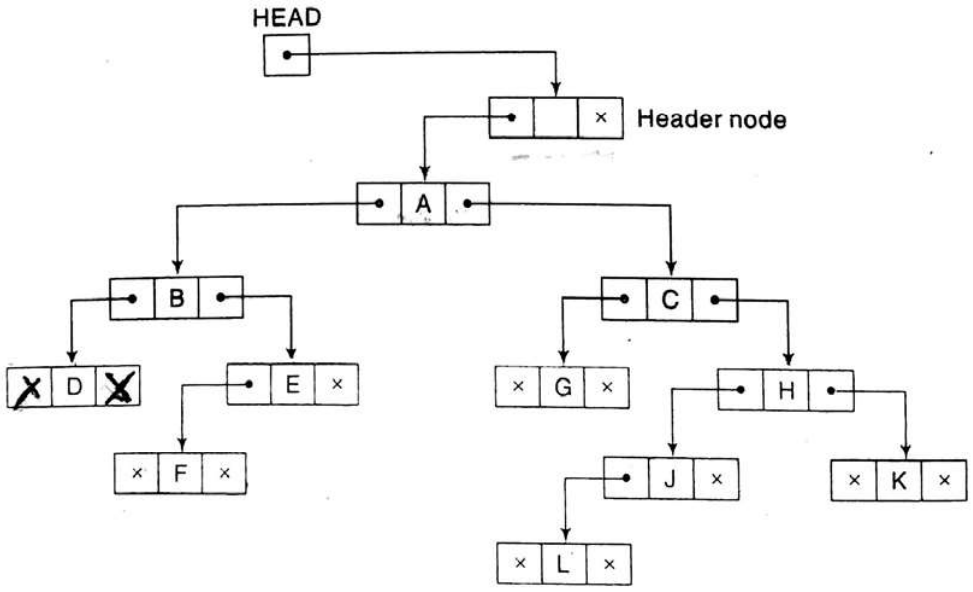


Fig. 7.18

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.

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

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

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.

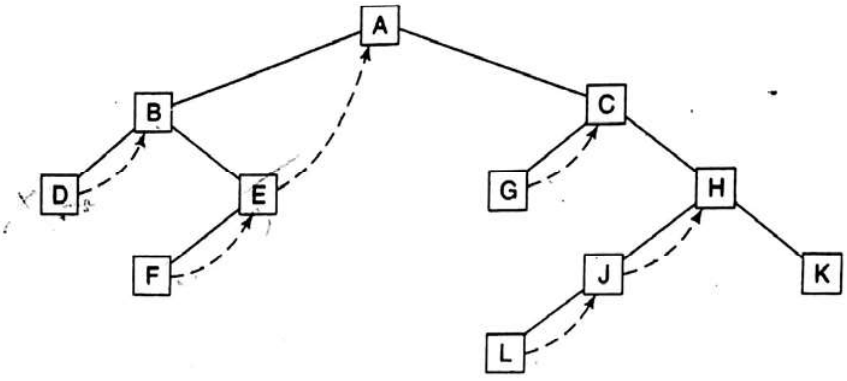
There are many ways to thread a binary tree T, but each threading will correspond to a particular traversal of T. Also, one may choose a one-way threading or a two-way threading. Unless otherwise stated, our threading will correspond to the inorder traversal of T. Accordingly, in the one-way threading 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 Solved Problem 7.13.) On the other hand, there is no threading of T which corresponds to the postorder traversal of T.

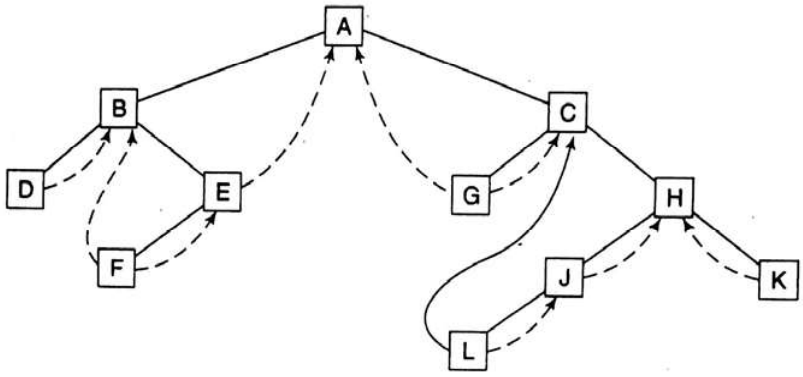
Example 7.12

Consider the binary tree T in Fig. 7.1.

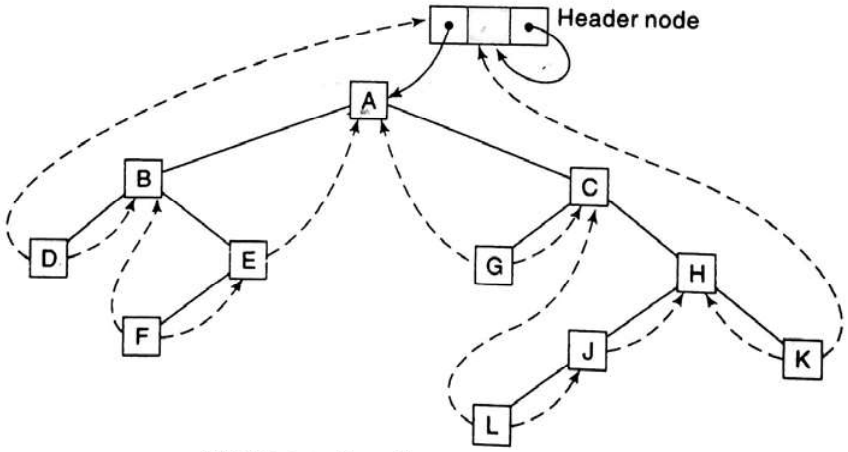
(a) The one-way inorder threading of T appears in Fig. 7.19(a). There is a thread from node E to node A, since A is accessed after E in the inorder traversal of T.



(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

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).

(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.

HEAD 20

AVAIL 8

	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

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.

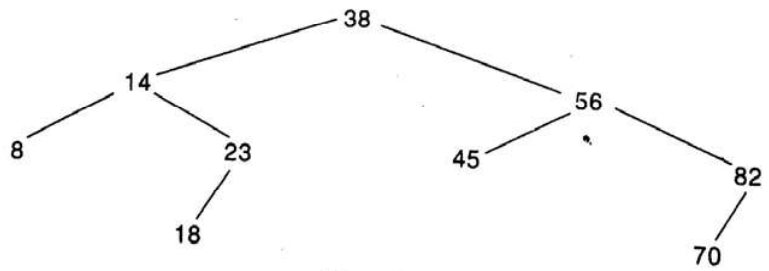


Fig. 7.21

(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.

(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:

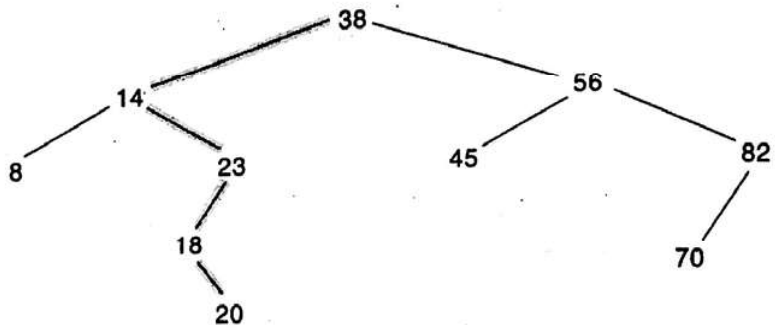


Fig. 7.22 *ITEM = 20 Inserted*

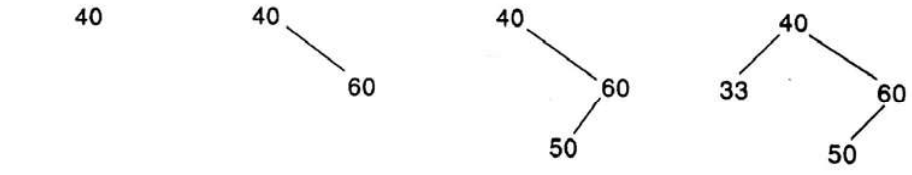
1. Compare ITEM = Davis with the root of the tree, Harris. Since Davis < Harris, proceed to the left child of Harris, which is Cohen.
2. Compare ITEM = Davis with Cohen. Since Davis > Cohen, proceed to the right child of Cohen, which is Green.
3. Compare ITEM = Davis with Green. Since Davis < Green,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

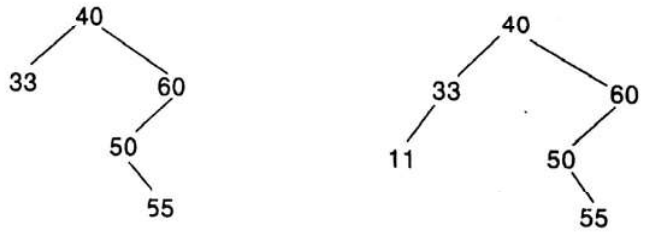
Suppose the following six numbers are inserted in order into an empty binary search tree:

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.



(1) ITEM = 40 (2) ITEM = 60 (3) ITEM = 50 (4) ITEM = 33



(5) ITEM = 55 (6) ITEM = 11

Fig. 7.23

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 PAB := 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].

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, ..., 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, ..., 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 AK.

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

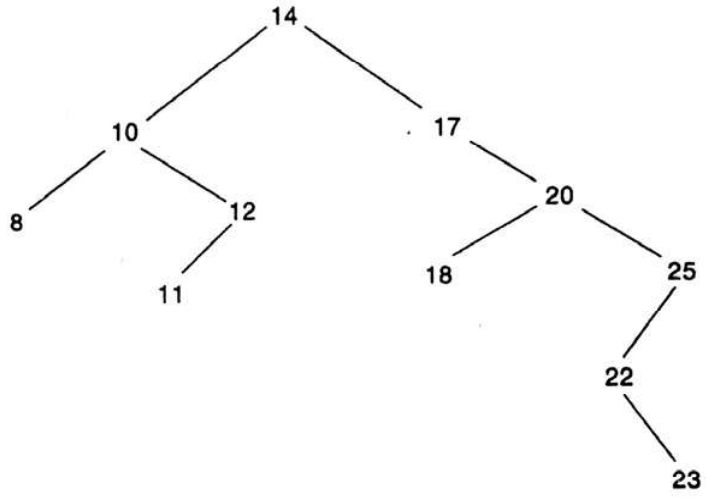


Fig. 7.24

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 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.

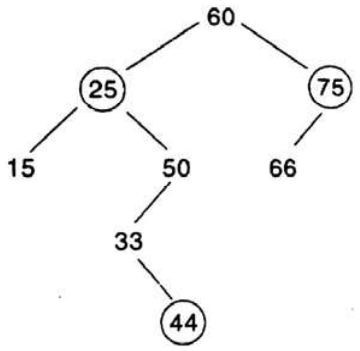
Case 2. N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

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.

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) Before deletions

ROOT: 3

AVAIL: 5

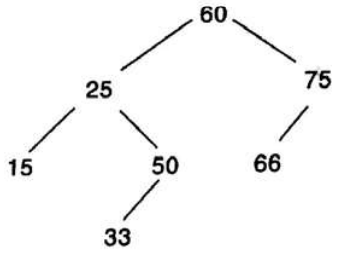
	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

(b) Linked representation

Fig. 7.25

(a) 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 44 is deleted

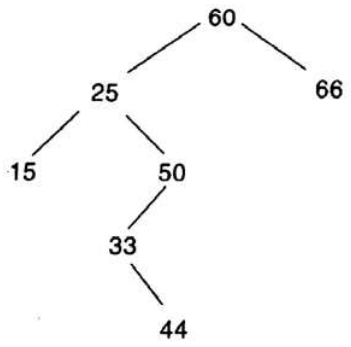
ROOT 3

AVAIL 9

	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		5	
10	50	1	0

(b) Linked representation

Fig. 7.26



(a) Node 75 is deleted

ROOT 3

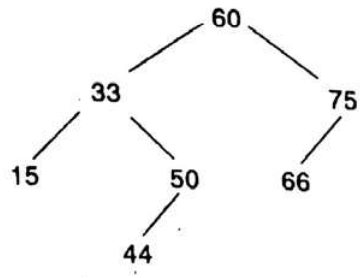
AVAIL 7

	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].



(a) Node 25 is deleted

ROOT 3

AVAIL 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

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.

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 AVL SEARCH TREES

The binary search tree data structure was discussed in sections 7.7–7.9. Consider the elements A, B, C, D,..., Z to be inserted into a binary search tree as shown in Fig. 7.29(a). Observe how the binary search tree turns out to be *right skewed*. Again the insertion of elements Z, Y, X,..., C, B, A in that order in the binary search tree results in a *left skewed binary search tree* (Fig. 7.29(b)). The disadvantage of a skewed binary search tree is that the worst case time complexity of a search is $O(n)$. Therefore there arises the need to maintain the binary search tree to be of balanced height. By doing so it is possible to obtain for the search operation a time complexity of $O(\log n)$ in the worst case. One of the more popular balanced trees was introduced in 1962 by Adelson–Velskii and Landis and was known as AVL trees.

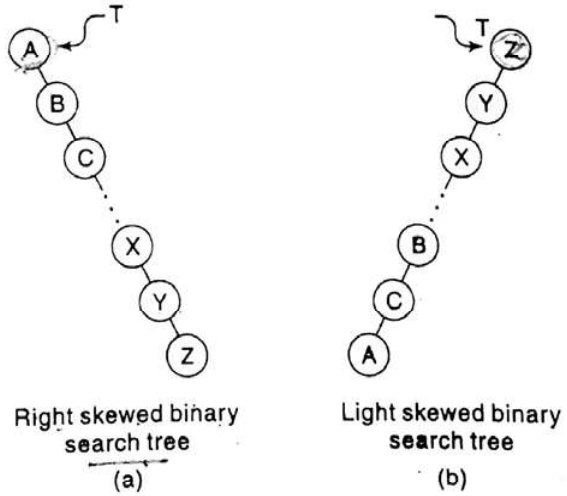


Fig. 7.29

Definition

An empty binary tree is an AVL tree. A non empty binary tree T is an AVL tree iff given T^L and T^R to be the left and right subtrees of T and $h(T^L)$ and $h(T^R)$ to be the heights of subtrees T^L and T^R respectively, T^L and T^R are AVL trees and $|h(T^L) - h(T^R)| \leq 1$.
$h(T^L) - h(T^R)$ is known as the *balance factor* (BF) and for an AVL tree the balance factor of a node can be either 0, 1, or –1.
An AVL search tree is a binary search tree which is an AVL tree.

Representation of an AVL Search Tree

AVL search trees like binary search trees are represented using a linked representation. However, every node registers its balance factor. Figure 7.30 shows the representation of an AVL search tree. The number against each node represents its balance factor.

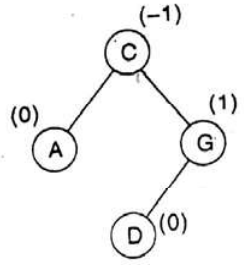


Fig. 7.30 *AVL*

Searching an AVL Search Tree

Searching an AVL search tree for an element is exactly similar to the method used in a binary search tree (Illustrated in procedure 7.4).

7.11 INSERTION IN AN AVL SEARCH TREE

Inserting an element into an AVL search tree in its first phase is similar to that of the one used in a binary search tree. However, if after insertion of the element, the balance factor of any node in the tree is affected so as to render the binary search tree unbalanced, we resort to techniques called *Rotations* to restore the balance of the search tree. For example, consider the AVL search tree shown in Fig. 7.30, inserting the element E into the tree makes it unbalanced the tree, since BF(C) = –2.

To perform rotations it is necessary to identify a specific node A whose BF(A) is neither 0, 1 or –1, and which is the nearest ancestor to the inserted node on the path from the inserted node to the root. This implies that all nodes on the path from the inserted node to A will have their balance factors to be either 0, 1, or –1. The rebalancing rotations are classified as LL, LR, RR and RL as illustrated below, based on the position of the inserted node with reference to A.

LL rotation: Inserted node is in the left subtree of left subtree of node A
RR rotation: Inserted node is in the right subtree of right subtree of node A
LR rotation: Inserted node is in the right subtree of left subtree of node A
RL rotation: Inserted node is in the left subtree of right subtree of node A

Each of the rotations is explained with an example.

LL Rotation

Figure 7.31 illustrates the balancing of an AVL search tree through LL rotation.

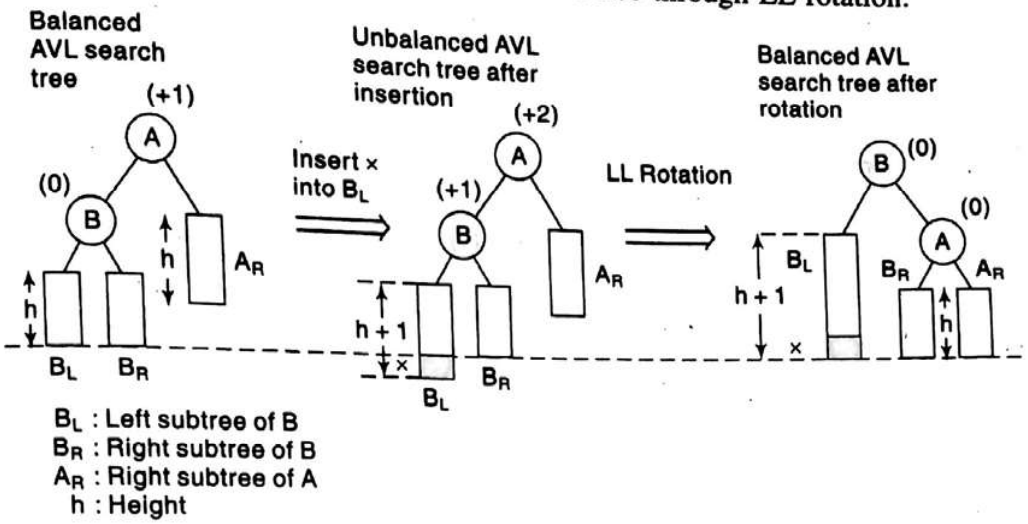


Fig. 7.31

The new element X is inserted in the left subtree of left subtree of A, the closest ancestor node whose BF(A) becomes +2 after insertion. To rebalance the search tree, it is rotated so as to allow B to be the root with B_L and A to be its left subtree and right child, and B_R and A_R to be the left and right subtrees of A. Observe how the rotation results in a balanced tree.

Example 7.19

Insert 36 into the AVL search tree given in Fig. 7.32(a). Figure 7.32(b) shows the AVL search tree after insertion and Fig. 7.32(c) the same after LL rotation.

RR Rotation

Figure 7.33 illustrates the balancing of an AVL search tree using RR rotation. Here the new element X is in the right subtree of A. The rebalancing rotation pushes B up to the root with A as its left child and B_R as its right subtree, and A_L and B_L as the left and right subtrees of A. Observe the balanced height after the rotation.

Example 7.20

Insert 65 into the AVL search tree shown in Fig. 7.34(a). Figure 7.34(b) shows the unbalanced tree after insertion and Fig. 7.34(c) the rebalanced search tree after RR rotation.

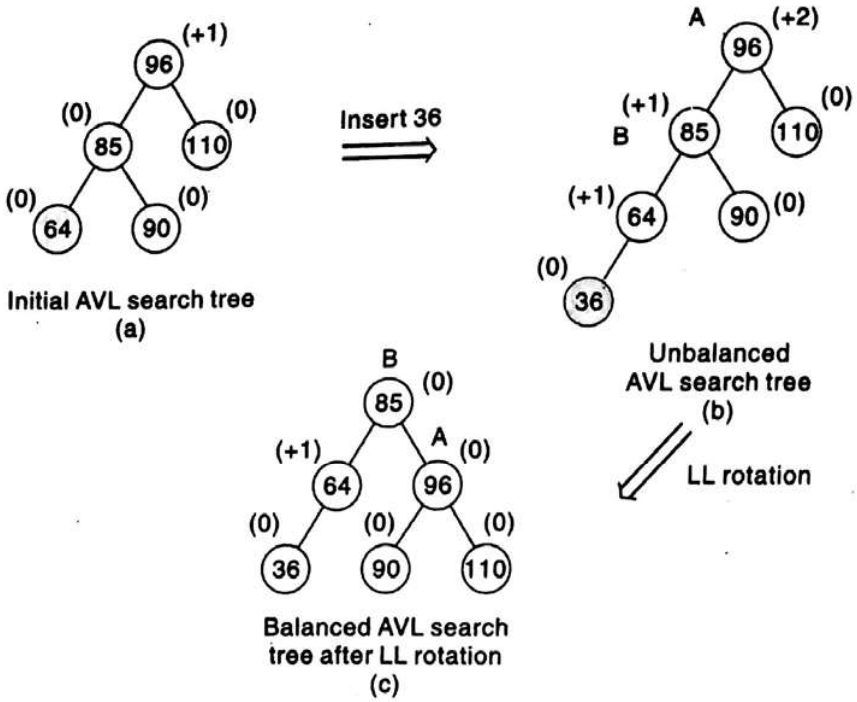


Fig. 7.32

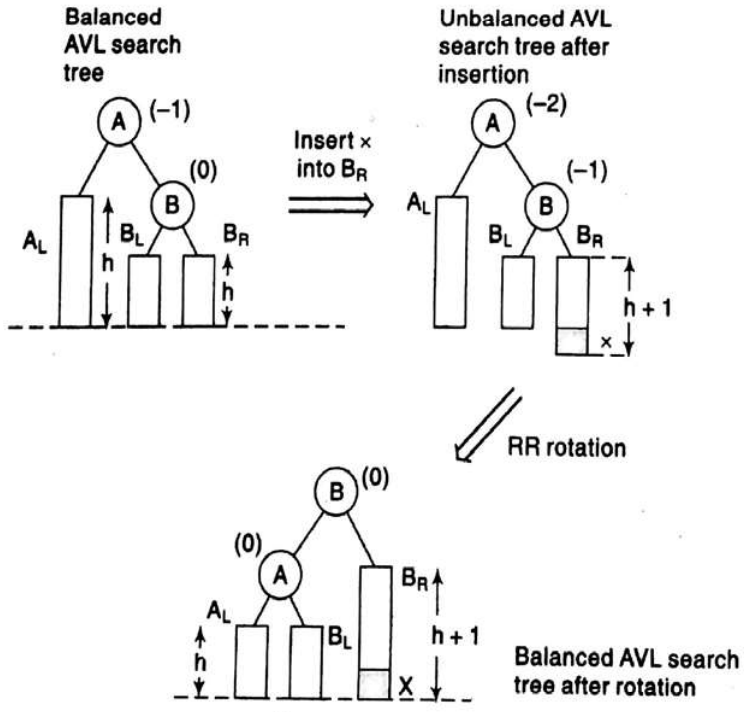


Fig. 7.33

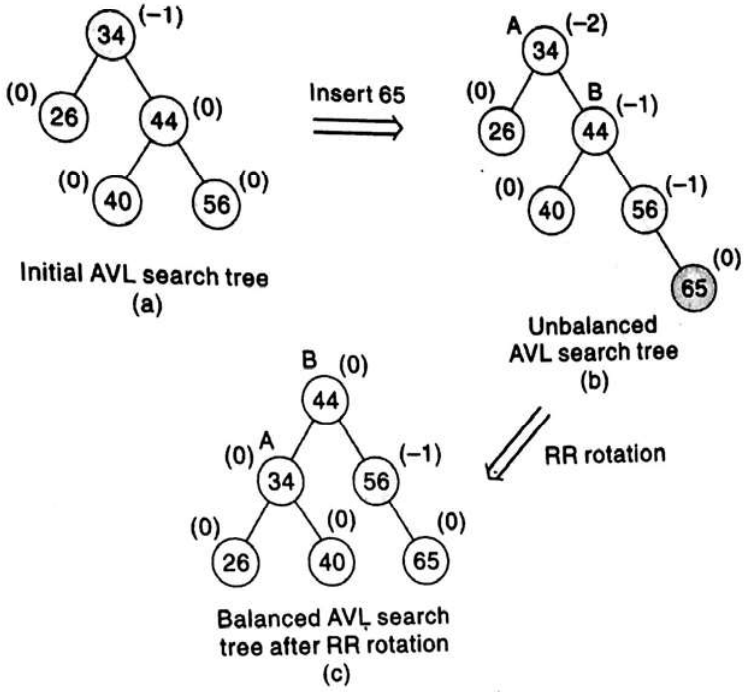


Fig. 7.34

LR and RL Rotations

The balancing methodology of LR and RL rotations are similar in nature but are mirror images of one another. Hence we illustrate one of them viz., LR rotation in this section. Fig. 7.35 illustrates the balancing of an AVL search tree using LR rotation.

In this case, the BF values of nodes A and B after balancing are dependent on the BF value of node C after insertion.

If BF (C) = 0 after insertion then BF(A) = BF(B)=0, after rotation
If BF (C) = –1 after insertion then BF(A) = 0, BF(B) = 1, after rotation
If BF (C) = 1 after insertion then BF(A) = –1, BF(B) = 0, after rotation

Example 7.21

Insert 37 into the AVL search tree shown in Fig. 7.36(a). Figure 7.36(b) shows the imbalance in the tree and Fig. 7.36(c) the rebalancing of the AVL tree using LR rotation. Observe how after insertion BF(C = 39) = 1 leads to BF(A = 44) = –1 and BF(B = 30) = 0 after the rotation.

Amongst the rotations, LL and RR rotations are called as *single rotations* and LR and RL are known as *double rotations* since, LR can be accomplished by RR followed by LL rotation and RL can be accomplished by LL followed by RR rotation. The time complexity of an insertion operation in an AVL tree is given by $O(height)=O(\log n)$.

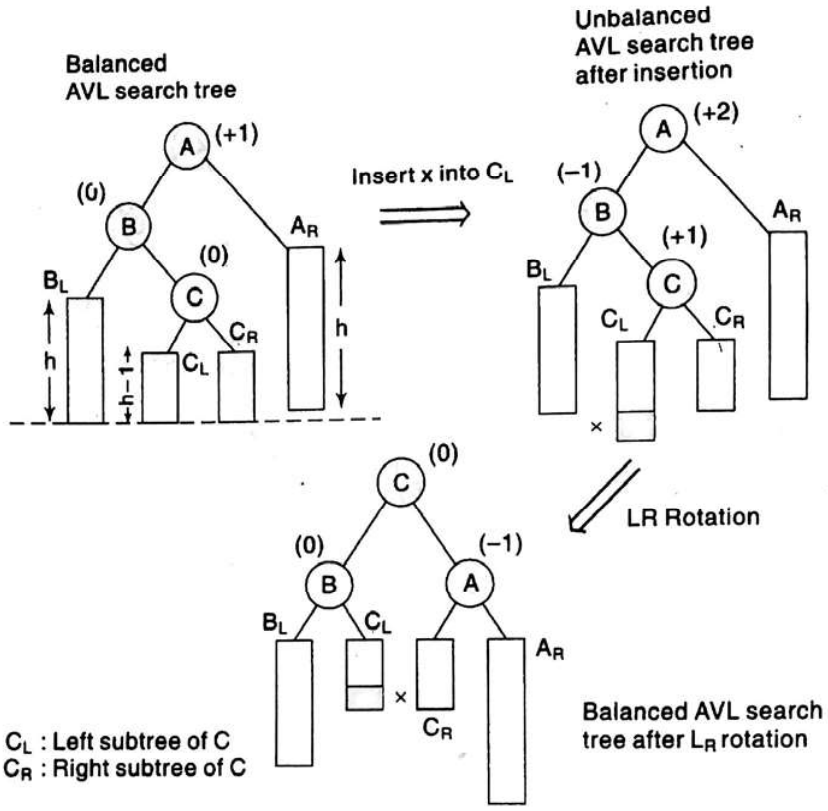


Fig. 7.35

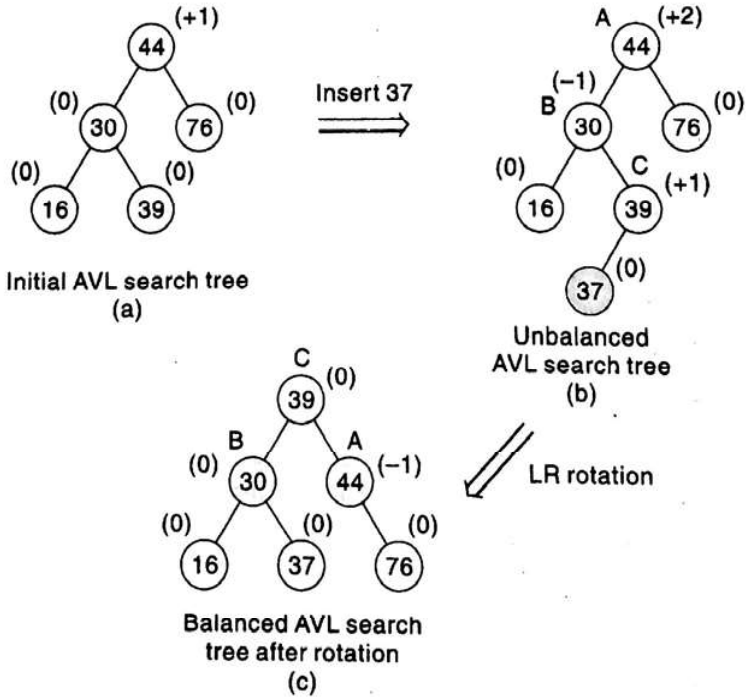


Fig. 7.36

EXAMPLE 7.22

Construct an AVL search tree by inserting the following elements in the order of their occurrence.

64, 1, 44, 26, 13, 110, 98, 85

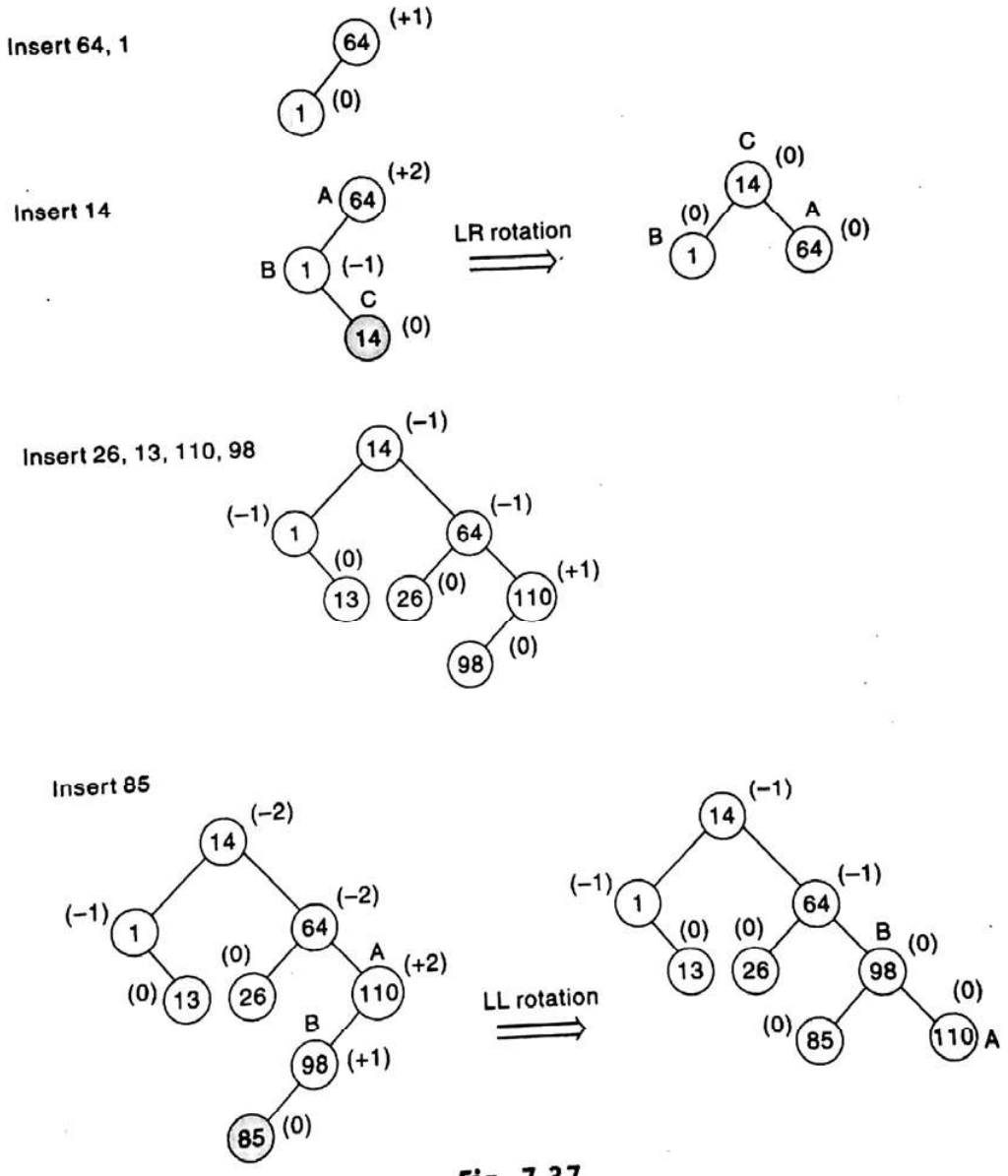


Fig. 7.37

7.12 DELETION IN AN AVL SEARCH TREE

The deletion of an element in an AVL search tree proceeds as illustrated in procedures 7.6, 7.7 and algorithm 7.8 for deletion of an element in a binary search tree. However, in the event of an imbalance due to deletion, one or more rotations need to be applied to balance the AVL tree.

On deletion of a node X from the AVL tree, let A be the closest ancestor node on the path from X to the root node, with a balance factor of +2 or –2. To restore balance the rotation is first classified as L or R depending on whether the deletion occurred on the left or right subtree of A.

Now depending on the value of BF(B) where B is the root of the left or right subtree of A, the R or L imbalance is further classified as R0, R1 and R-1 or L0, L1 and L-1. The three kinds of R rotations are illustrated with examples. The L rotations are but mirror images of these rotations.

R0 Rotation

If BF(B)=0, the R0 rotation is executed as illustrated in Fig. 7.38.

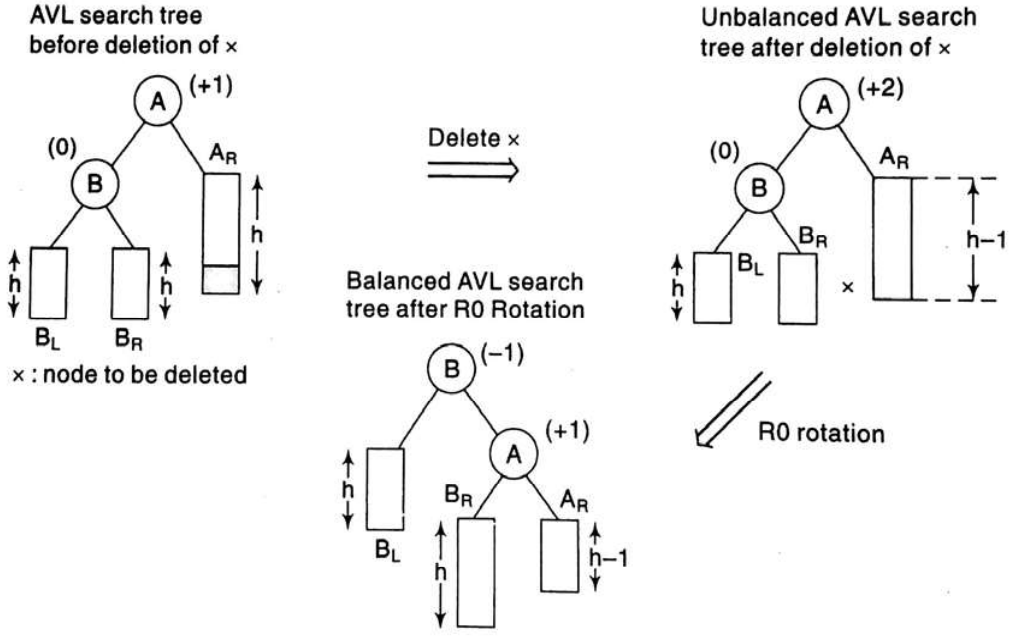


Fig. 7.38

Example 7.23

Delete 60 from the AVL search tree shown in Fig. 7.39(a). This calls for R0 rotation to set right the imbalance since deletion occurs to the right of node A = 46 whose BF(A = 46) = +2 and BF(B = 20) = 0. Figure 7.39(b) shows the imbalance after deletion and Fig. 7.39(c) the balancing of the tree using R0 rotation.

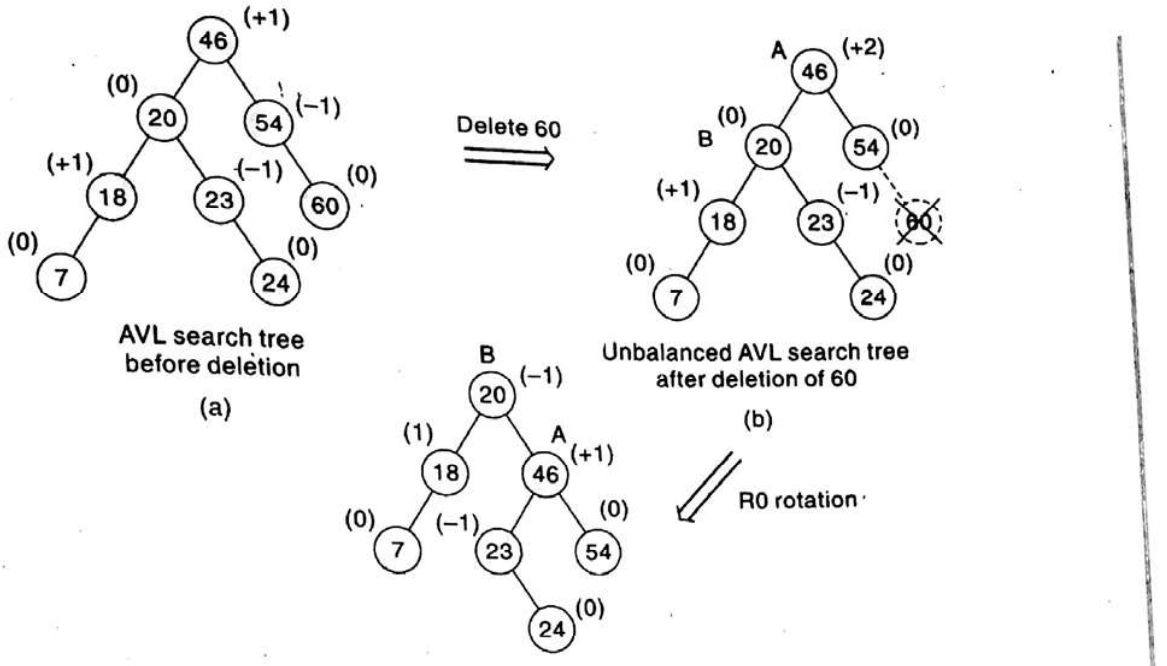


Fig. 7.39

R1 ROTATION

If BF(B) = 1, the R1 rotation as illustrated in Fig. 7.40 is executed.

Example 7.24

Delete 39 from the AVL search tree as shown in Fig. 7.41(a). This calls for R1 rotation to set right the imbalance since deletion occurs to the right of node A = 37 whose BF(A = 37) = +2 and BF(B = 26) = 1. Figure 7.41(b) shows the imbalance after deletion and Fig.7.41(c) the balancing of the tree using R1 rotation.

R-1 Rotation

If BF(B) = −1, the R−1 rotation as illustrated in Fig. 7.42 is executed.

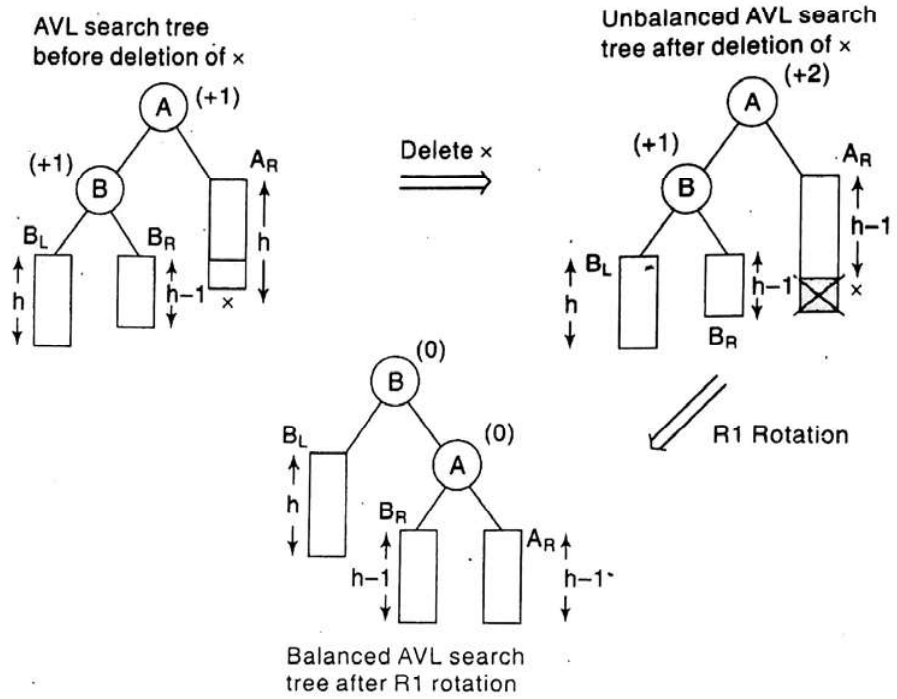


Fig. 7.40

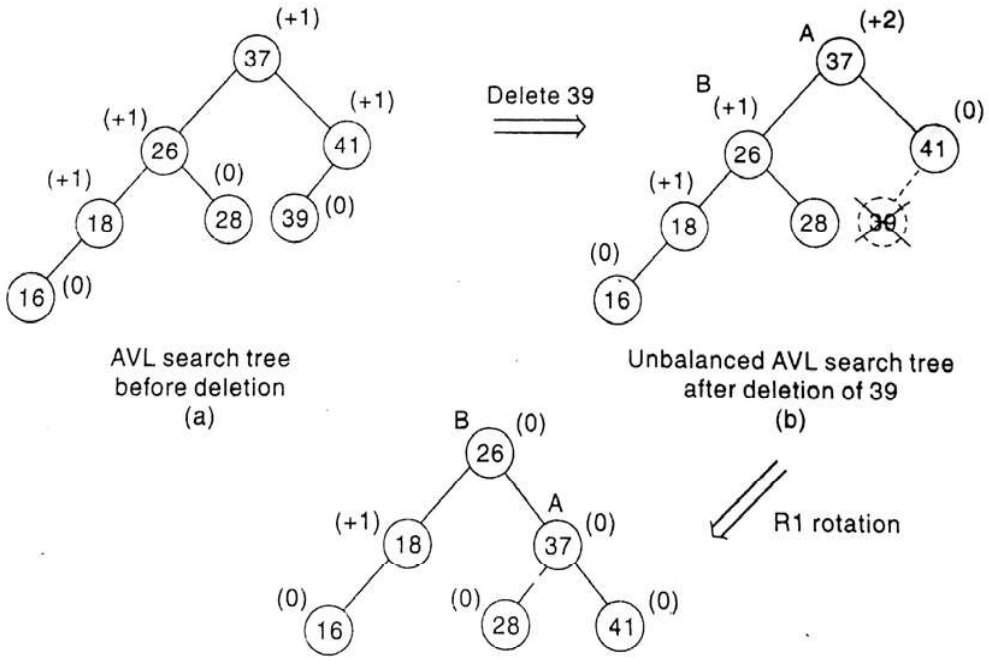


Fig. 7.41

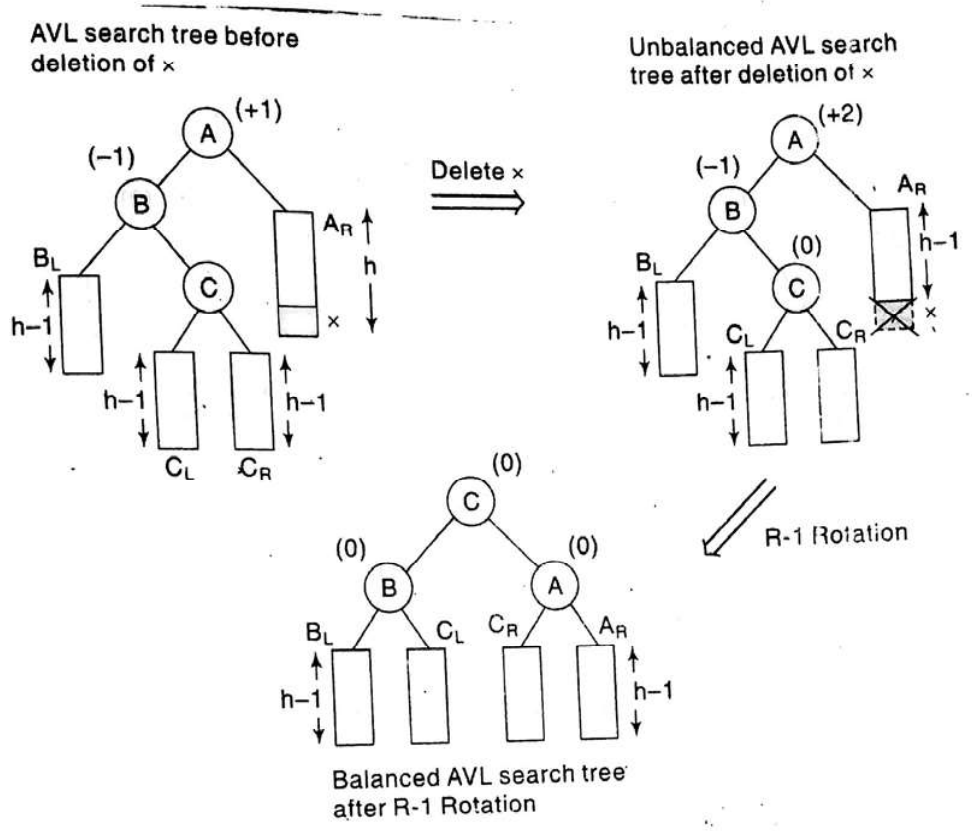


Fig. 7.42

Example 7.25

Delete 52 from the AVL search tree shown in Fig. 7.43(a). This calls for R-1 rotation to set right the imbalance since deletion occurs to the right of node A = 44 whose BF(A = 44) = +2 and BF(B = 22) = −1. Figure 7.43(b) shows the imbalance after deletion and Fig. 7.43(c) the balancing of the tree using R-1 rotation.

EXAMPLE 7.26

Given the AVL search tree shown in Fig. 7.44(a) delete the listed elements:

120, 64, 130

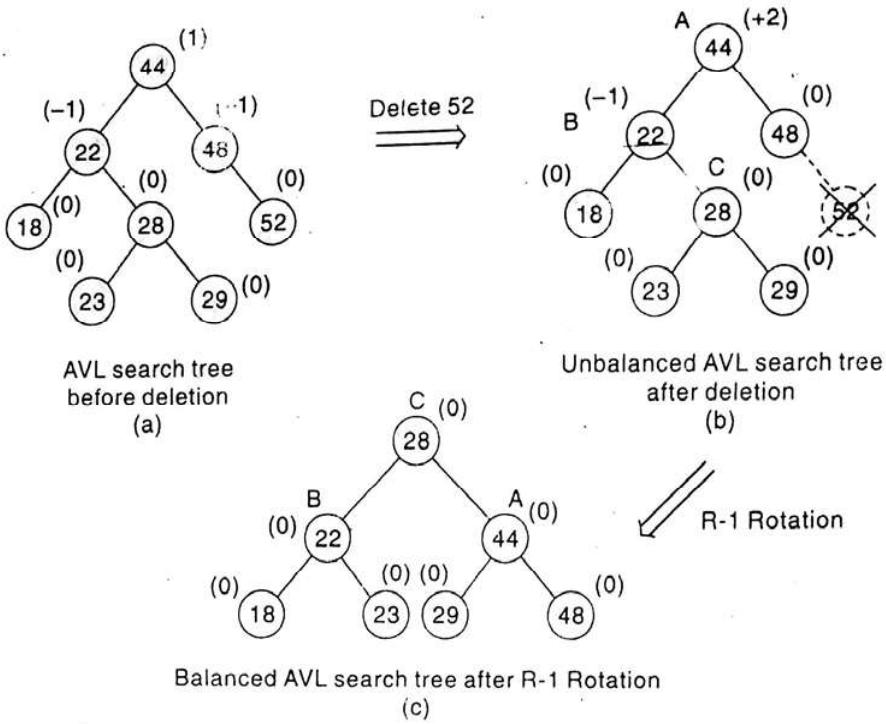


Fig. 7.43

7.13 *m*-WAY SEARCH TREES

All the data structures discussed so far favor data stored in the internal memory and hence support internal information retrieval. However, to favor retrieval and manipulation of data stored in external memory, viz., storage devices such as disks etc., there is a need for some special data structures. *m*-way search trees, B trees and B^+ trees are examples of such data structures which find application in problems such as file indexing.

m-way search trees are generalized versions of binary search trees. The goal of *m*-way search tree is to minimize the accesses while retrieving a key from a file. However, an *m*-way search tree of height *h* calls for $O(h)$ number of accesses for an insert/delete/retrieval operation. Hence it pays to ensure that the height *h* is close to $\log_m(n+1)$, because the number of elements in an *m*-way search tree of height *h* ranges from a minimum of *h* to a maximum of $m^h - 1$. This implies that an *m*-way search tree of *n* elements ranges from a minimum height of $\log_m(n+1)$ to a maximum height of *n*. Therefore there arises the need to maintain balanced *m*-way search trees. B trees are balanced *m*-way search trees.

Definition

An *m*-way search tree T may be an empty tree. If T is non empty, it satisfies the following properties:

(i) For some integer *m*, known as *order of the tree*, each node is of degree which can reach a maximum of *m*, in other words, each node has, at most *m* child nodes. A node may be

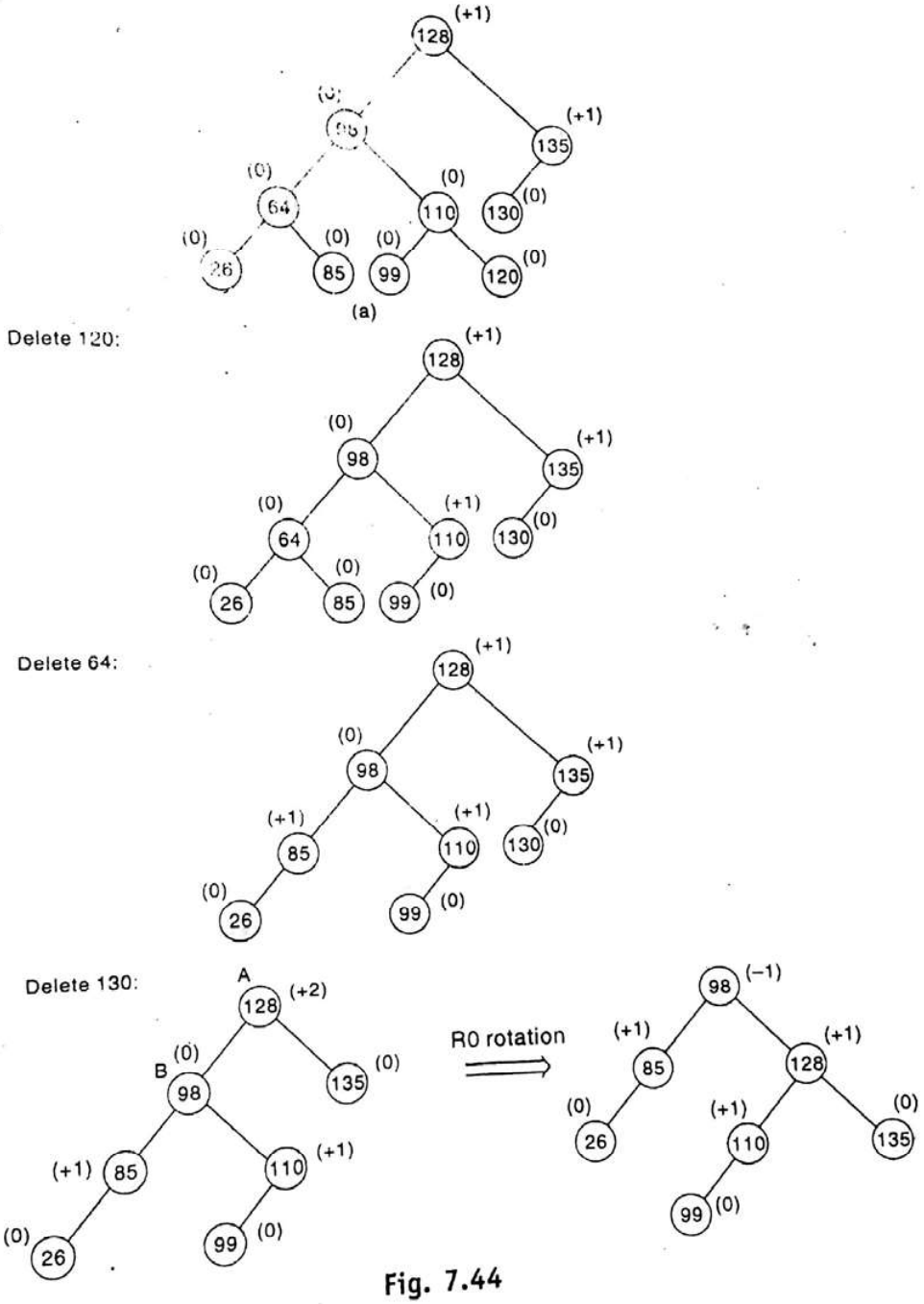


Fig. 7.44

represented as A_0, (K_1, A_1), $(K_2, A_2) \ldots (K_{m-1}, A_{m-1})$ where, K_i, $1 \le i \le m-1$ are the keys and A_i, $0 \le i \le m-1$ are the pointers to subtrees of T.

(ii) If a node has k child nodes where $k \le m$, then the node can have only $(k-1)$ keys K_1, K_2, ...K_{k-1}, contained in the node such that $K_i < K_{i+1}$ and each of the keys partitions all the keys in the subtrees into k subsets.

(iii) For a node A_0, (K_1, A_1), (K_2, A_2)....(K_{m-1}, A_{m-1}), all key values in the subtree pointed to by A_i are less than the key K_{i+1}, $0 \le i \le m-2$, and all key values in the subtree pointed to by A_{m-1} are greater than K_{m-1}

(iv) Each of the subtrees A_i, $0 \le i \le m-1$ are also m-way search trees.

An example of a 5-way search tree is shown in Fig. 7.45. Observe how each node has at most 5 child nodes and therefore has at most 4 keys contained in it. The rest of the properties of an m-way search tree can also be verified by the example.

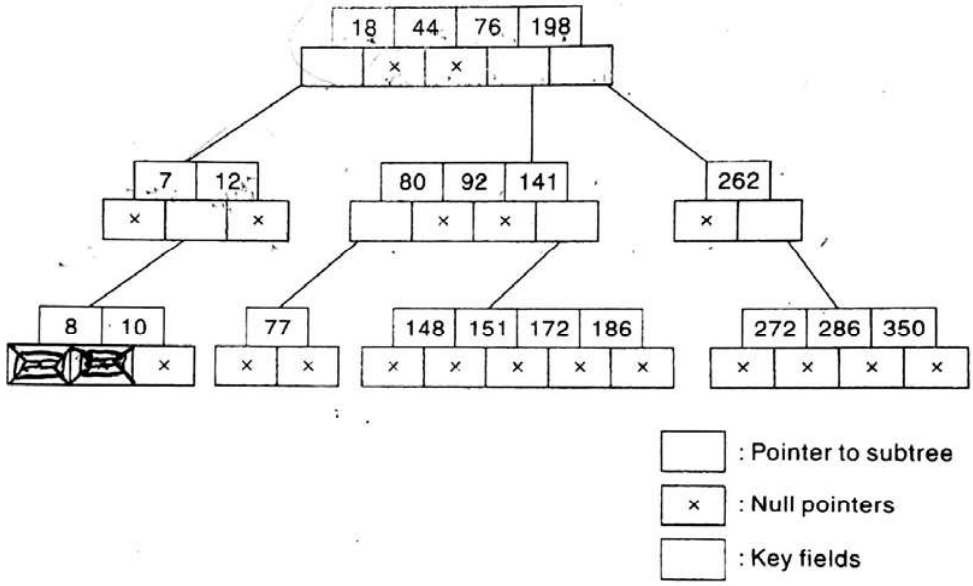


Fig. 7.45

7.14 SEARCHING, INSERTION AND DELETION IN AN *m*-WAY SEARCH TREE

Searching an *m*-Way Search Tree

Searching for a key in an m-way search tree is similar to that of binary search trees. To search for 77 in the 5-way search tree, shown in Fig. 7.45, we begin at the root and as $77 > 76 > 44 > 18$, move to the fourth subtree. In the root node of the fourth subtree, $77 < 80$ and therefore we move to the first subtree of the node. Since 77 is available in the only node of this subtree, we claim 77 was successfully searched.

To search for 13, we begin at the root and as $13 < 18$, we move down to the first subtree. Again, since $13 > 12$ we proceed to move down the third subtree of the node but fall off the tree having encountered a null pointer. Therefore we claim the search for 13 was unsuccessful.

Insertion in an *m*-Way Search Tree

To insert a new element into an m-way search tree we proceed in the same way as one would in

order to search for the element. To insert 6 into the 5-way search tree shown in Fig.7.45, we proceed to search for 6 and find that we fall off the tree at the node [7, 12] with the first child node showing a null pointer. Since the node has only two keys and a 5-way search tree can accommodate up to 4 keys in a node, 6 is inserted into the node as [6, 7, 12]. But to insert 146, the node [148, 151, 172, 186] is already full, hence we open a new child node and insert 146 into it. Both these insertions have been illustrated in Fig. 7.46.

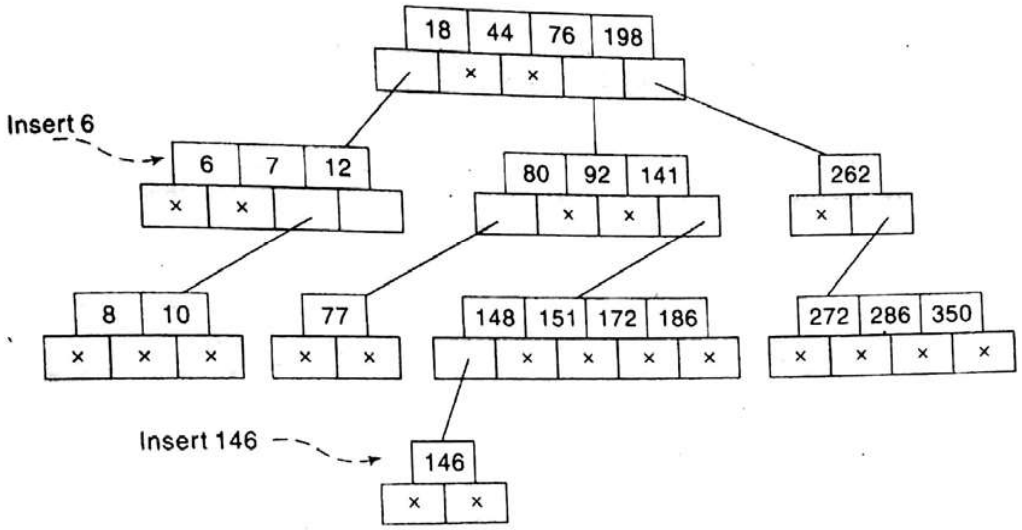


Fig. 7.46

Deletion in an *m*-Way Search Tree

Let K be a key to be deleted from the *m*-way search tree. To delete the key we proceed as one would to search for the key. Let the node accommodating the key be as illustrated in Fig. 7.47.

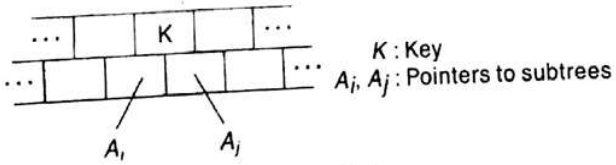


Fig. 7.47

If ($A_i = A_j = NULL$) then delete K.

If ($A_i \neq NULL$, $A_j = NULL$) then choose the largest of the key elements K′ in the child node pointed to by A_i, delete the key K′ and replace K by K′. Obviously deletion of K′ may call for subsequent replacements and therefore deletions in a similar manner, to enable the key K′ move up the tree.

If ($A_i = NULL$, $A_j \neq NULL$) then choose the smallest of the key elements K″ from the subtree pointed to by A_j, delete K″ and replace K by K″. Again deletion of K″ may trigger subsequent replacements and deletions to enable K″ move up the tree.

If ($A_i \neq NULL$, $A_j \neq NULL$) then choose either the largest of the key elements K′ in the subtree pointed to by A_i or the smallest of the key elements K″ from the subtree pointed to by A_j to replace K. As mentioned before, to move K′ or K″ up the tree it may call for subsequent replacements and deletions.

We illustrate deletions on the 5-way search tree shown in Fig. 7.45. To delete 151, we search for 151 and observe that in the leaf node [148, 151, 172, 186] where it is present, both its left subtree pointer and right subtree pointer are such that ($A_i = A_j = NULL$). We therefore simply delete 151 and the node becomes [148, 172, 186] . Deletion of 92 also follows a similar process.

To delete 262, we find its left and right subtree pointers A_i and A_j respectively, are such that ($A_i = NULL$, $A_j \neq NULL$). Hence we choose the smallest element 272 from the child node [272, 286, 300], delete 272 and replace 262 with 272. Note that, to delete 272 the deletion procedure needs to be observed again. Figure 7.48 illustrates the deletion of 262.

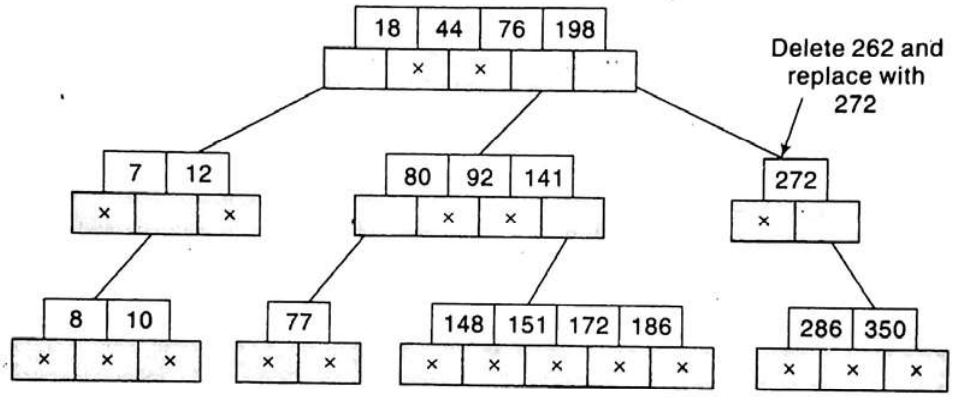


Fig. 7.48

To delete 12, we find the node [7, 12] accommodates 12 and the key satisfies ($A_i \neq NULL$, $A_j = NULL$). Hence we choose the largest of the keys from the node pointed to by A_i, viz., 10 and replace 12 with 10. Figure 7.49 illustrates the deletion of 12.

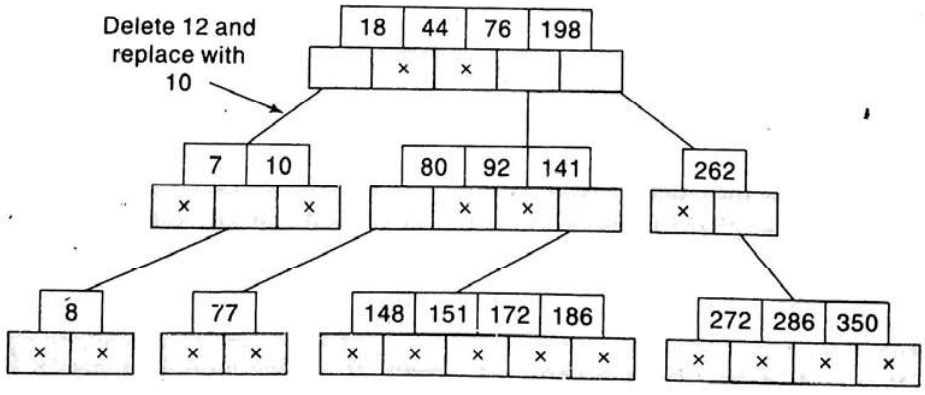


Fig. 7.49

Example 7.27

A 3-way search tree constructed out of an empty search tree with the following keys in the order shown, is illustrated in Fig. 7.50:

D, K, P, V, A, G

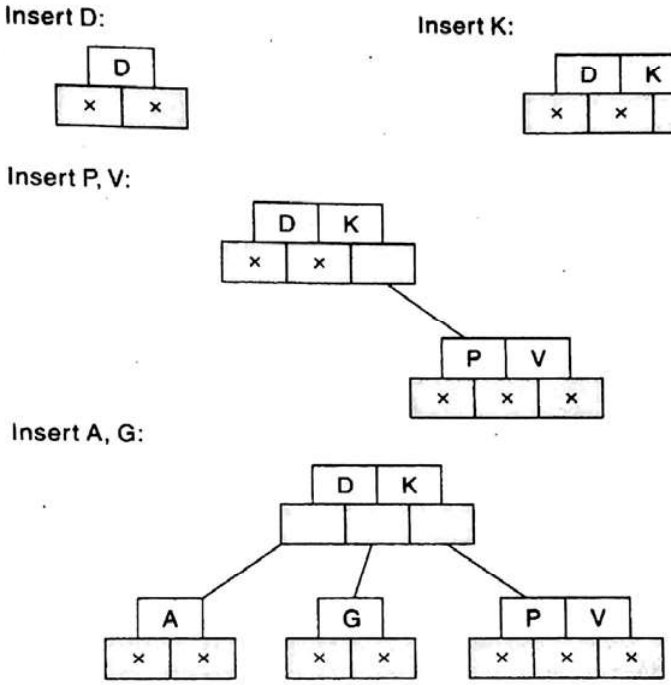


Fig. 7.50

Deletion of A and K in the 3-way search tree of Fig. 7.51 yields:

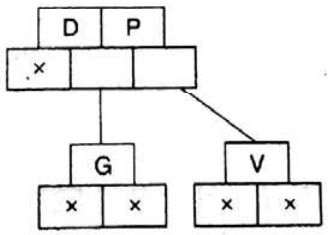


Fig. 7.51

7.15 B TREES

m-way search trees have the advantage of minimizing file accesses due to their restricted height. However it is essential that the height of the tree be kept as low as possible and therefore there

arises the need to maintain balanced m-way search trees. Such a balanced m-way search tree is what is defined as a B tree.

Definition

A B tree of order m, if non empty, is an m-way search tree in which:

(i) the root has at least two child nodes and at most m child nodes

(ii) the internal nodes except the root have at least $\left\lceil \frac{m}{2} \right\rceil$ child nodes and at most m child nodes.

(iii) the number of keys in each internal node is one less than the number of child nodes and these keys partition the keys in the subtrees of the node in a manner similar to that of m-way search trees.

(iv) all leaf nodes are on the same level

A B-tree of order 3 is referred to as 2–3 tree since the internal nodes are of degree 2 or 3 only.

Example 7.28

The tree shown in Fig. 7.52 is a B tree of order 5. All the properties of the B tree can be verified on the tree.

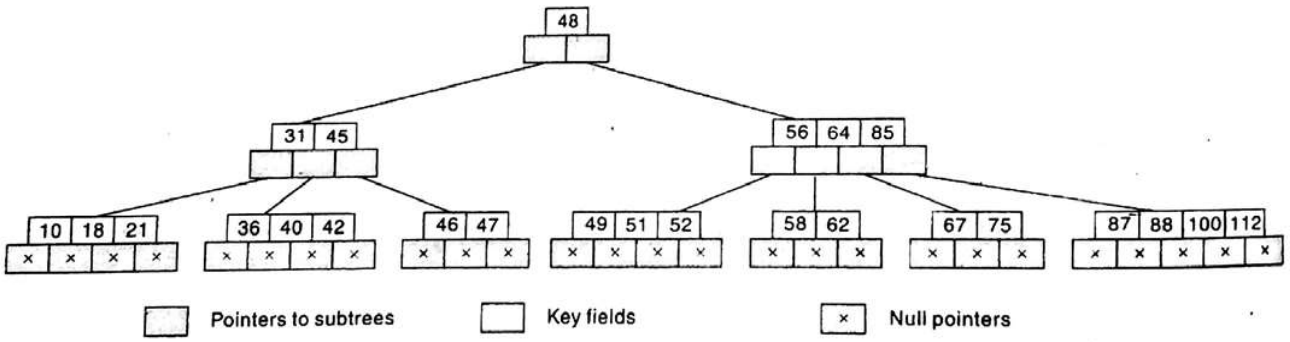


Fig. 7.52

7.16 SEARCHING, INSERTION AND DELETION IN A B-TREE

Searching a B-tree

Searching for a key in a B-tree is similar to the one on an m-way search tree. The number of accesses depends on the height h of the B-tree.

Insertion in a B-tree

The insertion of a key in a B-tree proceeds as if one were searching for the key in the tree. When the search terminates in a failure at a leaf node and tends to fall off the tree, the key is inserted according to the following procedure:

If the leaf node in which the key is to be inserted is not full, then the insertion is done in the node. A node is said to be full if it contains a maximum of $(m - 1)$ keys, given the order of the B-tree to be m.

If the node were to be full, then insert the key in order into the existing set of keys in the node, split the node at its median into two nodes at the same level, pushing the median element up by one level. Note that the split nodes are only half full. Accommodate the median element in the parent node if it is not full. Otherwise repeat the same procedure and this may even call for rearrangement of the keys in the root node or the formation of a new root itself.

Thus a major observation pertaining to insertion in a B-tree is that, since the leaf nodes are all at the same level, unlike m-way search trees, the tree grows upwards.

Example 7.29

Consider the B-tree of order 5 shown in Fig. 7.53. Insert the elements 4, 5, 58, 6 in the order given.

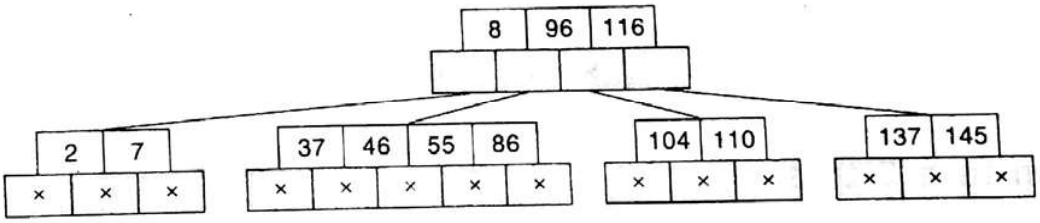


Fig. 7.53

The insertion of the given elements is shown in Fig. 7.54. Note how insertion of 4, 5 is easily done in the first child node of the root since the node is not full. But during the insertion of 58, the second child node of the root, where the insertion needs to be done is full. Now we insert the key into the list of keys existing in the node in the sorted order and split the node into two nodes at the same level at its median, viz., 55. Push the key 55 up to accommodate it in the parent node which is not full. Thus the key 58 is inserted.

The insertion of 6 is interesting. The first child node of the root where its place is due, is full. Now the node splits at the median pushing 5 up. But the parent node viz., the root, is also full. This in turn forces a split in the root node resulting in the creation of a new root which is 55.

It needs to be highlighted that all the keys in the node should be ordered every time there is a rearrangement of keys in a node.

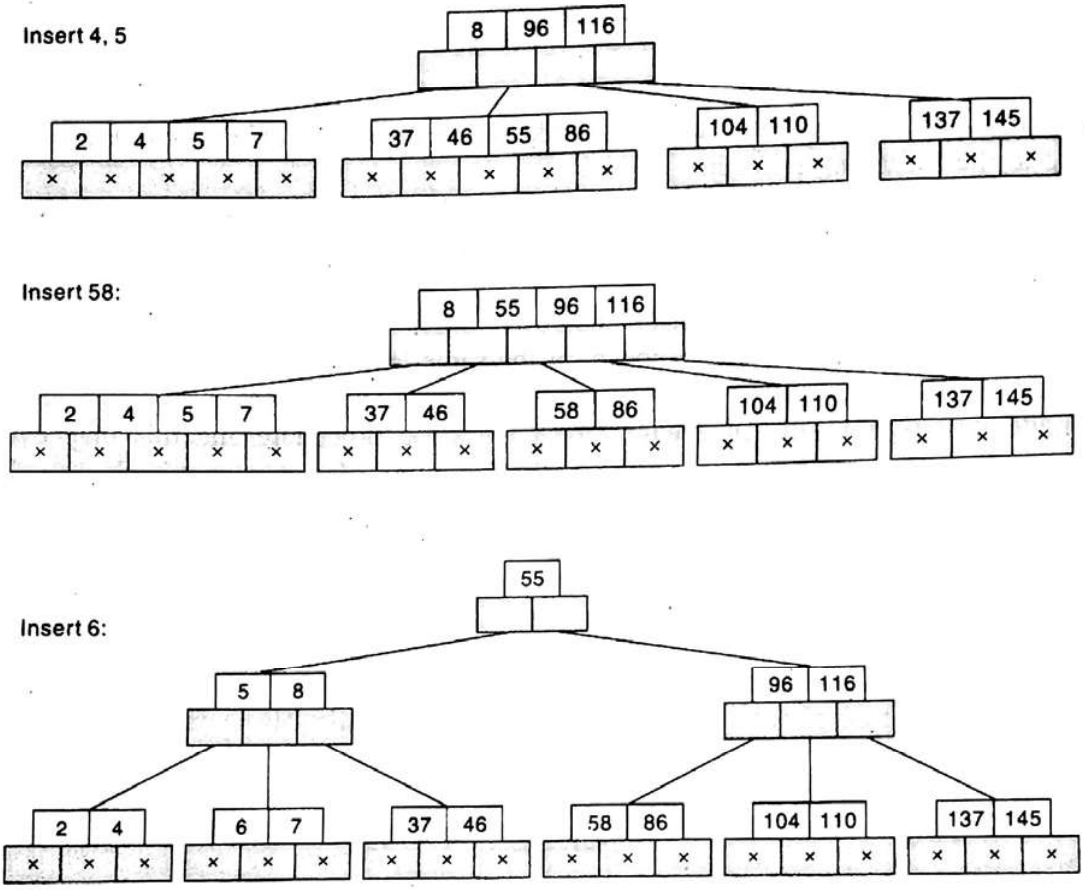


Fig. 7.54

Deletion in a B-tree

While deleting a key it is desirable that the keys in the leaf node are removed. However when a case arises forcing a key that is in an internal node to be deleted, then we promote a successor or a predecessor of the key to be deleted, to occupy the position of the deleted key and such a key is bound to occur in a leaf node (How?).

While removing a key from a leaf node, if the node contains more than the minimum number of elements, then the key can be easily removed. However, if the leaf node contains just the minimum number of elements, then scout for an element from either the left sibling node or right sibling node to fill the vacancy. If the left sibling node has more than the minimum number of keys, pull the largest key up into the parent node and move down the intervening entry from the parent node to the leaf node where the key is to be deleted. Otherwise, pull the smallest key of the right sibling node to the parent node and move down the intervening parent element to the leaf node.

If both the sibling nodes have only minimum number of entries, then create a new leaf node out of the two leaf nodes and the intervening element of the parent node, ensuring that the total number does not exceed the maximum limit for a node. If while borrowing the intervening element from the parent node, it leaves the number of keys in the parent node to be below the minimum number, then we propagate the process upwards ultimately resulting in a reduction of height of the B-tree.

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

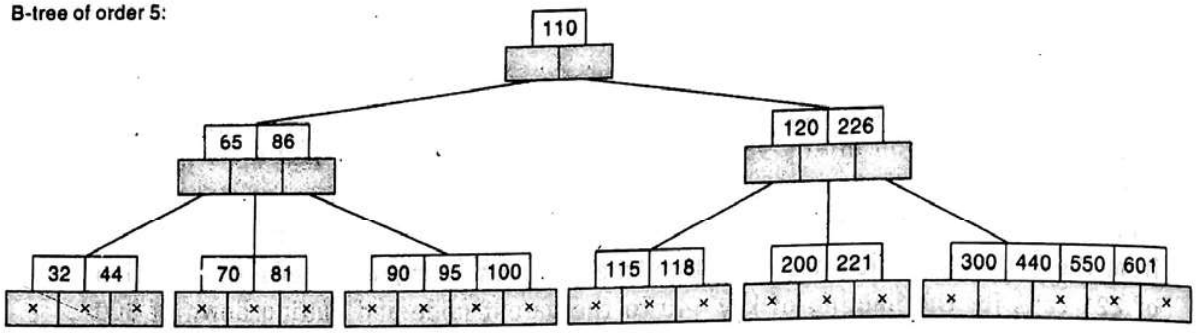
The B-tree after the operations is shown in Fig. 7.57

7.17 HEAP; HEAPSORT

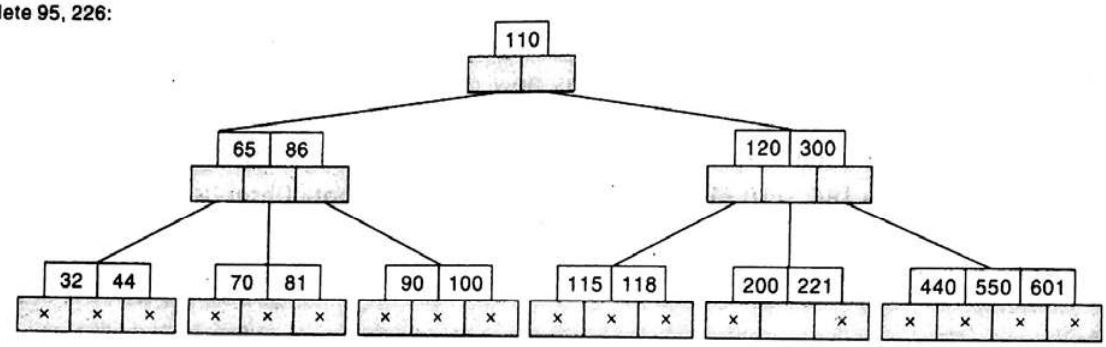
This section discusses another tree structure, called a *heap*. The heap is used in an elegant sorting algorithm called *heapsort*. Although sorting will be treated mainly in Chapter 9, we give the heapsort algorithm here and compare its complexity with that of the bubble sort and quicksort algorithms,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

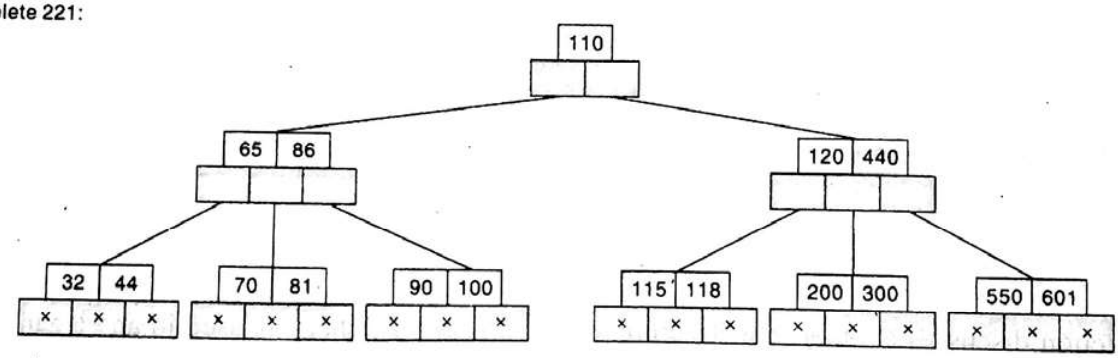
B-tree of order 5:



Delete 95, 226:



Delete 221:



Delete 70:

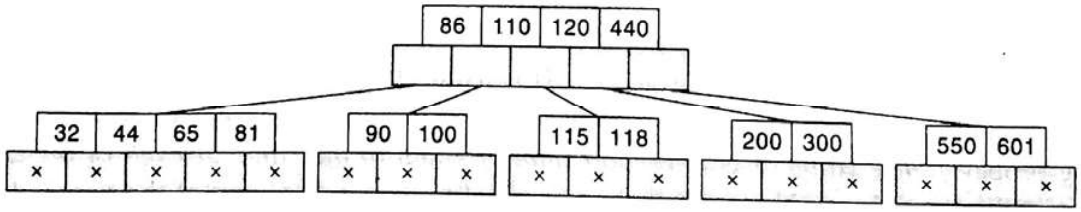


Fig. 7.55

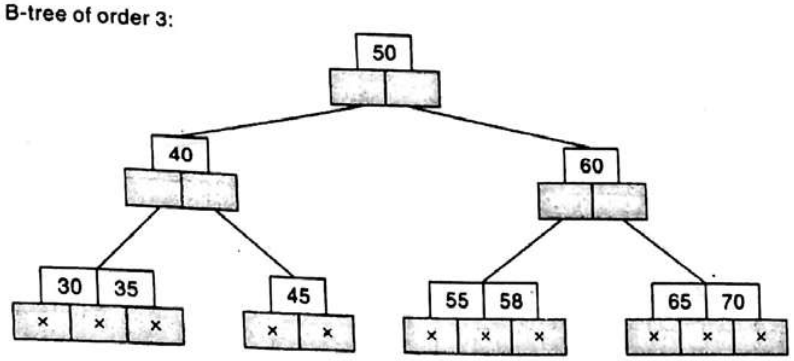


Fig. 7.56

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:

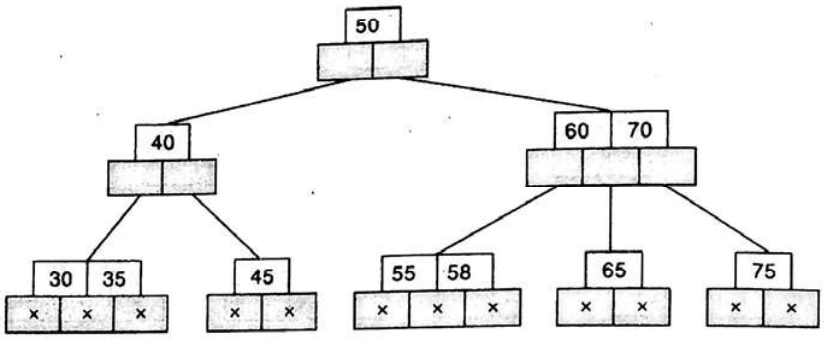
(1) First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.
(2) Then let ITEM rise to its "appropriate place" in H so that H is finally a heap.

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

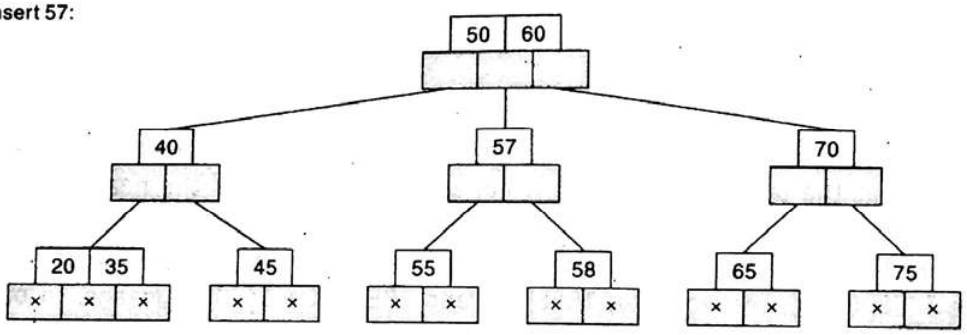
Example 7.33

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:

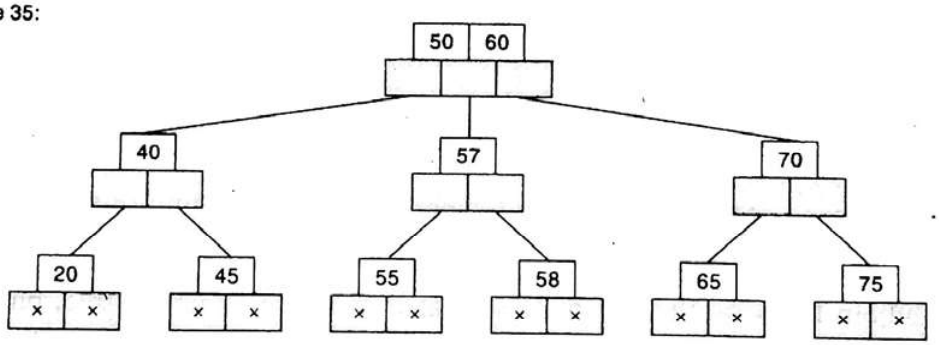
Insert 75:



Insert 57:



Delete 35:



Delete 65:

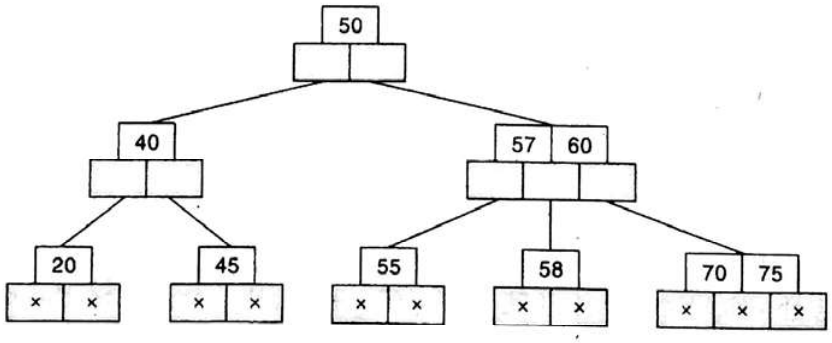
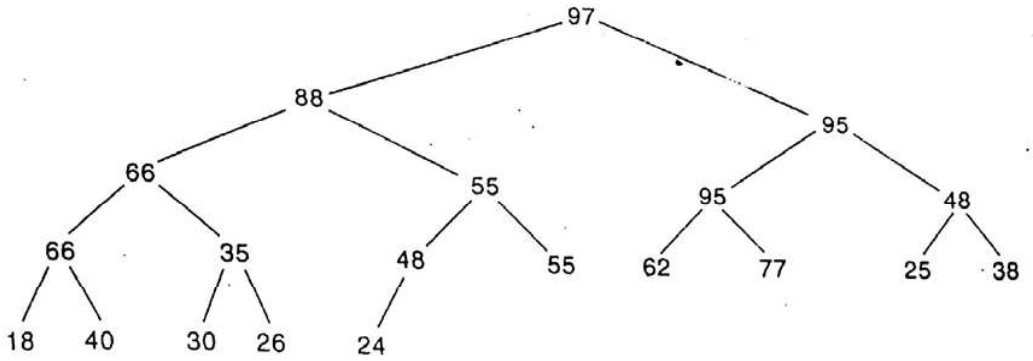


Fig. 7.57



(a) Heap

TREE

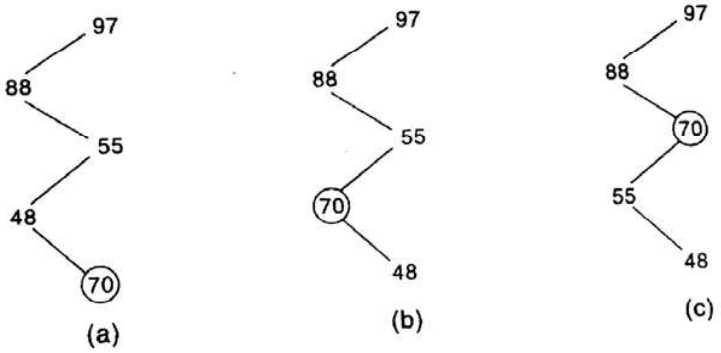
97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(b) Sequential representation

Fig. 7.58

(a) Compare 70 with its parent, 48. Since 70 is greater than 48, interchange 70 and 48; the path will now look like Fig. 7.59(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. 7.59(c).



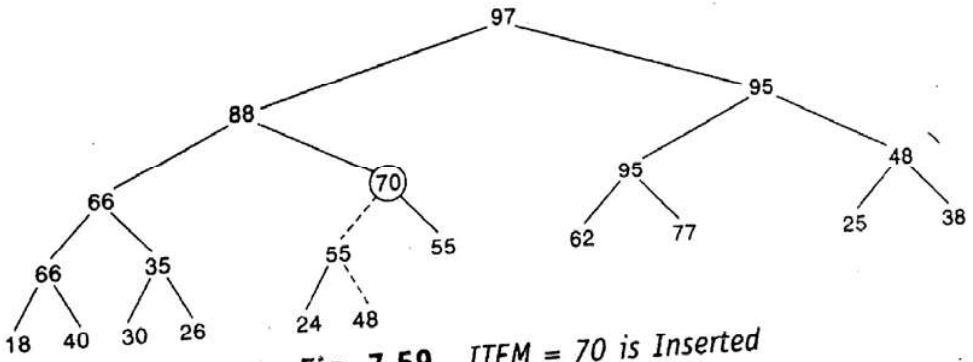


Fig. 7.59 *ITEM = 70 is Inserted*

(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, that is, 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.

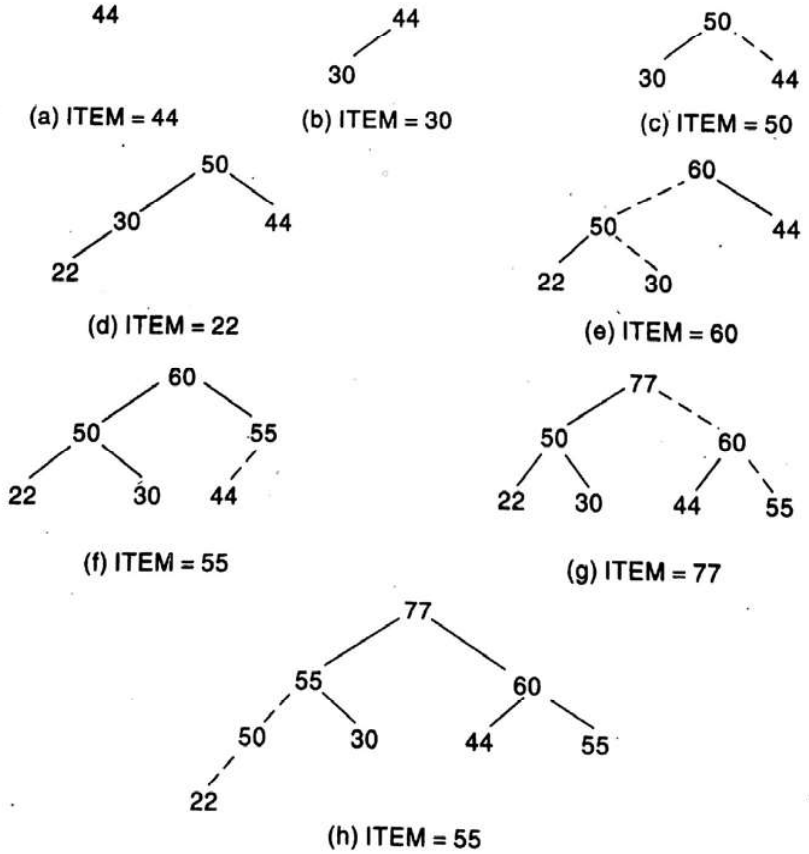


Fig. 7.60 *Building a Heap*

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[I] := 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 7.35

Consider the heap H in Fig. 7.61(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. 7.61(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:

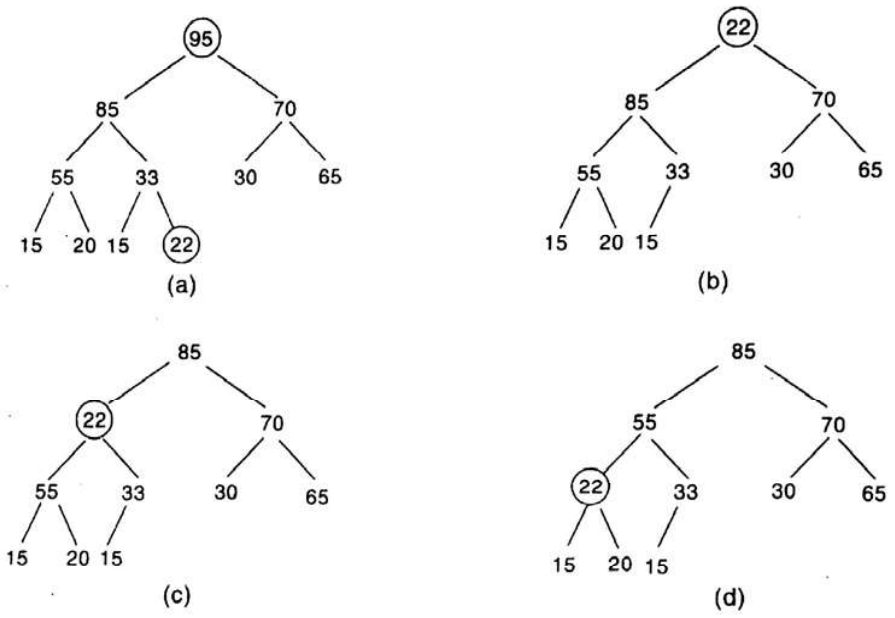


Fig. 7.61 *Reheaping*

(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. 7.61(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. 7.61(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. 7.61(d) is the required heap H without its original root R.

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 Chapter 9.

7.18 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 7.62 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 more than the number N_I of internal nodes; that is,

$$N_E = N_I + 1$$

For example, for the 2-tree in Fig. 7.62, $N_I = 6$, and $N_E = N_I + 1 = 7$.

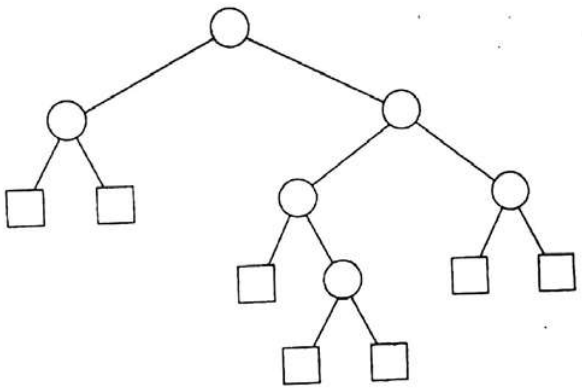


Fig. 7.62

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. 7.62,

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

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.,

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

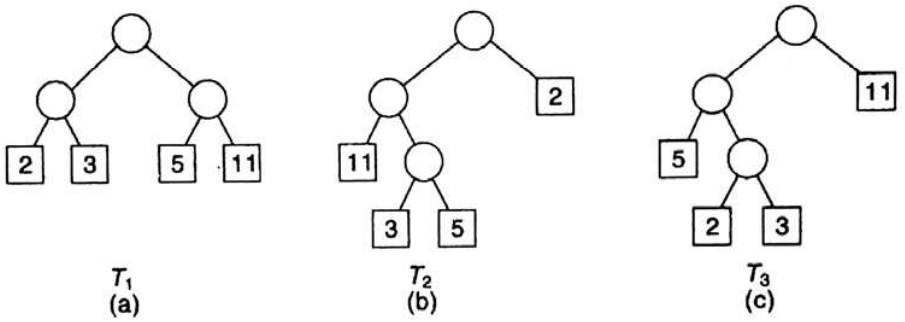
where W_i and L_i denote, respectively, the weight and path length of an external node N_i.

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 7.36

Figure 7.63 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:

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



T_1 (a) T_2 (b) T_3 (c)

Fig. 7.63

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.

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

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

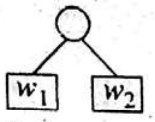
Among all the 2-trees with n external nodes and with the given n weights, find a tree T with a minimum-weighted path length. (Such a tree T is seldom unique.) Huffman gave an algorithm, which we now state, to find such a tree T.

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 7.37

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 7.64(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. 7.64(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 7.37. 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 7.65(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.

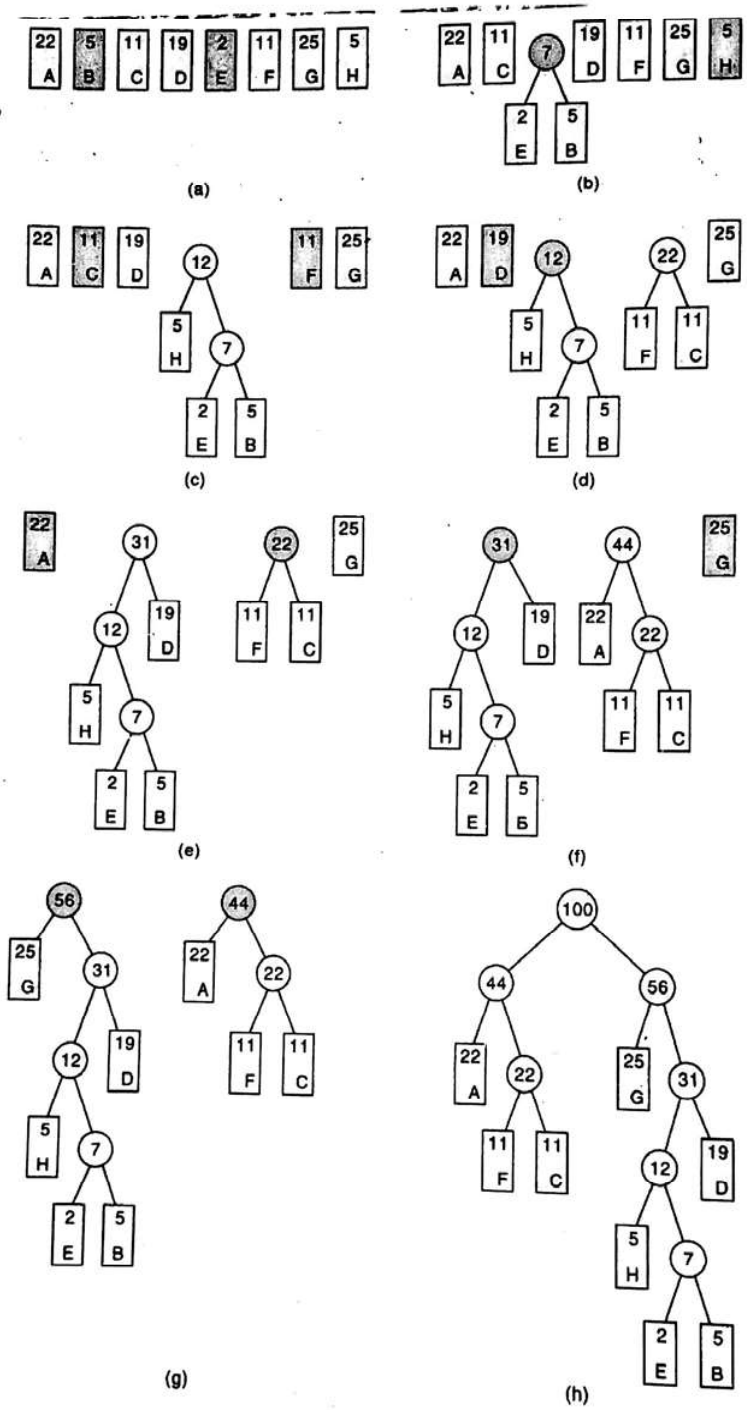


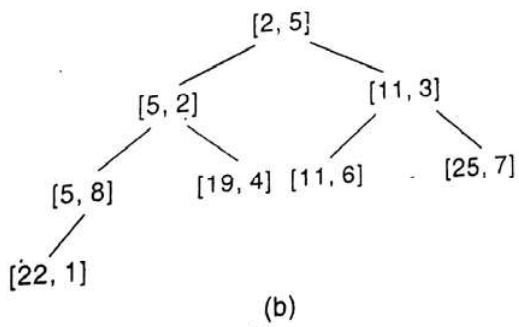
Fig. 7.64 *Building a Huffman Tree*

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. 7.65(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.)

	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)

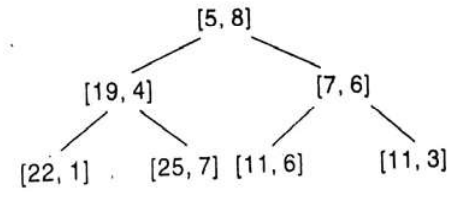


(b)

	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)



(d)

Fig. 7.65 *Implementation of Huffman's Algorithmn*

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 LEFT[9] = 5 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.

The shaded area in Fig. 7.65(c) shows the new node, and Fig. 7.65(d) shows the new heap, which has one less element than the heap in Fig. 7.65(b).

Repeating the above step until the heap is empty, we obtain the required tree T in Fig. 7.65(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

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

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 an 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.

Consider the extended binary tree T in Fig. 7.66 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 the 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. 7.66 determines the following code for the external nodes:

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

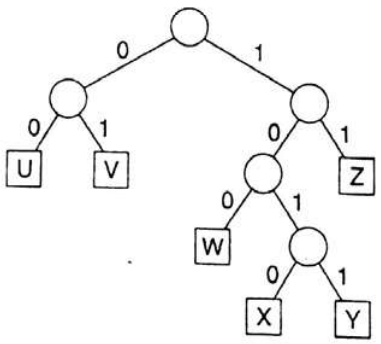


Fig. 7.66

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 7.37. Suppose the weights represent the percentage probabilities that the items will occur. Then the tree T of minimum-weighted path length constructed in Fig. 7.64, appearing with the bit labels in Fig. 7.67, 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

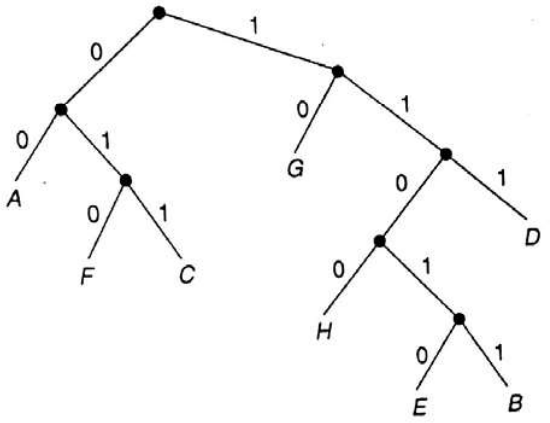


Fig. 6.67

7.19 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 7.38

Figure 7.68 pictures a general tree T with 13 nodes,

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

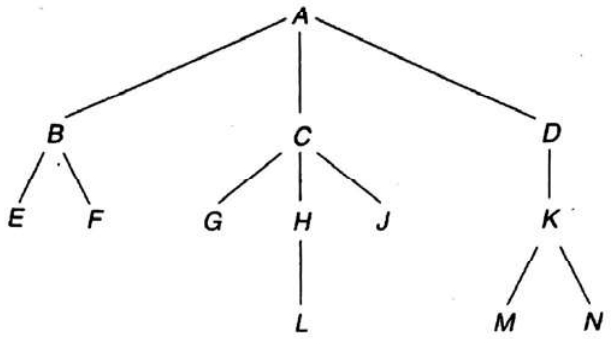


Fig. 7.68

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.

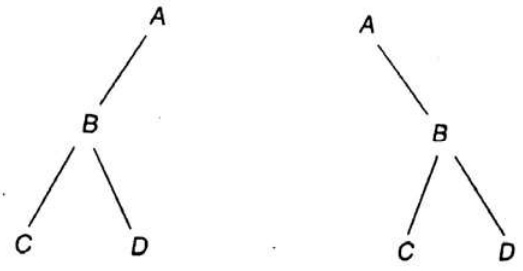
The last group of nodes, those with no children, are called *terminal nodes*.

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. 7.69. 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 *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.



(a) Tree T_1 (b) Tree T_2

Fig. 7.69

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, ..., 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 7.39

Consider the general tree T in Fig. 7.68. Suppose the data of the nodes of T are stored in an array INFO as in Fig. 7.70(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.

	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. 7.70

And so on. Figure 7.70(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.

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 7.40

Consider the general tree T in Fig. 7.68. The reader can verify that the binary tree T' in Fig. 7.71 corresponds to the general tree T. Observe that by rotating counterclockwise the picture of T' in Fig. 7.71 until the edges pointing to right children are horizontal,

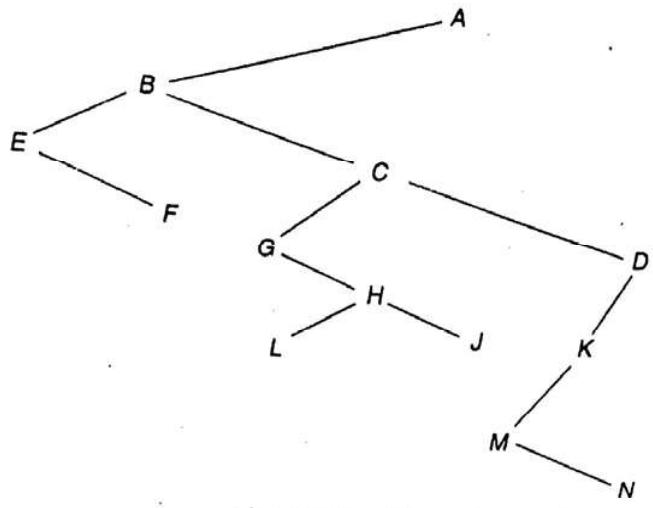


Fig. 7.71 *Binary Tree T′*

we obtain a picture in which the nodes occupy the same relative position as the nodes in Fig. 7.68.

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. 7.72. 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.

(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. 7.73(a). Repeating the above process with each new node, we finally obtain the required tree T in Fig. 7.73(b).