Binary Search Trees

This section discusses a special type of binary tree, called a binary search tree.

Consider the binary tree in Figure 11-6.

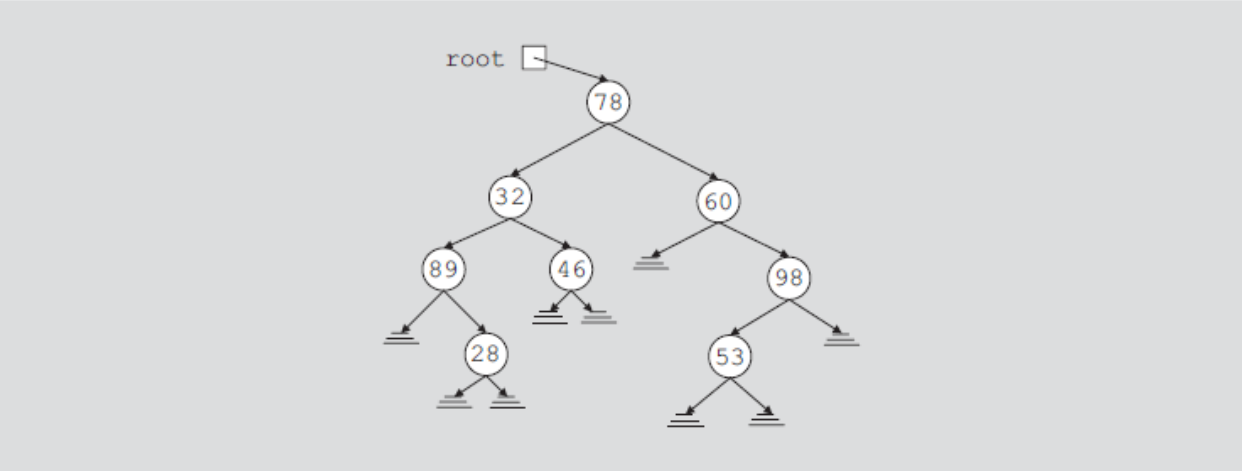

FIGURE 11-6 Arbitrary binary tree

Suppose that we want to determine whether 50 is in the binary tree. To do so, we can use any of the previous algorithms (in order, pre order, post order) to visit each node and compare the search item with the data stored in the node. However, this could require us to traverse a large part of the binary tree, so the search would be slow. We need to visit each node in the binary tree until either the item is found or we have traversed the entire binary tree because no criteria exist to guide our search. This case is like an arbitrary linked list where we must start our search at the first node, and continue looking at each node until either the item is found or the entire list is searched.

On the other hand, consider the binary tree in Figure 11-7.

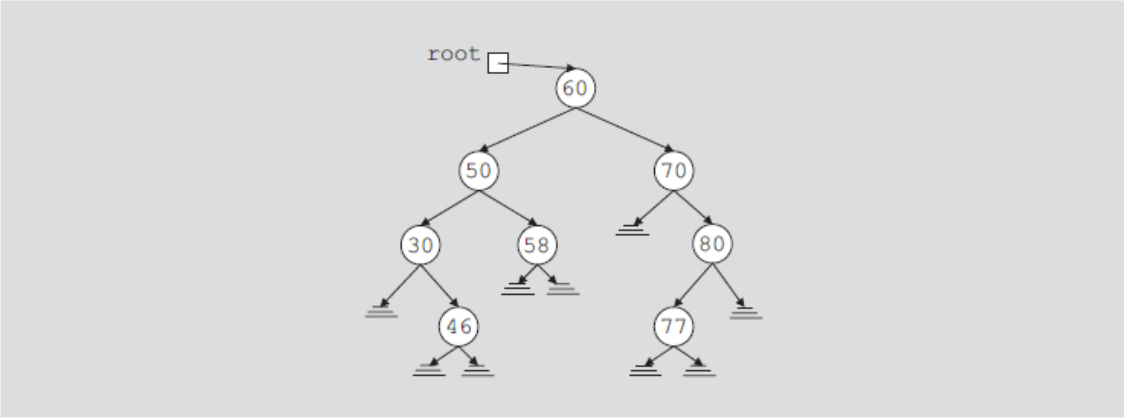

FIGURE 11-7 Binary search tree

In the binary tree in Figure 11-7, the data in each node is

- Larger than the data in its left sub tree
- Smaller than the data in its right sub tree

The binary tree in Figure 11-7 has some structure. Suppose that we want to determine whether 58 is in this binary tree. As before, we must start our search at the root node. We compare 58 with the data in the root node; that is, we compare 58 with 60. Because $58 \neq 60$ and $58 < 60$, it is guaranteed that 58 will not be in the right sub tree of the root node. Therefore, if 58 is in the binary tree, it must be in the left sub tree of the root node. We follow the left pointer of the root node and go to the node with info 50.We now apply the same criteria at this node. Because $58 > 50$, we must follow the right pointer of this node and go to the node with info 58. At this node we find item 58. This example shows that every time we move down to a child, we eliminate one of the sub trees of the node from our search. If the binary tree is nicely constructed, the search is very similar to the binary search on arrays.

The binary tree given in Figure 11-7 is a special type of binary tree, called a binary search tree. (In the following definition, by the term key of the node we mean the key of the data item that uniquely identifies the item.)

Definition: A binary search tree, T, is either empty or the following is true:

i. T has a special node called the **root** node.

ii. T has two sets of nodes, L_T and R_T, called the left sub tree and right sub tree of T, respectively.

iii. The key in the root node is larger than every key in the left sub tree and smaller than every key in the right sub tree.

iv. L_T and R_T are binary search trees.

The following operations are typically performed on a binary search tree:

- Search the binary search tree for a particular item.
- Insert an item in the binary search tree.

- Delete an item from the binary search tree.
- Find the height of the binary search tree.
- Find the number of nodes in the binary search tree.
- Find the number of leaves in the binary search tree.
- Traverse the binary search tree.
- Copy the binary search tree.

Clearly, every binary search tree is a binary tree. To do inorder, preorder, and postorder traversals of a binary search tree are the same as those for a binary tree.

Search

The function search searches the binary search tree for a given item. If the item is found in the binary search tree, it returns **true**; otherwise, it returns **false**. Because the pointer **root** points to the root node of the binary search tree, we must begin our search at the root node. Furthermore, because **root** must always point to the root node, we need a pointer, say **current**, to traverse the binary search tree. The pointer **current** is initialized to **root**.

If the binary search tree is nonempty, we first compare the search item with the info in the root node. If they are the same, we stop the search and return **true**. Otherwise, if the search item is smaller than the info in the node, we follow **left** to go to the left sub tree; otherwise, we follow **right** to go to the right sub tree. We repeat this process for the next node. If the search item is in the binary search tree, our search ends at the node containing the search item; otherwise, the search ends at an empty sub tree. Thus, the general algorithm is as follows:

```
if(root == NULL)
{
    Cout<<" Cannot search an empty tree" ;
    return;
}
```

```
else
{
      current = root;
      while (current != NULL && current->info != the element)
      {
           if(current->info > the element)
              current = current -> left;
          else
              current = current -> right;
      }
      if (current->info == the element)
          cout<< "found the element";
          return;
     else
         cout<<"the element does not exit"
         return
}
```

Insert

After inserting an item in a binary search tree, the resulting binary tree must also be a binary search tree. To insert a new item, first we search the binary search tree and find the place where the new item is to be inserted. Here we traverse the binary search tree with two pointers—a pointer, say **current**, to check the current node and a pointer, say **trailCurrent**, pointing to the parent of current. Because duplicate items are not allowed, our search must end at an empty sub tree. We can then use the pointer **trailCurrent** to insert the new item at the proper place. The item to be inserted, **insertItem**, is passed as a parameter to the function **insert**. The general algorithm is as follows:

a. Create a new node and copy insertItem into the new node. Also set left and right of the new node to NULL.

b. if the root is NULL, the tree is empty so make root point to the new node.

```
else
{
    current = root;
    while (current != NULL)        //search the binary tree
    {
        trailCurrent = current;
        if (current->info == the insertItem)
            cout<<"  Error: Cannot insert duplicate";
             exit
        else if(current->info > insertItem)
             current = current->left;
        else
             current = current->right;
    }

    //insert the new node in the binary tree

    if (trailCurrent->info > insertItem)
      trailCurrent->left = newnode   ( make the new node the left child of trailCurrent)
    else
          trailCurrent->right = newnode ( make the new node the right child of trailCurrent)
}
```

Delete

As before, first we search the binary search tree to find the node to be deleted. To help you better understand the delete operation, let us consider the binary search tree given in Figure 11-8.

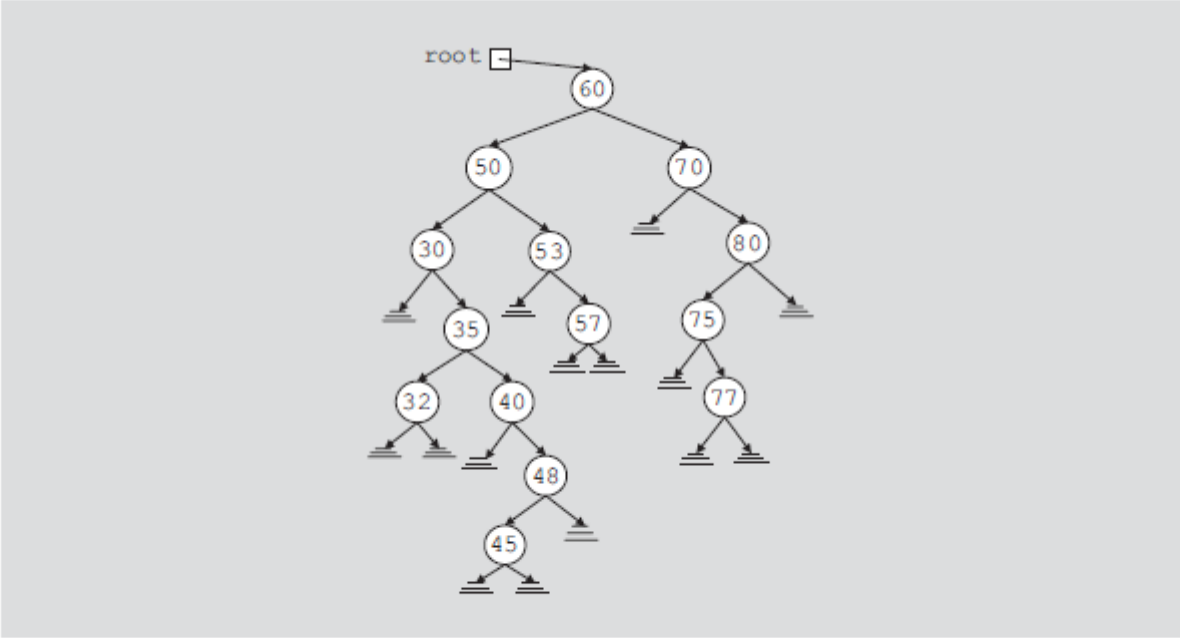

FIGURE 11-8 Binary search tree before deleting a node

After deleting the desired item (if it exists in the binary search tree), the resulting tree must be a binary search tree. The delete operation has four cases, as follows:

Case 1: The node to be deleted has no left and right sub trees; that is, the node to be deleted is a leaf. For example, the node with info 45 is a leaf.

Case 2: The node to be deleted has no left sub tree; that is, the left sub tree is empty, but it has a nonempty right sub tree. For example, the left sub tree of node with info 40 is empty and its right sub tree is nonempty.

Case 3: The node to be deleted has no right sub tree; that is, the right sub tree is empty, but it has a nonempty left sub tree. For example, the left sub tree of node with info 80 is empty and its right sub tree is nonempty.

Case 4: The node to be deleted has nonempty left and right sub trees. For example, the left and the right sub trees of node with info 50 are nonempty.

Figure 11-9 illustrates these four cases.

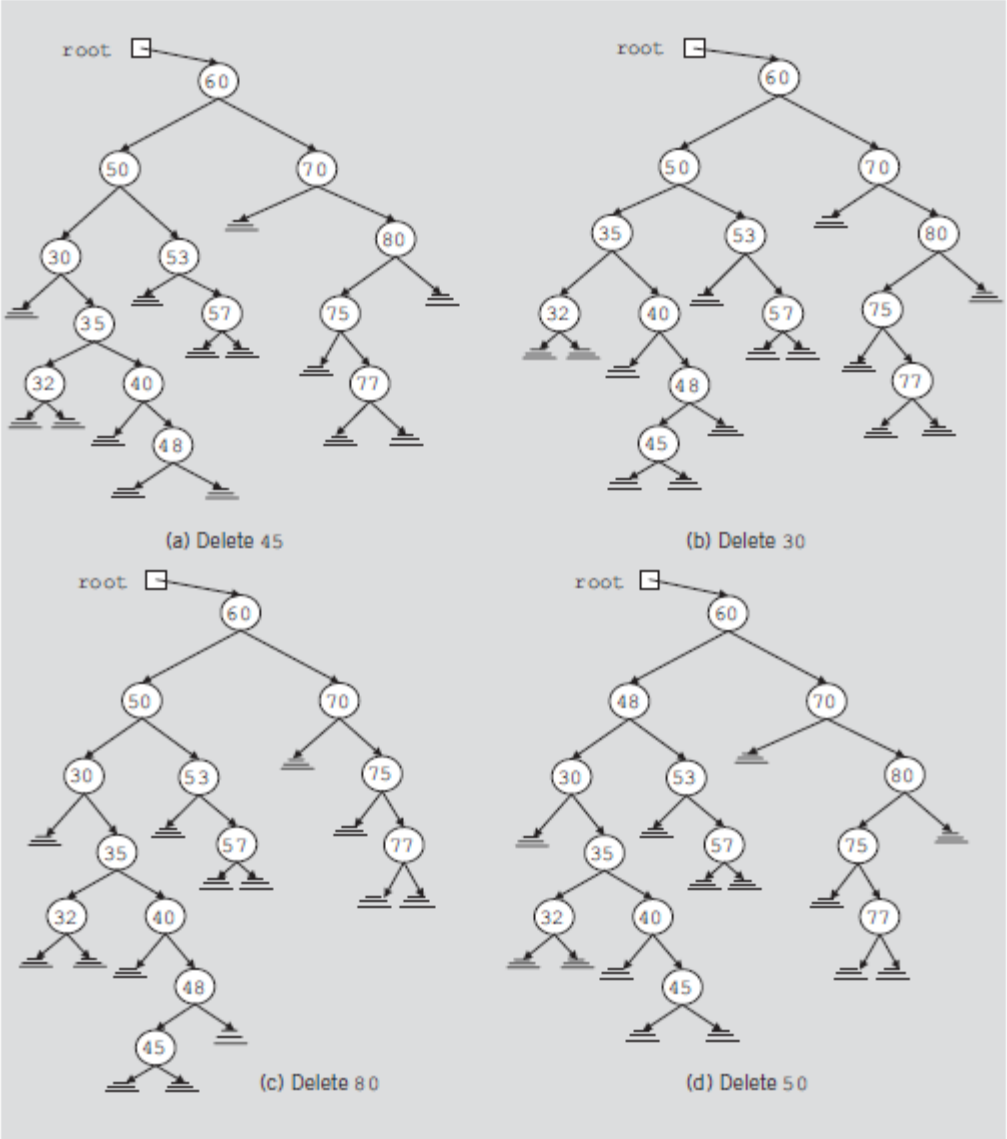

FIGURE 11-9 The binary tree of Figure 11-8 after deleting various items

Case 1: Suppose that we want to delete 45 from the binary search tree in Figure 11-8. We search the binary tree and arrive at the node containing 45. Because this node is a leaf and is the left child of its parent, we can simply set the llink of the parent node to NULL and deallocate the memory occupied by this node. After deleting this node, Figure 11-9(a) shows the resulting binary search tree.

Case 2: Suppose that we want to delete 30 from the binary search tree in Figure 11-8. In this case, the node to be deleted has no left sub tree. Because 30 is the left child of its parent node,

we make the llink of the parent node point to the right child of 30 and then deallocate the memory occupied by 30. Figure 11-9(b) shows the resulting binary tree.

Case 3: Suppose that we want to delete 80 from the binary search tree of Figure 11-8. The node containing 80 has no right child and is the right child of its parent. Thus, we make the rlink of the parent of 80—that is, 70—point to the left child of 80. Figure 11-9(c) shows the resulting binary tree.

Case 4: Suppose that we want to delete 50 from the binary search tree in Figure 11-8. The node with info 50 has a nonempty left sub tree and a nonempty right sub tree. Here, we first reduce this case to either Case 2 or Case 3 as follows. To be specific, suppose that we reduce it to Case 3—that is, the node to be deleted has no right sub tree. For this case, we find the immediate predecessor of 50 in this binary tree, which is 48. This is done by first going to the left child of 50 and then locating the rightmost node of the left sub tree of 50. To do so, we follow the rlink of the nodes. Because the binary search tree is finite, we eventually arrive at a node that has no right sub tree. Next, we swap the info in the node to be deleted with the info of its immediate predecessor. In this case, we swap 48 with 50. This reduces to the case wherein the node to be deleted has no right sub tree. We now apply Case 3 to delete the node. (Note that because we will delete the immediate predecessor from the binary tree, we, in fact, copy only the info of the immediate predecessor into the node to be deleted.) After deleting 50 from the binary search tree in Figure 11-8, the resulting binary tree is as shown in Figure 11-9(d).

In each case, we see that the resulting binary tree is again a binary search tree.
From this discussion, it follows that to delete an item from a binary search tree, we must do the following:

1. Find the node containing the item (if any) to be deleted.
2. Delete the node.