Step 7: Stop

Let us consider another example, Example – 2 and apply the pseudo code just given.

Example – 2

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

According to Step 1, A becomes the root. Next element B is added to the left of current root A, because of B appearing on left of A in the inorder sequence (see Figure 7.11(c)). The element C should be on right of B, then D should be left of A and right of B and also C.

You must notice here that, when an element is taken for insertion, it should be checked with all the previous elements added already. Continuing the same way, the next element is E and can be added to the left of D for the reason just explained.

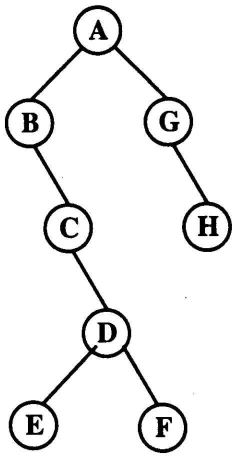


Fig. 7.11(c) Binary tree for the given inorder and preorder

Adding F to the tree is crucial. It may look that, this element appears to the left of A and right of E just added. So we should see just the previous node and decide on the insertion. Instead we must strat from the root, and proceed until we reach the last node added in the tree.

We shall see the placement of F in the inorder carefully with the partial tree constructed. Node F appears on the left of A, then right B, right of C, and finally right of D. Hence, F goes to the right of D and **not to the right of E.**

Nodes G and H are very easy to add because they appear to the right of A in the inorder sequence.

7.7 BINARY TREE OPERATIONS AND ITS ADT

Since the abstract data type definition is to be independent of any implementation, we shall present the same in Figure 7.11(d).

```
ADT BinaryTree
{
    Instances:
          Collection of nodes stored in a linked
          arrangement.
    Operations:
          MakeTree(x): Build a binary tree by inserting
          element x in an ordered fashion (i.e., Binary
          search Tree).
          Delete(x): Delete the node with data x.
          PreOrder(): Traverse the tree in Preorder.
          PostOrder(): Traverse the tree in Postorder.
          InOrder(): Traverse the tree in Inorder.
          Search(x): Search for a given data x, return 1
          if found else return 0.
}
```

Fig. 7.11(d) ADT of a binary Tree

In addition to the operations specified in the ADT, there are additional operations that are there for a binary tree as listed below.

- Determine the height.
- Determine the total number of nodes.
- Get the copy.
- Determine whether two binary trees are identical.
- Delete the entire tree.
- Evaluate an expression tree.

Search for a key node in a binary tree. If it is found return true, else insert the data in the tree.

7.8 IMPLEMENTATION OF BINARY TREES

This section covers the various implementation issues of ADT operations discussed in Section 7.7 in C language. It is easier to build a binary tree if we assume that we are constructing a Binary Search Tree rather than an ordinary binary tree. So for the rest of the topics we shall assume search trees as the primary example.

7.8.1 Binary Search Tree (BST)

Definition

A **binary search tree (BST)** 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.

BST is also called as an **ordered tree**. Figure 7.12 shows sample BSTs and non BSTs. In Figure 7.12(a) all the nodes

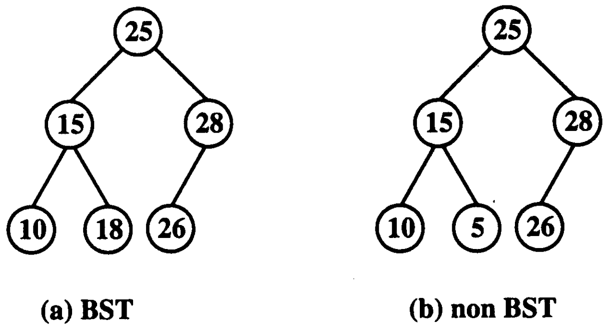


Fig. 7.12 Sample Binary Search Trees and non-search trees

to the left of 25 (root) are less and all nodes on its right are greater. This is true for every subtree also. The tree shown in Figure 7.12(b) is not a BST as node 5 appears on the right of node 15.

7.8.2 Building a BST

We can build a binary search tree starting with an empty tree. A new node is added to the BST first by obtaining the address of the node. Then it is added to the left or right depending upon the value of the new element. If the new elemental value is less, then it is added to the right, otherwise it is added to the right.

Take for instance the binary search tree shown in Figure 7.12(a) and assume that we wish to add a new element with value 20. Starting from the root (node 25), we search for the address of the node for insertion. This is done by systematically traversing the BST recursively either to the left of the root or to the right. If the new elemental value is less than the root of the tree (or any subtree) then traverse to the left, else traverse through right. In our example, we traverse to left subtree and reach node 15 and traverse further to reach node 18 and stop. Since right of node 18 is NULL no more iteration is initiated.

Now the new node 20 is inserted to the right of 18. In this problem, we do not carry any specific value to the previous calling sequence or even to the calling function, unlike the case of the factorial function. The reader is advised to go through Chapter 4 again for a better understanding of trees.

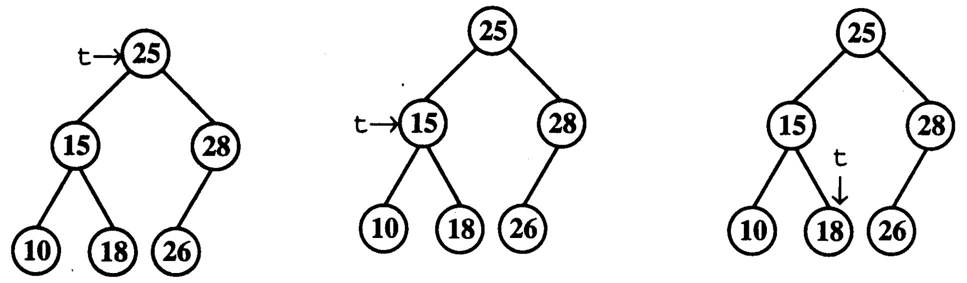
The Program 7.1 shows the C code for building a BST based upon the logic just explained.

Program 7.1
Building a binary search tree

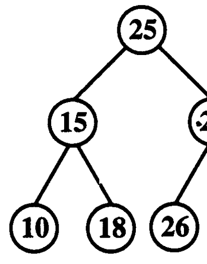
```
TNODE MakeTree (TNODE t, int x)
{
     if ( t == NULL)
     {
           t = talloc();
           t->info = x;
           t->left = t->right = NULL;
     }
     else if (x < t->info)
                 t->left = MakeTree(t->left, x);
           else
                 t->right = MakeTree(t->right, x);
     return t;
}

TNODE talloc(void)
{
     return (TNODE) malloc (sizeof(struct Tree));
}
```

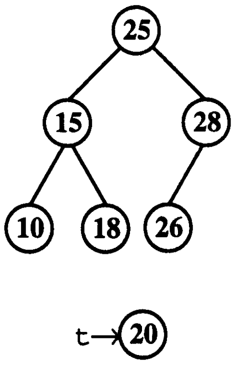
The function `MakeTree(TNODE t, int x)` is invoked recursively until the address of the node after which the new node x to be inserted. Creating the first node or the root node is straight forward - taken care by the first `if` statement. Figure 7.13 shows the steps required to obtain the address of the node for insertion considering the example tree shown in Figure 7.12(a).



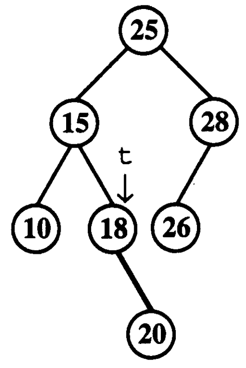
(a) Initial tree **(b) Traverse to Left subtree** **(c) Traverse to Right subtree**



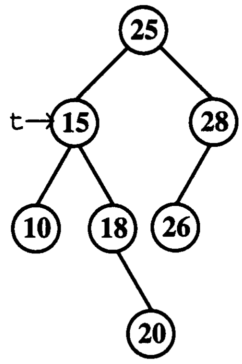
(d) t = NULL



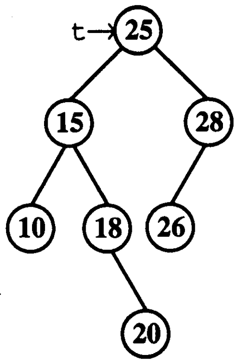
(e) New node 20 is created



(f) Control back to node 18



(g) Next the control is back to 15



(h) The new tree is returned

Fig. 7.13 Insert node 20

The Figure 7.13(a) to (d) are recursive calls to `MakeTree ( )` function and (f) to (h) are return paths. Once the variable `t` reaches the root node, the address of the new tree (with new node added) is returned to the calling program.

7.8.3 Tree Traversal

Writing C programs for preorder, inorder and postorder traversals may be done based on the algorithms shown in Figure 7.9. All the three traversals are shown as separate functions in Program 7.2 that receives the address of the root node `t`.

Each function systematically traverses through the tree both left and right and displays the elemental value. As explained already, the only difference that exists among these functions is the time at which the `printf` is executed. For preorder, it happens first, in inorder it occurs after the left subtree traversal and in postorder it comes after traversing both the left and right subtree.

7.8.6 Size

The objective of the function `Size()` is to find the number of nodes in a binary tree. We can use any of the three traversal methods to find the number of nodes, as the traversal method visits each node exactly once. The function `Size()` is shown in Program 7.5.

Program 7.5
Number of nodes in a binary tree

```
void Size (TNODE t, int *count)
{
     if (t)
     {
           (*count)++;
           Size(t->left, count);
           Size(t->right, count);
     }
}
```

The program uses the preorder traversal with a reference parameter `count`. `Size()` is to be invoked as `Size(root, &cnt)`, where `root` is the root node address (pointer to the tree) and `cnt` is initialized to zero which receives the number of nodes. Instead of `printf()` statement in `Preorder()`, `(*count)++;` is written so that when a node is visited the counter is incremented and at the end it would have the number of nodes that have been visited.

7.8.7 Leaf Nodes

A leaf node is one that does not have both left and right subtrees. To obtain the count of the number of leaf nodes, we simply traverse the tree until we reach a node which has both its children absent. Then increment the count (see Program7.6) and continue the traversal.

Program 7.6
Number of leaf nodes

```
void LeafNodes (TNODE t, int *count)
{
     if (t)
     {                                      /* is leaf node? */
          if (!t->left && !t->right) ++(*count); /* yes */
```

```
            else {
                        LeafNodes(t->left, count);
                        LeafNodes(t->right, count);
            }
        }
    }
```

Again we use the same method adopted in program 7.5. The inner `if` statement checks for a leaf node and when it is so, count is incremented, otherwise traverse the tree recursively left and as well as right side.

7.8.8 Find the Min element

Finding a minimum (or maximum) element in a BST can be done by systematically traversing to the left subtrees until the leaf node is reached. Take a closer look at the tree shown in Figure 7.14 and you notice that the minimum element in the whole tree is located at the left most subtree. This will be the case for all search trees.

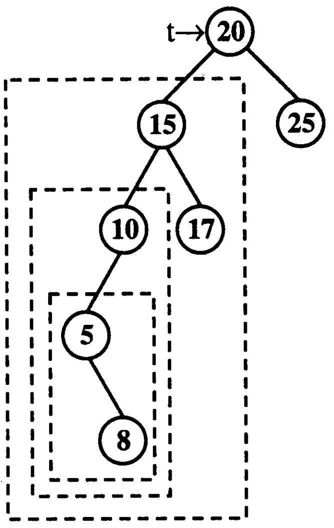


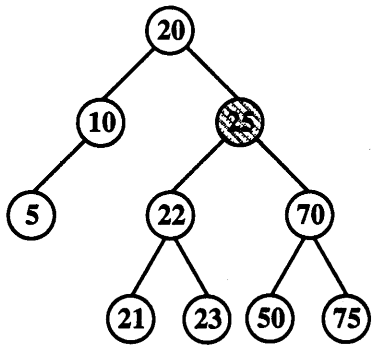
Fig. 7.14 A BST with 7 nodes

The left subtree of the root node 20 (shown in broken line boxes) is not empty and hence we proceed to subtree containing 15 as the root and again its left subtree is not empty and so we move to the subtree whose root is 10, and so on. When we reach node 5 whose left link is `NULL` we are at the minimum element. The C code for finding the minimum element `FindMin()` appear in Program 7.7.

The `if-else` clause return the `info` field of the left most node when its left link is `NULL`, else `FindMin()` is called recursively to reach the left most node using

Case 1

To delete node 21 of Figure 7.15(a) which is a leaf node, we simply set the left link of its parent (node 22) to NULL. Then, deallocate the memory for node 21.



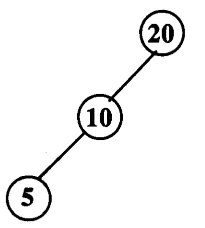
(c) Largest node i.e. 25 in the left subtree takes the position of p
Fig. 7.15 (Contd.)

Suppose, the node to be deleted is 23, then set the right link of node 22 to NULL and discard node 23.

Case 2

In this case, the element to be deleted is p that has only one nonempty subtree; may be left or right. Here, there are again two cases that may arise:

1) The node p may have a parent.
2) The node p may not have parent.



For the second case (see the Figure on right), the node to be deleted is the root itself. Considering the above figure, assume that node 20 is to be deleted. In this case, we simply discard the root itself and make the child node (left or right) 10 as the new root.

For the first case, in Figure 7.15(a) consider the node 10 is to be deleted. We make the left link of its parent to point to node 5. That is,

```
parent->left = p->left;
```

where, p points to the node to be deleted and parent points to its parent node.

Case 3

To delete a node that has two non-empty subtrees, we have two options:

1. To replace p with the largest element in the left subtree, or
2. To replace p with the smallest element in the right subtree.

In either option, the resultant tree will be a binary search tree.

For example, in the tree of Figure 7.15(a) assumes that we wish to delete node 30. **We can replace this node with 25 - which is the largest in the left subtree or with 50 - which is the smallest in the right subtree.** By following the first option, the resultant tree appears in Figure 7.15(c) and by following the second option the tree looks like the Figure 7.15(b).

In designing the program for deletion, we can find the minimum element in the right subtree by traversing the left subtree until NULL is reached (see section 7.8.8), similarly, the largest element in the left subtree is obtained by traversing the right subtree until a NULL is reached. The program for deleting a node in a binary search tree appears in Program 7.9.

Program 7.9
Deleting a node

```
TNODE Delete (TNODE root, int k)
{
    TNODE p = root,       /* pointer to root */
          parentp = NULL; /* parent of p */
    TNODE s, parents; /* largest in LST & its parent */
    TNODE c;
    int e;                /* for replacement */

    /* search for the key */
    while (p && p->info != k)
    {
         parentp = p;
         if (k < p->info) p = p->left;
         else p = p->right;
    }
    if (!p)
    {
         printf("Node not found\n");
         return root;
    }
    e = p->info;

    /* handle case - 3, p has both the children */
    if (p->left && p->right)
    {
         /* move to the larger in left subtree */
         s = p->left;
         parents = p;
         while (s->right)
         {
```

The solid lines indicate the actual links, whereas the broken lines indicate missing nodes in the complete binary tree. Each element of the tree occupies an array location and the numbers written adjacent to the nodes corresponds to the array locations. The locations 2, 4, 5 and 7 are vacant is Figure 7.17(a) and similarly locations 5 and 7 are vacant in Figure 7.17(b).

In this representation, if more elements are missed out, lot of memory locations will be wasted. For instance, if the tree is skewed on either left side or right side then lot of memory will be wasted (see Figure 7.18). In fact, right skewed binary trees are more wasteful.

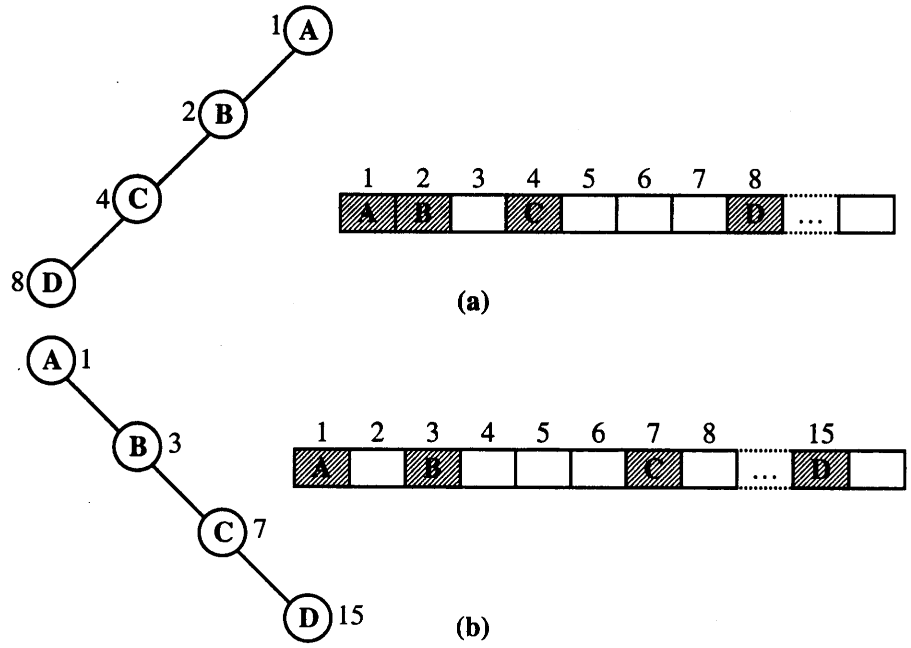


Fig. 7.18 Left and right skewed trees

Property 1

The maximum array size required to store an n-element tree is = 2^n - 1.

Property 2

The root node will always be at location = 1.
The left son of ith node will be at location = $2*i$
The right son of ith node will be at location = $2*i + 1$
For example, the right son of node C in Figure 7.18(b) is at location 15. That is, 2 * 7 + 1 (node C is at 7th location).

Property 3

If C arrays are used, the first location is addressed as 0. Hence, the addresses for left and right son may differ with respect to Property-2.

The root may be stored at location	$= 0$
The left son of *i*th node will be at location	$= 2*i + 1$
The right son of *i*th node will be at location	$= 2*i + 2$

For implementation in C language, we follow Property-3.

Implementation

Only few locations in the array contain data and the other locations with junk values, if we want to store an arbitrary binary tree in an array. Therefore, we shall use an additional field called **used** along with the actual data. This approach is shown as follows:

```
#define n 100
struct STree
{
     int info;
     int used;
};
typedef struct STree SNODE[n];
```

Initially all the locations of the used field will be filled with zeros, indicating that all are unused. When a data is stored in a particular field its used field will be assigned 1. Using this approach, when we traverse the tree (array) the locations whose used field is 1s only need to be displayed.

7.9.1 MakeTree

To construct a binary search tree with the implementation strategy explained above, we follow the same method as that of a linked implementation. This means that we store the key in the search tree such that the tree remains ordered. To insert the new element we traverse to the left subtree using 2*i + 1 and right subtree using 2*i + 2 recursively. The root node is stored at location 0th location and the subsequent elements are stored by comparing with the key and traverse left or right subtree until an unused location is found. The function MakeTree() is shown in Program 7.10.

Program 7.10
Building a implicit binary search tree

```
int MakeTree (SNODE t, int index, int x)
{  /* inserts x in to the tree and returns 1, */
   /* otherwise return 0                      */

    if (index >= n)
         return 0;

    /* position found, so insert x */
```

```
        if (!t[index].used)
        {
              t[index].info = x;
              t[index].used = 1;
              return 1;
        }

        if (x < t[index].info)
              return (MakeTree(t, 2*index+1, x));
        else
              return (MakeTree(t, 2*index+2, x));
   }
```

The function is invoked as `MakeTree(t,0,key)` where `t` is the array of structures of type `SNODE`, `index` value is set to 0 for the root node creation and key is the element to be inserted in the tree. Since, the array has an upper bound, *n* when the index is out of this bound no function returns 0 indicating the array is *full.*

Element `x` is inserted once an appropriate location is found; this means that the field must be 0 indicating that it is not filled already. Recursive calls to `MakeTree()` are initiated upon the value of the element to be inserted. The last `if` statement takes care of this.

7.9.2 Display

The `Display()` function uses almost a similar approach as that of the `Inorder()` function of linked implementation.

Traversing on left subtree is done by using,

```
          Display(t, 2*index + 1);
```

and the right subtree is by the statement,

```
          Display(t, 2*index + 2);
```

Terminating conditions for both of these must be that any `index` value calculated must be with in the array bound. (<= n). Program 7.11 shows the resulting code.

Program 7.11
Display the tree

```
void Display (SNODE t, int index)
{
    /* don't display unused locations */
    if (t[index].used)
    {
```