```
            if (2*index+1 <= n)
                  Display(t, 2*index + 1);
            printf("%d\n", t[index].info);
            if (2*index+2 <= n)
                  Display(t, 2*index + 2);
      }
}
```

Preorder and postorder traversals of the sequential trees can be also be carried out by shifting `printf()` statement to the beginning and to the end respectively. Note that the outer `if` statement does not allow the unused locations to be printed.

7.10 PRIORITY QUEUE USING HEAP

A **Priority Queue** is a list of elements with an associated priority. Two types of priority queues - **Max priority** and **Min priority** queues were discussed in Section 5.6. It was said that the efficient implementation technique that could be used for realizing priority queues is **heap**. We develop functions to create a heap, insert an element in a max priority queue and finally delete the largest element from the max priority queue.

7.10.1 Heaps

Definition

A **max heap** is a tree (need not be a binary tree) that satisfies the following properties:

1. Every leaf node is of height *h* or *h* - 1.
2. Every leaf node of height *h* appears to the left of every leaf node of height *h*.
3. The value in each node is greater than or equal to those in its children (if any).

A **min heap** is same as max heap except that in condition 3 the value of each node should be less than or equal to those of its children. Figure 7.19(a) and (b) shows some max and min heaps respectively.

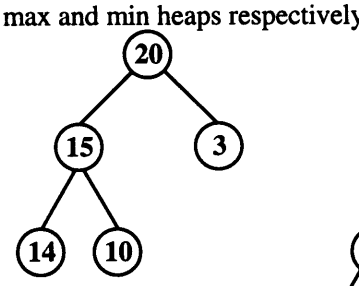


Fig. 7.19(a) Max Heap

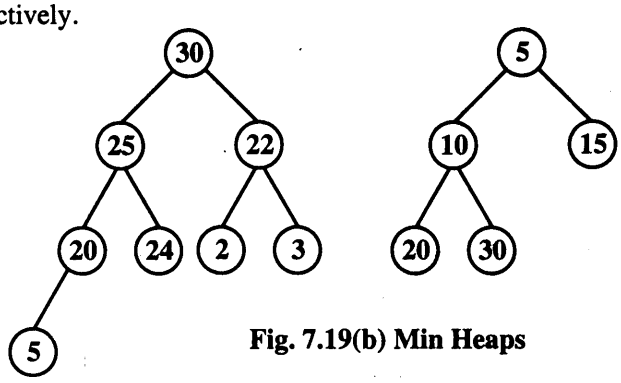


Fig. 7.19(b) Min Heaps

Since, the heap structure is organized in this way, it can be efficiently represented in a one-dimensional array. No locations in the array will have gaps and this means that all the elements in the heap can be stored successively. Also, accessing an *i*th node and its children can be done using 2**i* and 2**i* + 1 formulas. Once a heap is constructed, insertion and deletion in the heap can also be done more efficiently than an equivalent linked representation.

7.10.2 Initial Heap Construction

When we start with an array `a[1:n]` of `n` elements, it may not be in a heap format. Therefore, we construct the initial nonempty heap by performing `n` insertions into an initially empty heap. We shall show the steps in creating a max heap for an array `a[1:10]` = [15, 11, 30, 10, 8, 70, 20, 12, 5, 2]. This array may be shown as a binary tree in Figure 7.20(a).

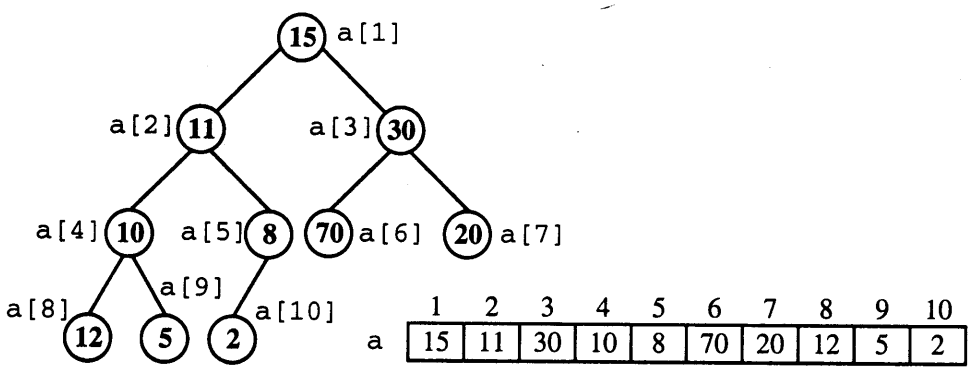
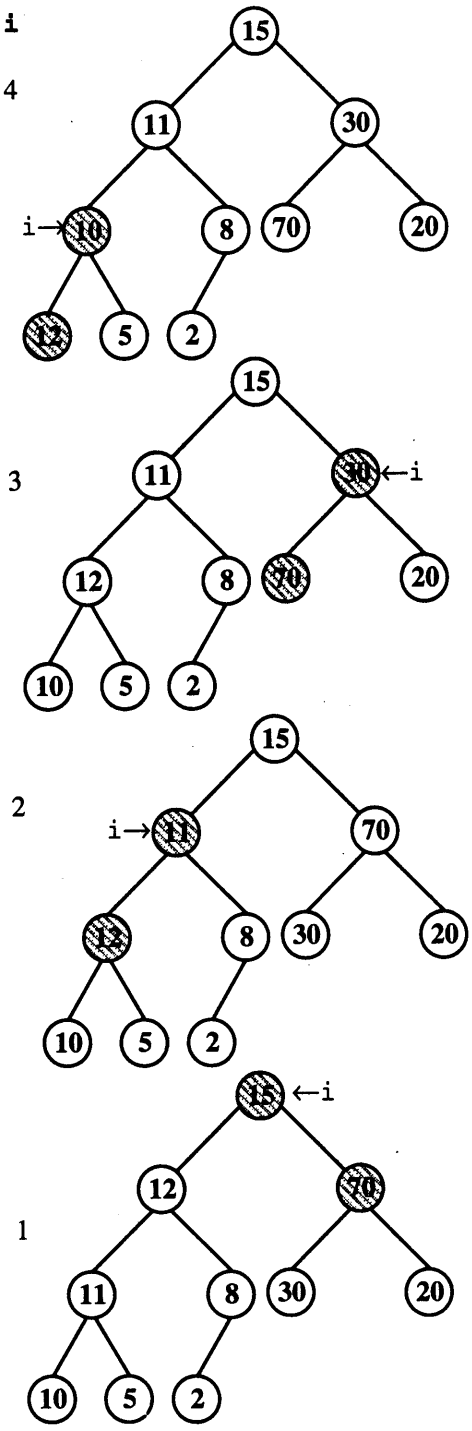


Fig. 7.20(a)

To convert a binary tree into max a heap, we begin with the middle of the array i.e., `i` = [n / 2] = 5. If the subtree is already a max heap, then we do not do anything. Suppose, if the subtree is not a heap then we adjust this subtree to make it a max heap by interchanging elements of root and its children. Next, we move to `i`-1, `i`-2, ...,1 and check at each iteration the subtrees and make them a max heap wherever necessary.

Let us apply this strategy to the tree in Figure 7.20(a). The first step with `i` = 5 does not require any adjustment as the root (8) is greater than its child at `a[10]` (2). The remaining steps are shown in Figure 7.20(b).



Comments

Element 10 at a[4] is to be interchanged with *Max*(12, 5). That is, a[i] ⇔ *Max*(a[2i], a[2i+1]).

Interchange element 30 with *Max*(70,20). a[3] ⇔ *Max*(a[6], a[7]).

Element 11 at a[2] is interchanged with *Max*(12,8). Element at a[4] is already in a heap and so no need for further adjustment.

a[1] ⇔ *Max*(a[2], a[3]), 15 and 70 are interchanged. Then, a[3] ⇔ Max[a[6], a[7] 30 and 15 are interchanged.

Figure 7.21(a) shows a max heap to which we wish to insert an element 1. It could be inserted to the left of element 3 and the resulting tree is still a max heap (see Figure 7.21(b)). Suppose, if the element value is 10 as shown in Figure 7.21(c), we need to move node whose value is 3 to make it as a max heap. Now the parent element is larger than its child node. Next, if an element of value 30 is to be inserted into the heap, then it has to occupy the root itself (see Figure 7.21(d)). Element 3 is moved to the left child position (just like Figure 7.21(c)) and 30 can not be its parent, because 30 is greater than its parent i.e., 25. Hence, node 25 is moved to the right child and 30 is placed in the root.

Implementation

We start from `i = CurrentSize + 1` position to see whether the new element, `x` could be inserted. To check this, `x` is compared with its parent i.e., `i/2` (note that `2*i` and `2*i + 1` are the addresses of left child and right child and similarly `i/2` is the address of the parent of `i`th node). If `x` is less, then it is immediately inserted as shown in Figure 7.21(a).

Otherwise, we use a loop to move up and try to get a position for `x` and so on until the root node is reached (until `i` is equal to 1). That is,

```
while (i != 1 && x > heap[i/2])
{
        move up by shifting the current element down.
}
```

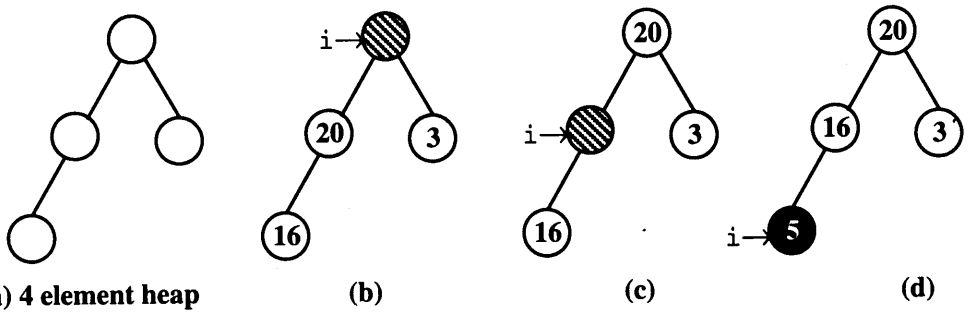
The above logic is incorporated in `MaxInsert()` function and is shown Program 7.13.

Program 7.13
Inserting into a Max Heap

```
void MaxInsert (int heap[], int *CurrentSize, int x)
{
     int c, e, i;
     if (*CurrentSize == MAX)
     {
          printf("Heap Full\n");
          return;
     }
     i = ++(*CurrentSize); /* last element */
     while (i != 1 && x > heap[i/2])
     {
          heap[i] = heap[i/2];
          i /= 2;
     }
     heap[i] = x;
}
```

7.10.4 Max Delete - Deletion from a Max Heap

In this section we shall show the deletion of the maximum element from the heap, which is exactly what we require for a descending priority queue. We know that in a max heap, the largest element will be at the root. But, when we delete this root element, the resulting structure will not be a max heap! Take a look at Figure 7.21(a) and assume that we wish to delete 25. Now after this deletion, the heap will have only 4 elements. A 4 element heap should look like the one shown in Figure 7.22.



(a) 4 element heap (b) (c) (d)

Fig. 7.22 Deletion of max and readjusting

To get this structure, simply check whether the last element i.e., 5 could be moved to the root (Figure 7.22(b)). However, the subtree with root node 5 is not a max heap, so move the larger of the two child nodes to the root. Element 20 is moved to the root as shown in Figure 7.22(c). Again, the subtree with root 5 is not a max heap, so move 5 down so that it occupies position 4 (see Figure 7.22(d)). Now, this is the final max heap structure.

The function `MaxDelete()` (see Program 7.14) the heap array and the number of elements (or size) in heap and `CurrentSize`. The parameter `CurrentSize` is of reference type, because when an element is deleted from the heap, the size of the array should be reduced by one and this should be reflected to the calling program.

Program 7.14
Delete max element from a Heap

```
int MaxDelete (int heap[], int *CurrentSize)
{
     int x, c, e, i;
     if (*CurrentSize == 0) return -1;

     x = heap[1]; /* save the maximum element */
     e = heap[*CurrentSize]; /* get the last element */
     (*CurrentSize)--;
```

```
    /* Heap the structure again */
    i = 1; c = 2;
    while (c <= *CurrentSize)
          {
               if (c < *CurrentSize
                         && heap[c] < heap[c+1])
                    c++; /* pick larger of children */
               if (e >= heap[c])
                    break;/* subtree in max heap */


               /* move the child to root */
               heap[i] = heap[c];
               i = c;
               c *= 2; /* go to the next level */
          }
    heap[i] = e;
    return x;
}
```

The main `while` loop is executed until the appropriate position for the last element `e`. The child pointer `c` will be moved to each level in the tree depending upon the value of the last element and the remaining elements in the heap. In the example tree shown in Figure 7.22, `i` points to the root node and `c` to the left child of `i` and `c + 1` to the right child of `i`. Since, `e` cannot occupy the root, 20 takes the root position. Pointer `i` next goes to `c`'s position and `c` is advanced to the left child of `i` (position 2). Again `e` is less than `heap[c]`, 16 is moved to the parent node i.e., position 2 or `i`th position. The `while` loop terminates and element 5 is inserted at `i`th position (4). The various positions of `i` is shown with an arrow in Figures 7.22(b), (c) and (d).

7.11 THREADED BINARY TREES

Threaded binary trees are binary trees with unnecessary NULLs absent in nodes. Instead of NULLs, there are special links provided in these nodes. These links are called **threads**. Now we shall discover the need for such links.

Recall the inorder traversal operation in a binary tree that used recursive technique. Even if you try to implement using iterative method, you would need a stack to store the addresses of the nodes that are pushed in a stack until a NULL is encountered. While returning, these addresses are to be popped from the stack so that the remaining subtree can be explored. Stacking and un-stacking consumes lot of time in the program execution. In other words, we must totally eliminate the stack for the traversal. Yes, it is possible to do so by making a left subtree pointer pointing directly to its inorder

successor. This link actually takes the control directly to the node last visited during the traversal; otherwise this would be obtained from the stack as explained earlier.

Let us consider an example tree and see how the threads help us traversing with out remembering the node addresses (see Figure 7.23). In the inorder traversal, the left subtree is traversed (`t->left`) until a NULL is reached.

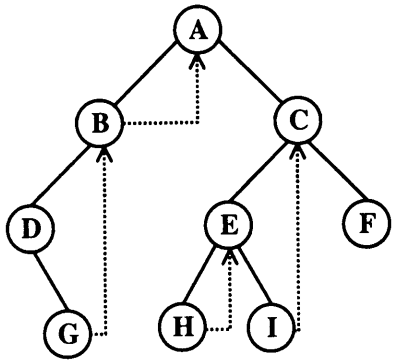


Fig.7.23 Right in-threaded binary tree

Note that only the right links of one of the nodes will have threads pointing to their inorder successor (shown in dotted lines). The left links will have still NULLs. The last node's right link cannot have a thread because it does not have any inorder successor. Let us come back to our original problem of stacking the return addresses.

When we reach the left most subtree with root node D the `Inorder()` function would print D and then G. Next, the control should go back to the subtree in the previous level, B. Now, instead of popping from the stack, the control goes via the thread from G to B (G → B). Similarly, after printing B, using the thread B → A it points to A, and so on. We can show the inorder traversal of the tree of Figure 7.23 as,

These types of trees in which the right most node still have a NULL right link, are called as **right in-threaded binary trees.**

To implement a binary tree with threads, modification to our tree definition is required and is as follows:

```
struct Thread
{
      int info;
      struct Thread *left;
      struct Thread *right;
      int rt;     /* right thread */
};
typedef struct Thread *THNODE;   /* thread node */
```

The significance of the extra field `rt` is to know whether a thread link exists or not. If the value of `rt` is 1, then right link is NULL and when `rt` is 0 it has a thread. For the tree of Figure 7.23, the values of `rt` field for the various nodes are:

`rt` = 1 for the nodes : B, F ,G, H, I
`rt` = 0 for the nodes: A, C, D, E

When `rt` = 1, we can traverse through the thread. Program 7.15 shows the C code for an inorder traversal for a right in-threaded binary tree.

Program 7.15
Traversal of a right-in threaded binary tree

```
Void ThreadInorder (THNODE t)
{
     THNODE q;
     do
     {
          q = NULL;
          while (t)  /* traverst LST */
          {
               q = t;
               t = t->left;
          }
          if (!q)
          {
               printf("%d ", q->info);
               t = q->right;
               while (q->rt && t)
               {
                    printf("%d ", t->info);
                    q = t;
                    t = t->right;
               }
          }
     } while (q);
}
```

7.12 SUMMARY

- **Trees** are ideal data structure for representing hierarchical structures, fast searching, priority queues, etc.,
- We have studied two major types of trees **general trees** or (simply trees) and **binary trees.**
- A **binary tree T** is a finite set of nodes one of with the following properties:
  (1) Either the set is empty, T = 0, or
  (2) The set consists of a root *r*, and exactly two distinct binary trees called ***left subtree*** (T_L) and a **right subtree** (T_R).
- The children of the same parent are called as **sublings.**
- Every node that is reachable from a node, say V, is called a **descendent** of V. If a descendent node appears on the left subtree, then it is called a **left-descendent,** otherwise it is called as **right-descendent.**
- In a binary tree or a tree, a node with no children is called as a **leaf node.**
- By definition, the root is at the highest level and its children nodes are at one level more than the root. In general, the **level** of any node is one more than its father.
- The **depth** of a binary tree is the longest path from the root to any leaf node.
- The **degree** of a node is the number of children it has.
- There are three traversal algorithms: (1) Preorder (2) Inorder (3) Postorder.
- A **binary search tree (BST)** or **ordered tree** is a binary tree that may be empty. A nonempty BST satisfies the following properties:
  1) Every element has a key or elemental value and no two elements have the same key. That is, all keys are distinct.
  2) The keys (if any) in the left subtree of the root are *smaller* than the key in the root.
  3) The keys (if any) in the right subtree of the root are *larger* than the key in the root.
  4) The left and right subtrees of the root are also BSTs.
- The various operations of binary trees are:
  building a tree, adding a node, deleting a node, counting the number of nodes, counting the number of leaves, copying a tree, etc.
- A binary tree can be built using arrays and is called as an **implicit** representation.
- A **max heap** is a tree (need not be a binary tree) that satisfies the following properties:
  1. Every leaf node is of height *h* or *h* - 1.
  2. Every leaf node of height *h* appears to the left of every leaf node of height *h*.
  3. The value in each node is greater than or equal to those in its children (if any).
- Implementation of a priority queue can be done efficiently using a heap. Two major functions were discussed: Insertion into a Max Heap and deletion from a Max Heap.

- A threaded tree is same as a binary tree with extra fields called threads. These threads will guide faster searching in a BST and save unnecessary stack space during recursive traversal.

7.13 EXERCISES

7.1 Define the following (write appropriate figures):
(a) General Tree
(b) Binary Tree
(c) Strictly binary tree
(d) Complete binary tree
(e) Binary Search Tree
(f) Almost complete binary tree
(g) Threaded binary tree

7.2 Write C functions to convert a (1) BST into a singly linked list and (2) Singly linked list to a BST.

7.3 Construct a BST for the following data set:
[23, 34, 11, 5, 78, 3, 90, 4, 20]
Write the preorder, *inorder* and *postorder* traversal for the tree just constructed.

7.4 Write iterative routines for *iorder*, preorder and *postorder* traversals.

7.5 Given the following traversals, construct the tree:
(1) *Inorder*: 4, 7, 2, 1, 5, 3, 6 *Preorder*: 1, 2, 4, 7, 3, 5, 6
(2) *Inorder*: a + b * c / d – e + f *Preorder*: + / * + a b c – d e f

7.6 Write necessary functions in C to construct a binary tree (not a BST) such that we could add a node either to the left subtree or right subtree.

7.7 Design a function to display the number of non-leaf nodes in a binary tree.

7.8 Write an algorithm to convert a general tree to a binary tree.

7.9 From the problem 7.3, trace the deletion algorithm for deleting the key, 4.

7.10 What are (1) Heaps (2) Max Heap (3) Min Heaps. Discuss its applications.

7.11 For the data set shown below, show how a Min Heap can be constructed.
[45, 23, 11, 10, 48, 12, 25, 74, 17]

7.12 What do you understand by threaded binary trees? Explain with an example.

7.13 Show the right-in threading for the tree of problem 7.12.

7.14 The text only shows the traversal of a threaded tree. Develop C functions to build right-in threaded trees.