BINARY SEARCH TREE

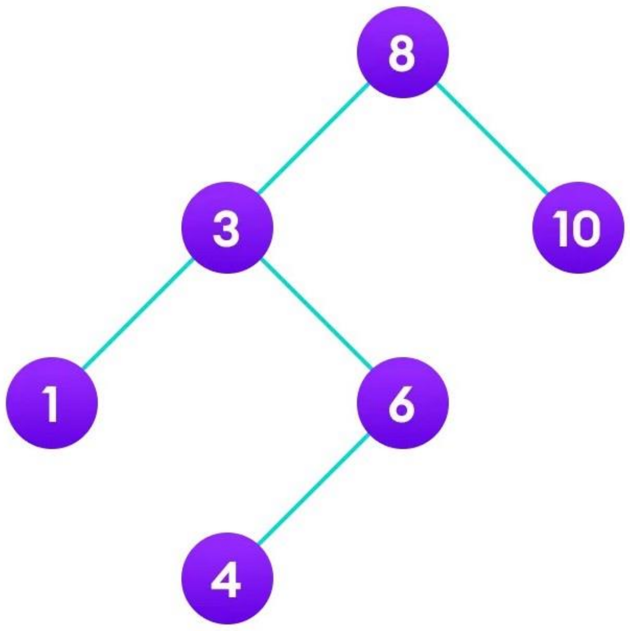
(IN CPP)

AUTHORS: ATHVAITHAM, SACHIN, DINESH DEVAN, NITHIL

<u>INDEX</u>

INTRODUCTION TO BINARY SEARCH TREE

Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value.



STRUCTURE OF A BST

```
class node {
int data;
node* left;
node* right;
};
```

We have implemented the bst structure in linked list.

Data refers to the node's data to be added.

*left denoted the address of left side node.

*right denotes the address of right side node.

Application of BST:

- Dynamic set operation
- Associative arrays
- Sorted data storage
- Symbol tables

Advantages of BST:

- Efficient searching
- Ordered structure
- Dynamic insertion
- Balanced structure
- Doubly ended priority queue

Disadvantages of BST:

- Not self-balanced
- Worst case time complexity
- Not suitable for large dataset
- Limited functionality

Now let us see the insertion, deletion and traversal in BST.

BST Traversal:

Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

Traversal is used to print every node in the tree.

There are three types of traversal in BST,

1. Preorder
2. Inorder
3. Postorder

Preorder traversal:

Here we print the node on its first visit during the traversal of a tree.

We print the nodes in the order like root node then left node then right node.

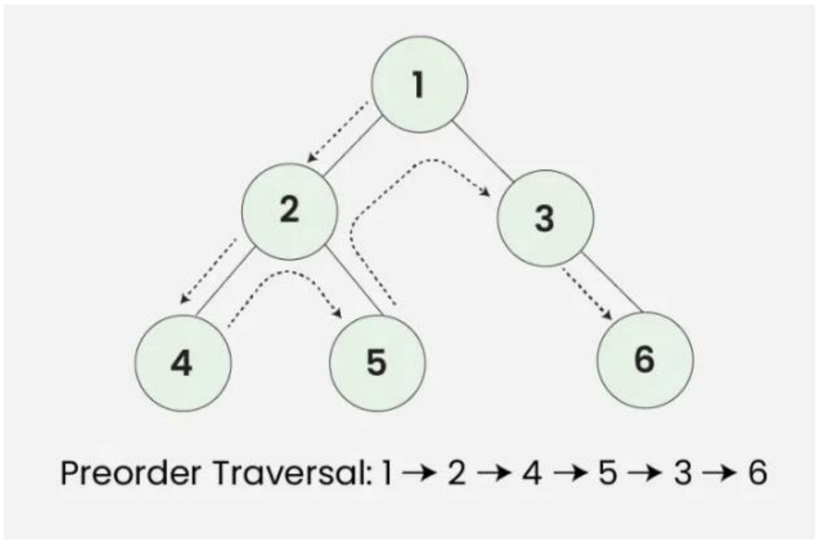
Time complexity:

Best case - O(h) h- height of BST

Worst case – O(n) n – no of nodes in BST

Algorithm:

- Visit the root
- Traverse the left subtree i.e. call Preorder (left subtree)
- Traverse the right subtree i.e. call Preorder(right subtree)
- Root → Left → Right



Code:

```
void preorder(node *root){
if(root == NULL)
return;
cout<<root->data; // root
preorder(root->left); // left
preorder(root->right); //right
}
```

Inorder Traversal:

Here we print the node on its second visit during the traversal of a tree.

We print the nodes in the order like left node then root node then right node.

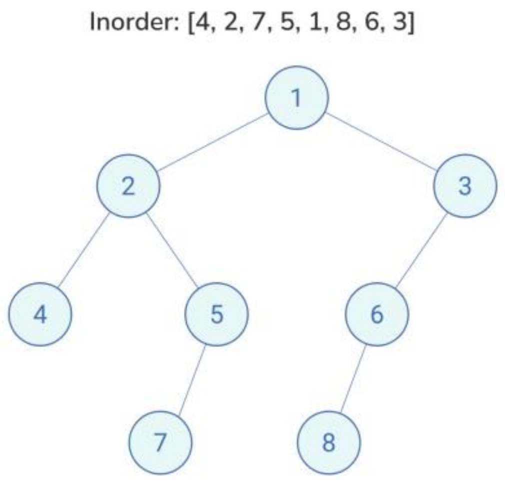
Time complexity:

Best case - O(h) h- height of BST

Worst case – O(n) n – no of nodes in BST

Algorithm:

- Traverse the left subtree i.e. call Inorder (left subtree)
- Visit the root
- Traverse the right subtree i.e. call Inorder(right subtree)
- Left → Root → Right



Code:

```
void inorder(node *root){
if(root == NULL)
return;
inorder(root->left); //left
cout<<root->data; // root
inorder(root->right); //right    }
```

Postorder Traversal:

Here, we go to the end node and print them.

We print the nodes in the order like left node then right node then root node.

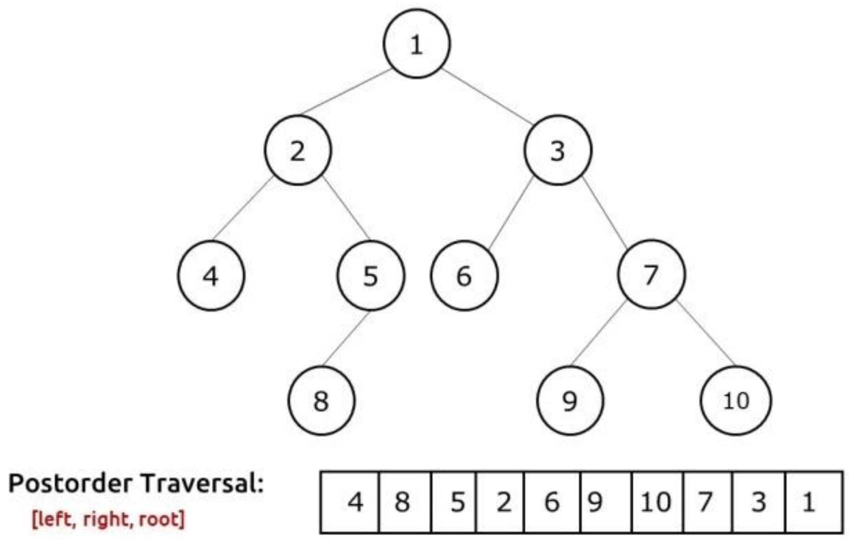
Time complexity:

Best case - O(h) h- height of BST

Worst case – O(n) n – no of nodes in BST

Algorithm:

- Traverse the left subtree i.e. call Postorder (left subtree)
- Traverse the right subtree i.e. call Postorder(right subtree)
- Visit the root
- Left → Right → Root



Code:

```
void postorder(node *root){
if(root == NULL)
return;
postorder(root->left); // left
postorder(root->right); // right
cout<<root->data; //root
}
```

Insertion in BST:

While inserting a node inside a BST, we must follow the rules of BST.

If there is no root node then the currently inserting node is the root node.

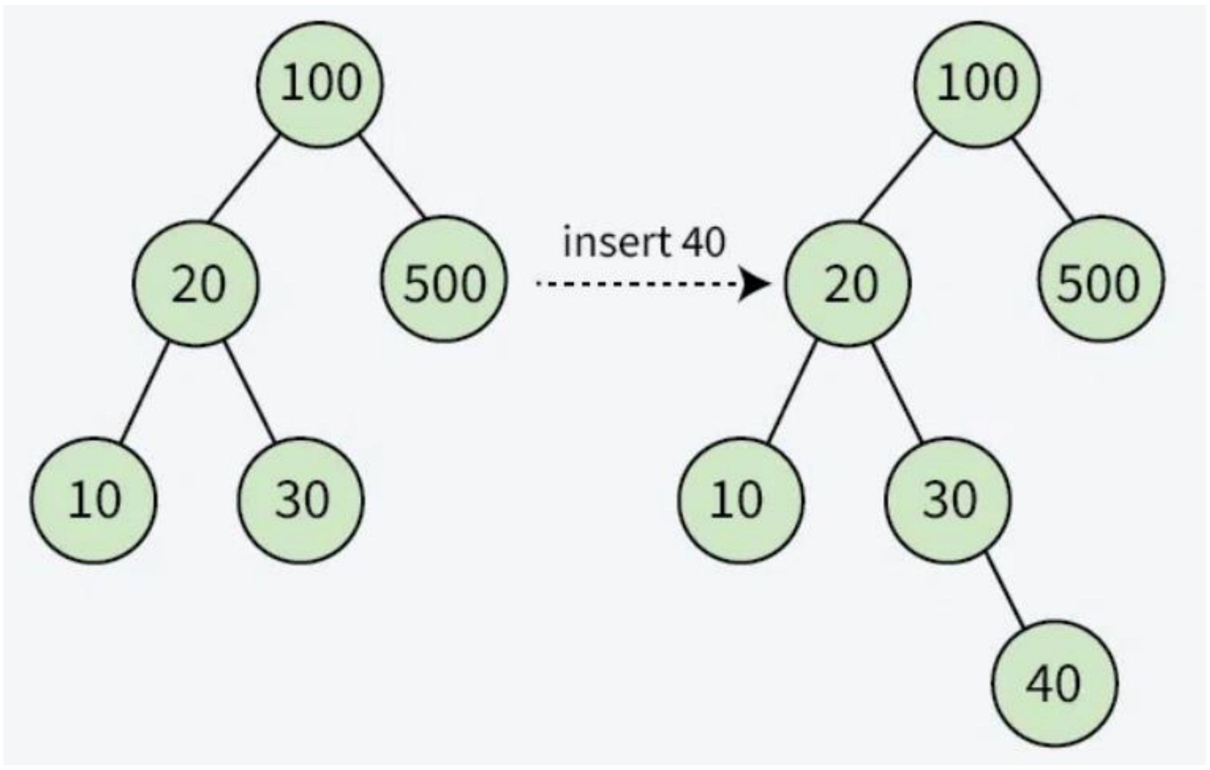
If there exists a root node, then the inserting node is compared with the root node.

- If it is smaller than the root node, then it is inserted in the left side of the root node or moved to the left side of the root node.
- If it is greater than the root node, then it is inserted in the right side of the root node or moved to the right side of the root node.

During this process, the comparison node for inserting a node may change.(only comparing).

The node is inserted when there is no node to compare and move.

Example:



Let us take this BST, Here 100 is the root node.

Now we want to insert 40 in this BST.

- First we compare 40 with the root node 100 and 40 is less than 100.
- Next we move left of the root node 100, now we have 20 on the left side of 100.
- Now we compare 40 with 20 and 40 is greater than 20. And we move to the right side of 20.
- We have 30 on the right side of 20, and now we compare 40 with 30. 40 is greater than 30. Therefore, we move to the right side of 30 and there is no node on the right side of 30.
- So we insert 40 on the right side of 30.

Now let us see the code insertion of nodes in BST.

```
class node{  //linked list definition for BST

public:

node data;

node *right,*left;

node (int x){       // constructor

data = x;

right = left =NULL; }

};

node *inserting(node *root, int key){

// checking the root is null or not

if (root==NULL)

return new node(key);  // if it's, create a new node and return it.

 // if root is not null and root's data is less than key

if (key<root->data)

 node->left = inserting(root->left, key);  // we move on left side of root
 // if root is not null and root's data is greater than key
elseif(key>node->data)
node->right = inserting(root->right, key);  // we move on right side of root


else      // if the key is equal to any node during the process
return root;
// we are not performing any insertion, since it is already in the BST
}
```

This code is in the concept of recursion. Since we compare different nodes during insertion, like if the inserting node is less than the root node then we move on the left side of the root and comparing the nodes in the left side of root and vice versa, if the inserting node is greater than the root node then we move on the right side of the root and comparing the nodes in the right side of root.

Deletion in BST:

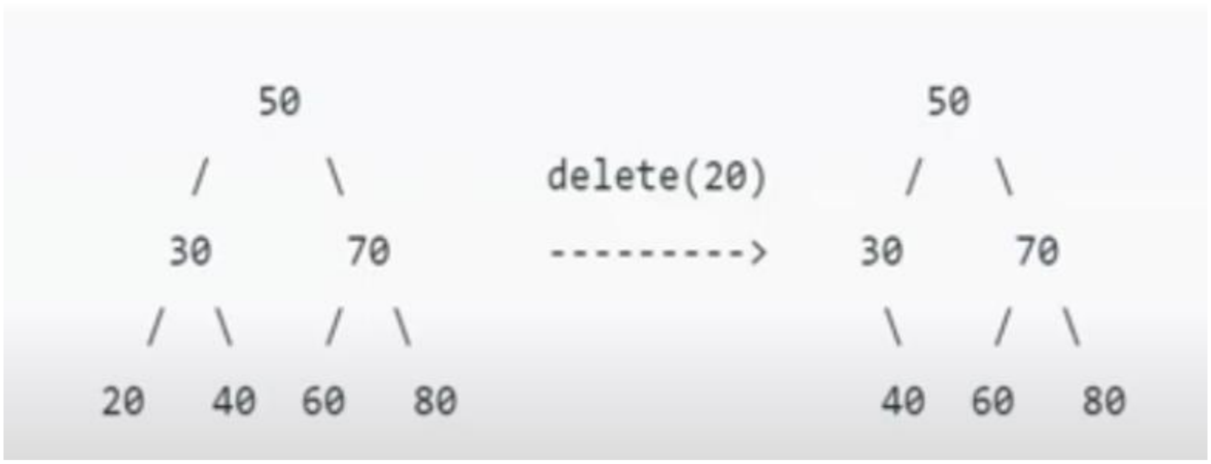
During the deletion of nodes in BST, there occurs three cases:

1. No children or no nodes below it
2. One child (left child or right child), left node or right node below it
3. Two children (both left and right child or node is below it)

Case 1:

This case will occur when the deleting node has no node below it neither left nor right node.

In this case, we can just delete the node simply.

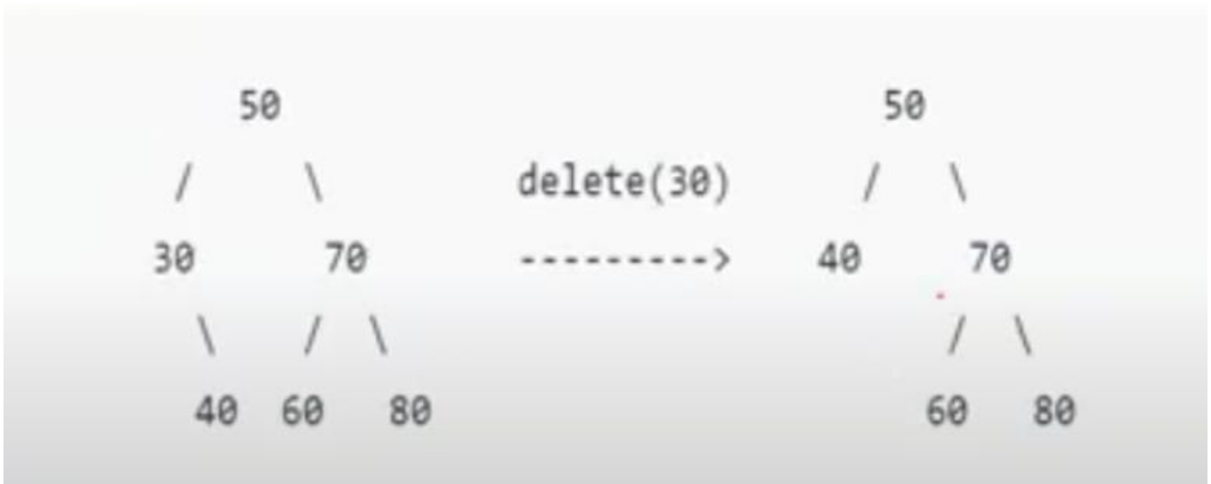


```
          50                                50
        /     \          delete(20)       /   \
      30       70        --------->     30     70
     /  \     /  \                        \   /  \
   20    40  60   80                      40  60   80
```

Case 2:

This case will occur when the deleting node has one node below it either left or right node.

In this case, we must replace the child node on the position of the parent node.

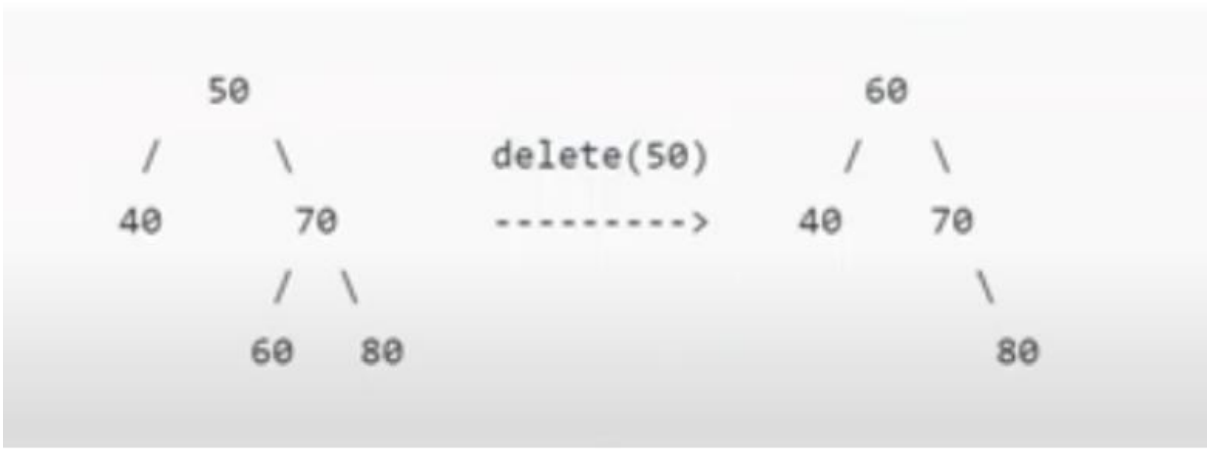


```
          50                              50
        /     \          delete(30)      /   \
      30       70        --------->    40     70
        \     /  \                           /  \
         40  60   80                        60   80
```

Case 3:

This case will occur when the deleting node has both left and right nodes below it.

In this case, we must find the inorder successor node or inorder predecessor node for the deleting node and replace the deleting node.



```
     50                              60
  /      \         delete(50)      /   \
40        70       --------->    40     70
        /   \                             \
       60    80                            80
```

Let us see the code for deleting a node.

```
node *delete(node *root, int key){
if (root == NULL)
return root;

if (root->data>key) // if root is not null and root’s data is less than key
root->left = delete(root->left, key); //continue deletion of key in left side of the
root

else if (root->data<key) // if root is not null and root’s data is greater than key
root->right = delete(root->right, key); //continue deletion of key in right side of
root
else{
// case1: no children
if(root->left == NULL && root->right == NULL)
return NULL;
// case 2: either left node or right node exists
else if(root->left == NULL){ // if only right node exists
node *temp = root->right;  // save the right node in temp node
delete root;  // delete the node to be deleted
return temp; // and return the temp node for replacing
}
else if(root->right == NULL){ //if only left node exists
node *temp = root->left; // save the left node in temp node
delete root; // delete the node to be deleted
return temp; //and the temp node for replacing
}
```

// case 3: both left and right nodes exists

node *succ = getsucc(root->right); // finding the inorder successor

root->data = succ->data; // replaced the data of root by inorder successor of it

root->right= delete(root->right, succ->data); // and delete the inorder successor

// Because right now the inorder successor is in two place (in root by replacing and in it is original place). So we delete the inorder successor in original place to avoid duplication

}

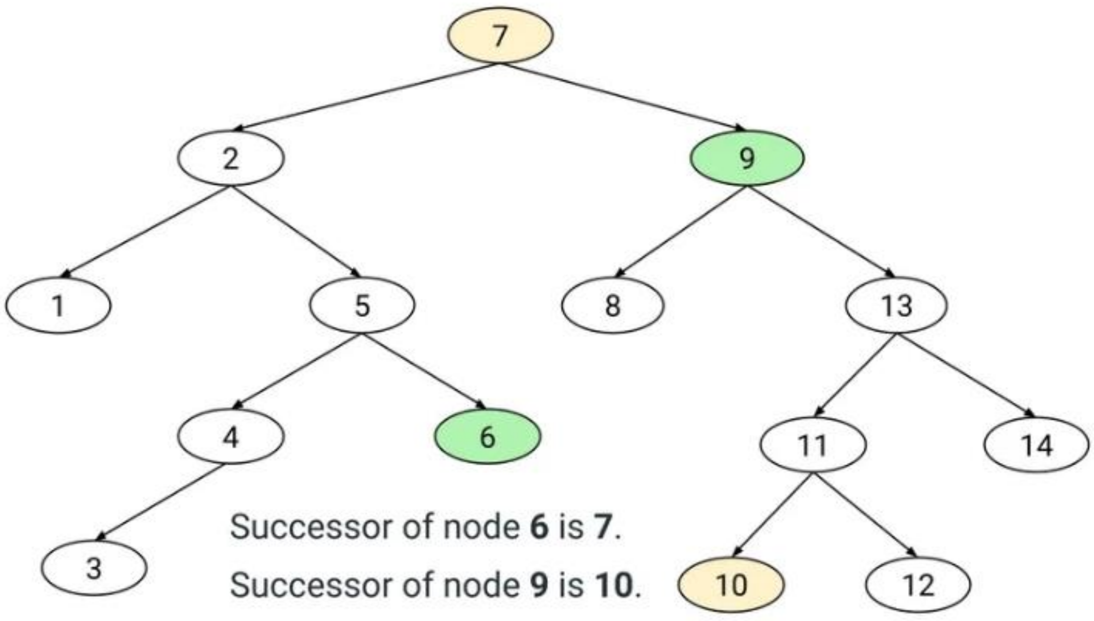
return root; // return root since the root may change during the deletion process

}

Inorder successor:

Let us see the explanation and code for finding the inorder successor of a node.

Inorder successor of a node is just the next node of the searching node in inorder traversal of the BST.



Inorder Traversal of the BST:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14]

There is a simple way to find the inorder successor of a node.

- Firstly we move one in the right side of the searching node
- And we go to the left most node and the left most node is the inorder successor of the searching node.

Here is the code for the finding the inorder successor.

```
node *get_succ(node *curr){
while (curr->left!=NULL) // move to the leftmost end
curr=curr->left;
return curr; // returns the leftmost node
}
```

In the deletion code, we have given the right node of the searching node.

Like node *succ = getsucc(root->right);

This is equivalent to the right jump for the first step of finding the inorder successor.

SEARCHING A NODE IN BST:

Searching a node in BST involves starting from the root and the search value with the current node' value.

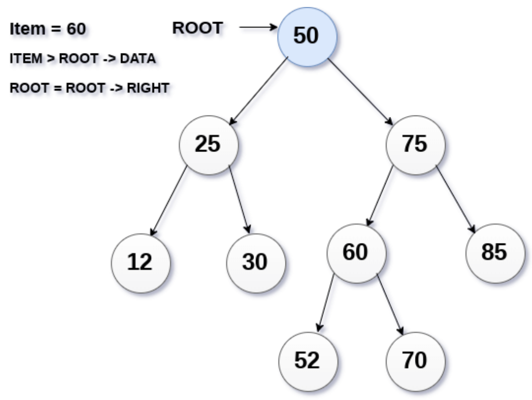
If the search value is lesser than the current node 's value, we move to the left child node or left subtree.

If the search value is greater than the current node's value, we move to the right child node or right subtree.

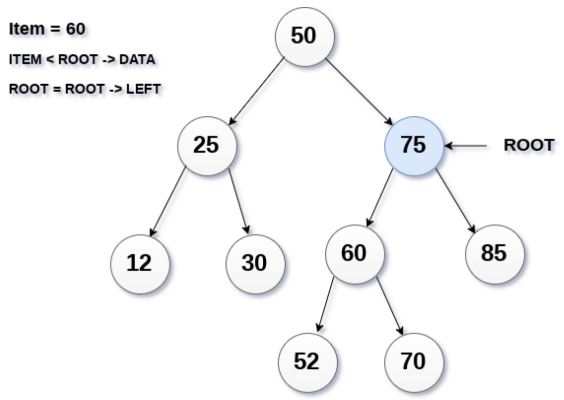
This process continues until you find the node or reach the end of the tree like reach a null node.

Here, the time complexity is O(h) h – height of the tree

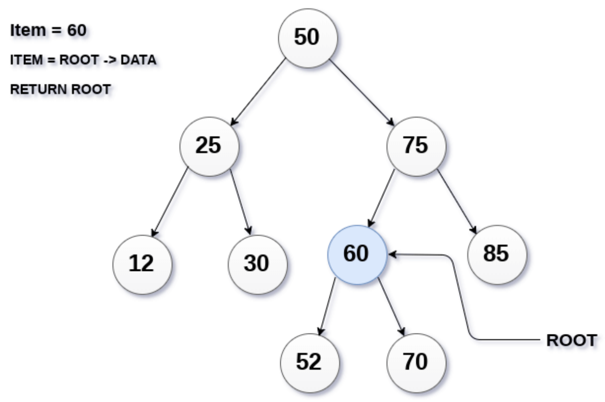
Example:



STEP 1



STEP 2



STEP 3

Here we have a BST with root 50 and we search 60 in this tree.

Step 1: First we compare the root node' data and search data, we see 50 is not equal to 60 and 60 is greater than 50. So we move towards right subtree.

Step 2: Now 75 is the root, 75 is not equal to 60 and 60 lesser than 75. So we move towards left subtree.

Step 3: Now 60 is the root, 60 is equal to the search value. Then the function returns the node where 60 is present.

Now let us see the code for searching a node in BST,

```
node searching(node *root, int search_val){
// root is null or search_val is present at the root
if(root == NULL || root->data == search_val)
        return root;   //return the root node
// search_val is greater than root's data, so we move toward right subtree
if(root->data < search_val)
        return searching(root->right, search_val)
//search_val is lesser than root's data, so we move toward left subtree
return searching(root->left, search_val);
}
```

<u>Problems on BST:</u>

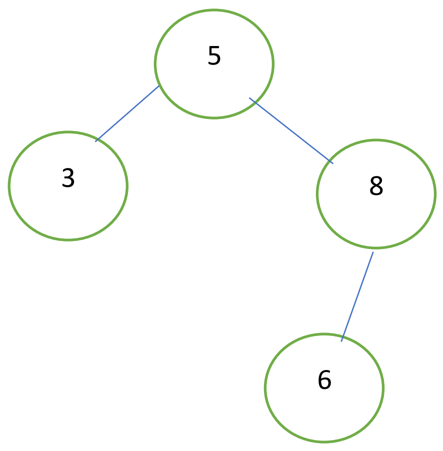
1. Find total no of nodes in the BST.

Solution:

Let us define a function for finding the total no of nodes in BST.

```
int total_nodes(node *root){
if(root == NULL)  // return zero if there is no node
        return 0;
// left variable has count of nodes in left subtree of the root or search node
int left = total_nodes(root->left);
// right variable has count of node in right subtree of the root or search node
Int right = total_nodes(root->right);
return 1+left+right;  // return sum of itself, left count and right count
}
```

Example:



Here 5 is the root node and sent inside the total_nodes.(og function call)

First it check the parameter node is NULL or not, it is not a NULL. So it goes to the next step left.

In left we call the same function by sending root's left node. (recursive function call 1)

Here 3 is in the left side of 5. 3 is sent inside the function, 3 is check to be null or not, it isn't. then it goes to next step sending the left of 3.

Here we have no nodes in the left of 3. So left variable's value will be 0.

Same as 3 have no right nodes, so right variable's value will be 0.

And returns, sum of 1+left+right is equal to 1. Stores it the left variable of the og function call.

Now, next step in og function call. In right we again call the function by sending root's right node.(recursive function call 2)

Here 8 is the right node of 5, 8 isnt a NULL. So it goes to the next step. Left variable the function is called again.

Since we have 6 in the left of 8. 6 is sent inside the function. (recursive function call 1 in recursive function call 2).

6 isnt a NULL. Then it moves to the next step, since 6 has no left and right nodes. Left variable and right variable has the value 0 and the sum as 1. (similarly to case node 3)

Now in recursive function call 2, the left variable stores 1 and moves to the next step in right variable, we call the function again by right nodes of 8.

Here, we have no nodes in the right side of 8. This function will return 0.And right variable stores 0.

Then the recursive function call 2, returns sum of left+right+1(itself) is equal to 2 to the right variable of the og function call.

Now in the og function call, return the sum of its left+right+1(itself) is equal to 1+2+1 = 4.

Therefore, there are 4 nodes is in the bst.

2. Find the sum of all nodes.

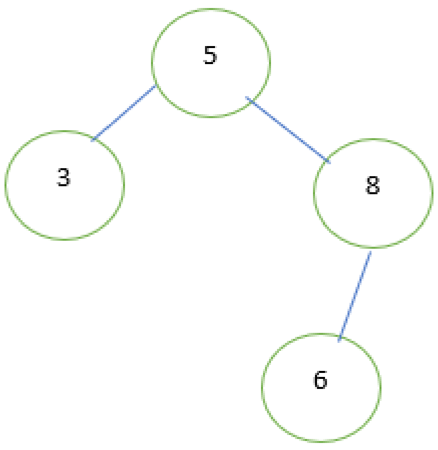
Solution:

The sum of all nodes function is similar to the function finding the count of the nodes.

Here is the code for it,

```
int sum_nodes(node *root){
if(root == NULL)
        return 0;
int left = sum_nodes(root->left);
int right = sum_nodes(root->right);
return (root->data + left + right);
}
```

Let us take the same BST from last problem,



As we saw how recursion working in previous problem. Let me explain this is in simpler way.

Here the 5 is the root node and sent inside the function, then left variable stores the sum of left subtree of 5(sum is 3) and right variable stores the sum of right subtree of 5(sum is 14) by recursion.

Then it returns the sum of root's data + left + right is equal to 5 + 3 + 14 is equal to 22.

AUTHORS NOTE:

In this e-book, we have tried our best to explain the concept of BST. Since we are also learning it.

There are n number problems in BST but we have explained only two problems with an understandable explanation (we guess so). For understanding the concepts of BST and recursive functions used in BST.

This e-book may help you in understanding the concepts of BST using c++.

THANK YOU

Reference:

1. https://www.geeksforgeeks.org/binary-search-tree-data-structure

2. https://youtu.be/u3QeyBtk7Jo?si=xpHiiq9X7FPvSndl