Two properties

Min heap

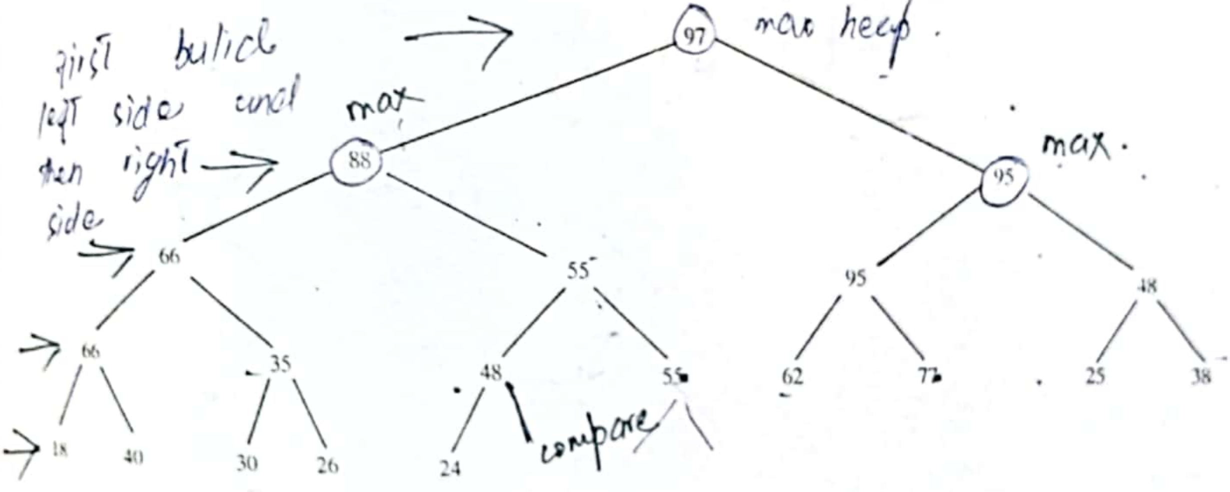
7.10 HEAP; HEAPSORT

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

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

EXAMPLE 7.19

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



(*a*) Heap.

– large from

TREE

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

(*b*) Sequential representation.

Fig. 7-29

Inserting into a Heap

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

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

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

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

EXAMPLE 7.20

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

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

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

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

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

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

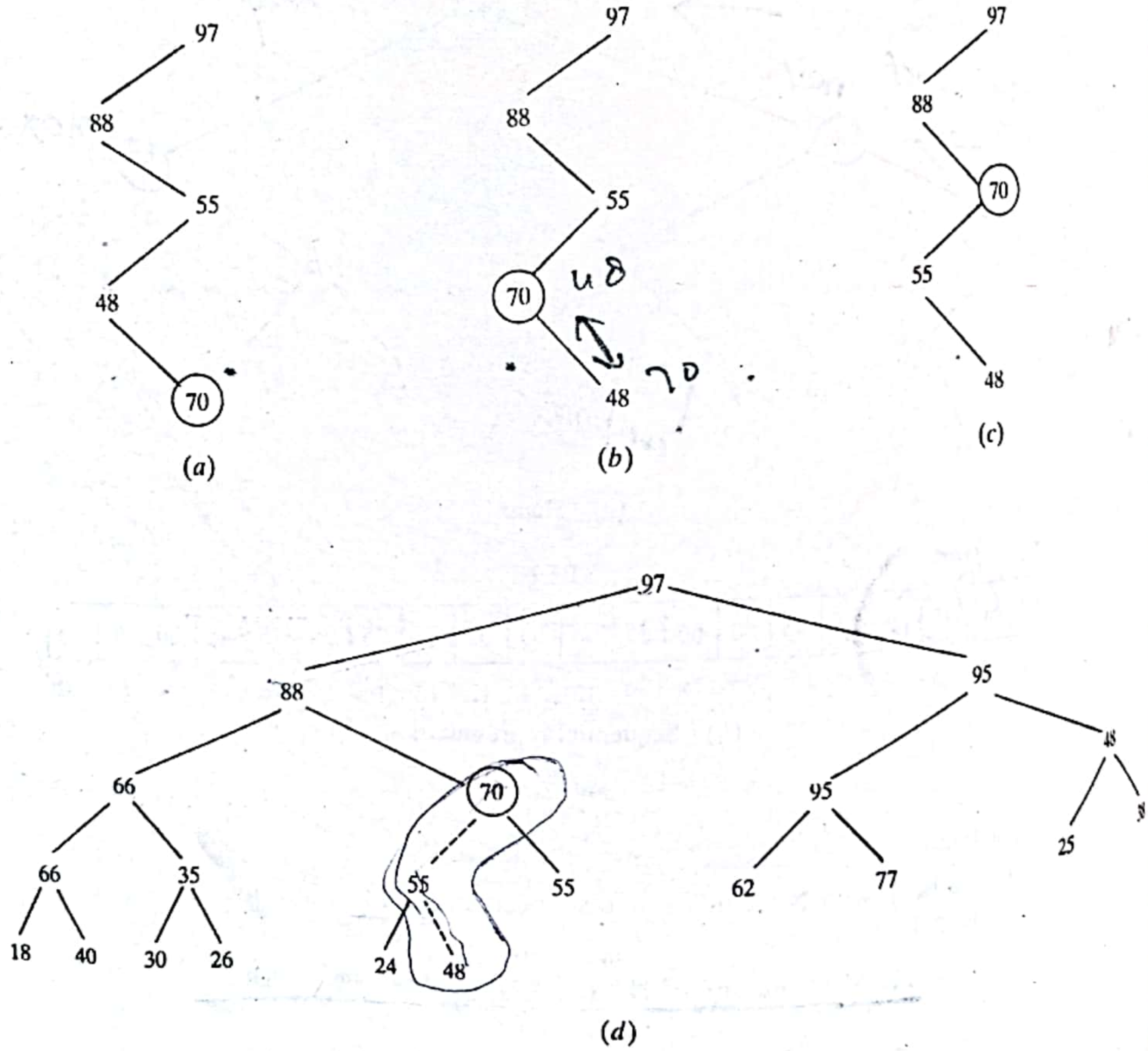


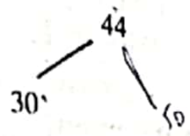
Fig. 7-30 ITEM = 70 is inserted.

EXAMPLE 7.21 Suppose we want to build a heap H from the following list of numbers:

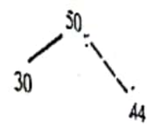
44, 30, 50, 22, 60, 55, 77, 55

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

(*a*) ITEM = 44.

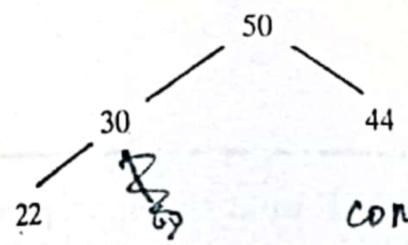


(*b*) ITEM = 30.



(*c*) ITEM = 50.

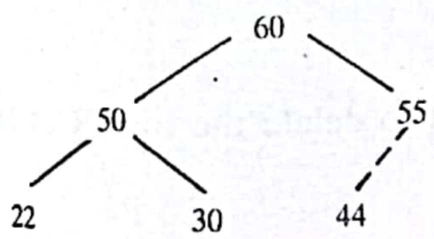
compare with
Parent node



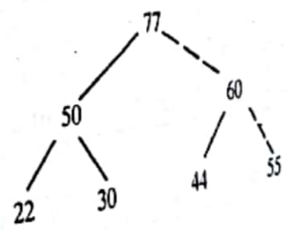
compare

(*d*) ITEM = 22.

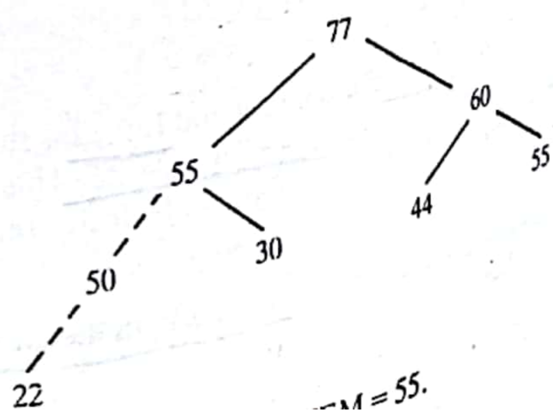
(*e*) ITEM = 60.



(*f*) ITEM = 55.



(*g*) ITEM = 77.



(*h*) ITEM = 55.

The formal statement of our insertion procedure follows:

Procedure 7.9: INSHEAP(TREE, N, ITEM)

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

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

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

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

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

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

Deleting the Root of a Heap

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

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

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

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

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

EXAMPLE 7.22

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

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

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

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

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

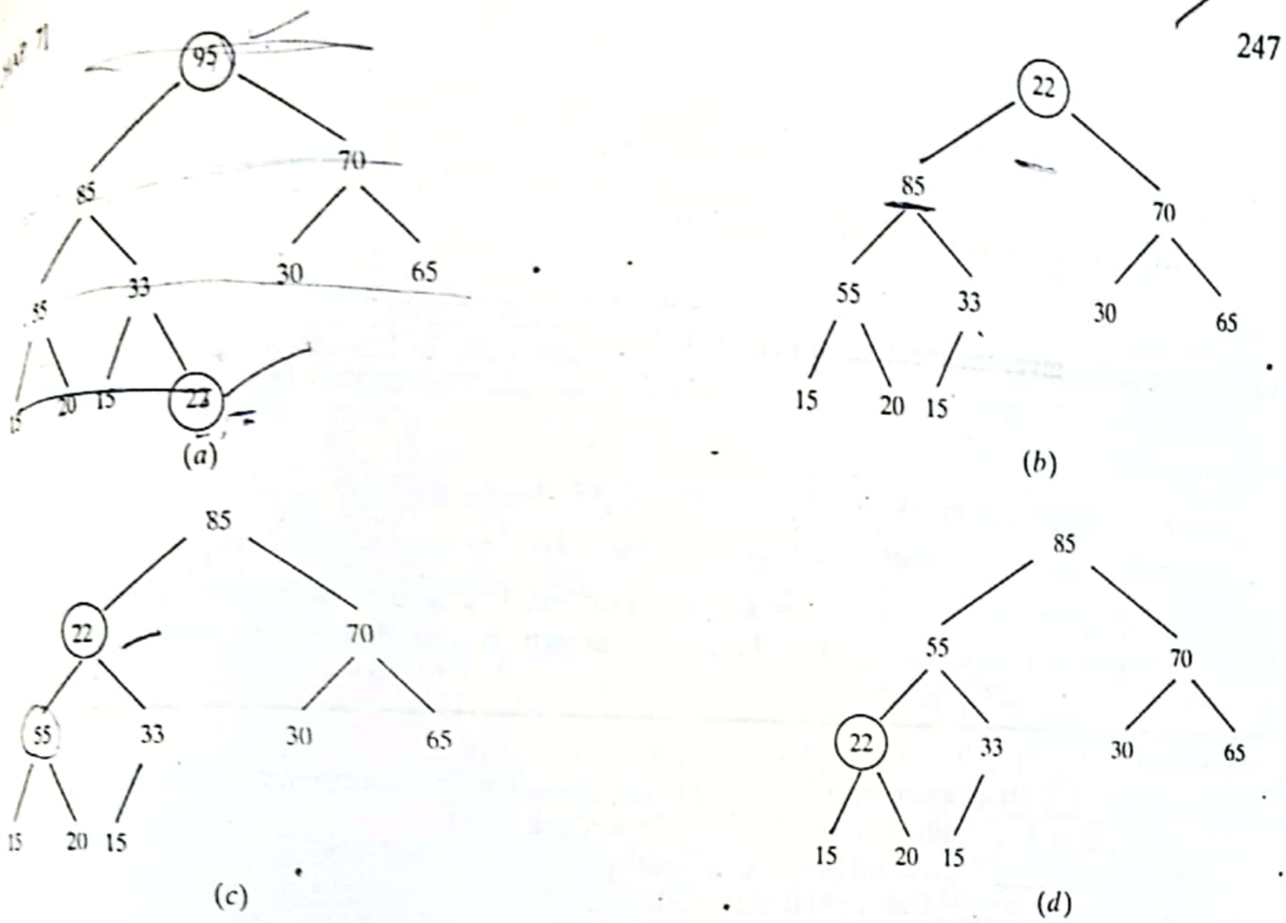


Fig. 7-32 Reheaping.

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

The formal statement of our procedure follows.

Procedure 7.10: DELHEAP(TREE, N, ITEM)

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

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

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

Application to Sorting

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

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

Phase B: Repeatedly delete the root element of H.

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

Algorithm 7.11: HEAPSORT(A, N)
An array A with N elements is given. This algorithm sorts the elements of A.

1. [Build a heap H, using Procedure 7.9.]
   Repeat for J = 1 to N − 1:
   Call INSHEAP(A, J, A[J + 1]).
   [End of loop.]
2. [Sort A by repeatedly deleting the root of H, using Procedure 7.10.]
   Repeat while N > 1:
   (*a*) Call DELHEAP(A, N, ITEM).
   (*b*) Set A[N + 1] := ITEM.
   [End of Loop.]
3. Exit.

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

Set B[N + 1] := ITEM

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

Complexity of Heapsort

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

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

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

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

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

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

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

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

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

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

Other sorting algorithms are investigated in Chap. 9.

7.11 PATH LENGTHS; HUFFMAN'S ALGORITHM

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

$$\boxed{N_E = N_I + 1}$$

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

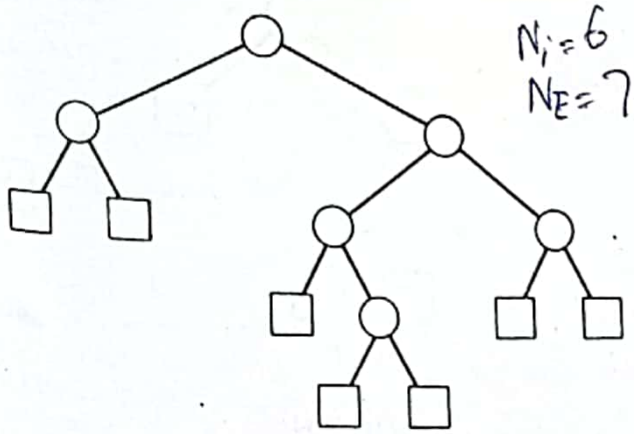


Fig. 7-33

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

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