**Binary Search Trees**

Suppose that we want to determine whether 50 is in the binary tree. To do so, we can use any of the previous traversal algorithms 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 following Figure



In the binary tree in the figure the data in each node is

• Larger than the data in its left subtree

• Smaller than the data in its right subtree

The binary tree in the figure 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 != 60 and 58 < 60, it is guaranteed that 58 will not be in the right subtree of the root node. Therefore, if 58 is in the binary tree, it must be in the left subtree 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, wemust 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 subtrees 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 the the figure 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, LT and RT, called the left subtree and right subtree of T, respectively.

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

iv. LT and RT 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.

**Binary search tree** A BST is a binary tree that is either empty or where every node

contains a key and satisfies the following conditions:

1. The key in the left child of a node, if it exists, is less than the key in its parent node.

2. The key in the right child of a node, if it exists, is greater than the key in its parent node.

3. The left and right subtrees of a node are again BSTs.

The definition ensures that no two entries in a BST can have equal keys.

The following are the operations commonly performed on a BST:

1. Searching a key

2. Inserting a key

3. Deleting a key

4. Traversing the tree

Build a BST from the following set of elements—100, 50, 200, 300,

20, 150, 70, 180, 120, 30—and traverse the tree built in inorder, postorder, and preorder.

**Recursive Algorithm for Insertion in a Binary Search Tree**

**1. if** the root is **null**

**2.** Replace empty tree with a new tree with the item at the root and return **true**.

**3. else if** the item is equal to root.data

**4.** The item is already in the tree; return **false**.

**5. else if** the item is less than root.data

**6.** Recursively insert the item in the left subtree.

**7. else**

**8.** Recursively insert the item in the right subtree.

**Searching for a Key**

To search for a target key, we first compare it with the key at the root of the tree. If it is the same, then the algorithm ends. If it is less than the key at the root, search for the target key in the left subtree, else search in the right subtree.

**Recursive Algorithm for Searching a Binary Search Tree**

**1. if** the root is **null**

**2.** The item is not in the tree; return **null**.

**3.** Compare the value of **target**, the item being sought, with root.data.

**4. if** they are equal

**5.** The target has been found, return the data at the root.

**else if** target is less than root.data

**6.** Return the result of searching the left subtree.

**else**

**7.** Return the result of searching the right subtree.

**Deleting a node**

Deletion of a node is one of the frequently performed operations. Let *T* be a BST and *X* be the node of key *K* to be deleted from *T*, if it exists in the tree. Let *Y* be a parent node of *X*. There are three cases when a node is to be deleted from a BST. Let us consider each case:

1. *X* is a leaf node.

2. *X* has one child.

3. *X* has both child nodes.

**Case 1: Leaf node deletion** If the node to be deleted, say *X*, is a leaf node, then the process is easy. We need to change the child link of the parent node, say *Y* of node to be deleted to Null, and free the memory occupied by the node to be deleted and then return. Consider the following tree given in Fig. Here, 5 is the node to be deleted.



**Case 2(a): Node not having right subtree** If the node to be deleted has a single child link, that is, either right child or left child is Null and has only one subtree, the process is still easy. If there is no right subtree, then just link the left subtree of the node to be deleted to its parent and free its memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a left child, then we need to set *Y*->Lchild *X*->Lchild and free the memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a right child, then we need to set *Y*->Rchild

*X*->Lchild and free the memory. Let the node to be deleted be with data 16 and data 8; the resultant tree is as shown in following Figures (a) and (b), respectively.

**Case 2(b): Node not having left subtree** If there is no left subtree, then just link the right subtree of the node to be deleted to its parent and free its memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a left child, then we need to set *Y*->Lchild *X*->Rchild and free the memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a right child, then we need to set *Y*->Rchild *X*->Rchild and free the memory. Let the node to be deleted be with data 5 and data 2; the resultant tree is as in the following Figs (c) and (d), respectively.



**Case 3: Node having both subtrees** Consider the case when the node to be deleted has both right and left subtrees. This problem is more difficult than the earlier cases. The question is which subtrees should the parent of the deleted node be linked to, what should be done with the other subtrees, and where should the remaining subtrees be linked. One of the solutions is to attach the right subtree in place of the deleted node, and then attach the left subtree onto an appropriate node of the right subtree. This is pictorially shown in the following Figure





Another way to delete *X* from *T* is by first deleting the inorder successor of the node *X*, say *Z*, then replace the data content in the node *X* by the data content in the node *Z* (successor of the node *X*). Inorder successor means the node that comes after the node *X* during the inorder traversal of *T*.

Let us consider following Fig, and let the node to be deleted be the node with data 12.



In this process, we are actually trying to maintain the properties and the structure of a binary tree as much as possible. While deleting a node with both subtrees, we attempt searching the best suitable node to place at the deleted node. There are two alternatives to achieve so:

1. One can search for the largest data in the deleted node’s left subtree and replace the deleted node with it.

2. One can search for the smallest data from the deleted node’s right sub tree and replace the deleted node with it.

TreeNode \*BSTree::del(int deldata)

{

 int found = 0;

 int fl ag;

 TreeNode \*temp = Root, \*parent, \*x;

 if (Root == Null) {

 cout << endl << "\t BST is empty";

 return Null;

 }

 Else {

 parent = temp;

 //Search a BST node to be deleted & its parent

 while (temp != Null) {

 if (temp->Data == deldata)

 break; // found

 if (deldata < temp->Data) {

 parent = temp;

 temp = temp->Lchild;

 }

 else {

 parent = temp;

 temp = temp->Rchild;

 }

 } // end of search

 if (temp == Null)

 return(Null);

 else {

 //case of BST node having right children

 if (temp->Rchild != Null) {

 //fi nd leftmost of right BST node

 //cout << "\n Temp is having right child";

 parent = temp;

 x = temp->Rchild;

 while (x->Lchild != Null) {

 parent = x;

 x = x->Lchild;

 }

 temp->Data = x->Data;

 temp = x;

 }

 //case of BST node being a leaf Node

 if (temp->Lchild == Null && temp->Rchild == Null){

 //cout << "\n Leaf node";

 if (temp != root){

 if (parent->lLchild == temp)

 parent->Rchild = Null;

 else

 parent->Rchild = Null;

 }

 else

 root = Null;

 delete temp;

 return(root);

 }

 else if(temp->Lchild != Null&&temp->Rchild ==Null)

 //case of BST node having left children

 {

 //cout << “\n only left”;

 if (temp != root) {

 if (parent->Lchild == temp)

 parent->Lchild = temp->Lchild;

 else

 parent->Rchild = temp->Lchild;

 }

 else

 root = temp->Lchild;

 delete temp;

 return(root);

 }

 }

 }

}