# 

# Department of Computing

# School of Electrical Engineering and Computer Science

**CS - 250: Data Structure and Algorithms**

**Class: BSCS 10AB**

**Lab 09 : Binary Search Tree**

**Date: 07th December, 2021**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Aftab Farooq

# Lab 09: Implementation of Binary Search Trees – Part A

**Introduction:**

This lab is based on the implementation of Binary Search tree and its functions.

**Objectives**

The objectives of this lab are the following:

* Become familiar with implementation of binary search trees
* Write simple applications using binary search tree

**Tools/Software Requirement**

Visual Studio 2012 or gcc or g++

**Description**

In computer science, a binary search tree (BST), which may sometimes also be called an ordered or sorted binary tree, is a node-based binary tree data structure which has the following properties:

* The left sub-tree of a node contains only nodes with keys less than the node's key.
* The right sub-tree of a node contains only nodes with keys greater than the node's key.
* Both the left and right sub-trees must also be binary search trees.

In this lab, you will expand implement binary search tree, study some statistical properties of BST and write a simple application using the BST. We will assume that duplicate insertions are not allowed.

Here is a template of how your class/structure looks like.

class BST\_Node{

Template data;

BST\_Node \*LeftChild;

BST\_Node \*RightChild;

};

**Lab Task**

**Tasks**

Implement the following operations of Binary Search Tree ADT

1. **bool IsEmpty();**

It checks whether the tree is empty or not. It returns true value, the tree is empty; false otherwise.

Code:

bool isEmpty()//checks whether the tree is empty

{

return length == -1;

}

1. **void Search(template value)**

It searches a value in a BST. It makes use of two pointer variables loc and ploc of type BT\_Node as explained in the class. If the searched value is found, loc should points to the node in which the searched value is found, and ploc to its parent node. If the value is not found, loc should contain NULL value, and ploc should point the logical parent node of the searched value. The following table represents the four possible combinations of values in loc and ploc and their interpretation:

|  |  |  |
| --- | --- | --- |
| **Loc** | **Ploc** | **Interpretation** |
| NULL | null | Value not found. It should be inserted as the root node implying the BST is currently empty. |
| non-null | Value not found. Ploc points to the logical parent node of the searched value. |
| Non-null | null | Value found in the root node of the BST. |
| non-null | Value found in a node other than the root. Loc points to the node in which the searched value is found; ploc points to its parent node. |

Code:

void Insert(int value)

{

tree\* newnode = new tree;

int flag = 0;

tree\* temp = new tree;

newnode->data = value;

newnode->left = l;

newnode->right = r;

if (!isEmpty())

temp = root;

flag = 0;

if (!Search(root, value)) //unique value then insert

{

if (isEmpty()) //the list is empty, then root node

{

root = newnode;

root->left = l;

root->right = r;

}

else //if the tree contains the value

{

while (flag == 0) //value is not inserted, then flag is false

{

if (value < temp->data) //if the value is less than current node

{

if (temp->left == l)

{

temp->left = newnode; //adding the node

flag = 1; //once inserted flag becomes true

}

else

temp = temp->left; //if left leaf is null, then temp value updated

}

else

{

if (temp->right == r) //if the value is more than current node

{

temp->right = newnode;//adding the node

flag = 1;//once inserted flag becomes true

}

else

temp = temp->right;//if right leaf is null, then temp value updated

}

}

}

length++;//the length of tree incremented

}

else

return;

}

1. **Void InsertWithoutDuplication(template value)**

This function calls the above mentioned Search() function to insert a new value in a BST.

* If the searched value already exists in the tree, its duplicate should not be inserted; exit the function by displaying a relevant message.
* If the search value is not found i.e. loc=NULL, the new value should be inserted using ploc.

Code:

bool Search(tree \*t,int value)

{

if (!isEmpty())

{

if (t->data == value) //if data is equal to root value

{

cout << " True" << endl; //if found, output true

return 1;

}

else

{

if (!(t == l || t == r))

{

count++;

Search(t->left, value); //treverse through each node of tree

Search(t->right, value);//treverse through each node of tree

}

}

}

if (count == length)

return 0;

}

1. Implement the following tree traversal methods
   1. **PreOrder traversal**

Code:

void PreOrder(tree\* node)

{

if (!(node == l || node == r))

{

cout << node->data << " "; //outputing first root node

PreOrder(node->left);//the left nodes and then right node of each node

PreOrder(node->right);

}

}

* 1. **InOrder traversal**

Code:

void InOrder(tree\* node)

{

if (!(node == l || node == r))

{

InOrder(node->left); //first left node, then root node and then right node

cout << node->data << " "; //then root node

InOrder(node->right);//then right node

}

}

* 1. **PostOrder traversal**

Code:

void PostOrder(tree\* node)

{

if (!(node == l || node == r))

{

PostOrder(node->left); //treversing through the left nodes

PostOrder(node->right);//treversing through the right nodes

cout << node->data << " "; //outputing a specific node

}

}

1. Implement a function that prints the **smallest** value of a BST.

Code:

int PrintSmall(tree\* node)

{

if (node->left == l)

return node->data;

if (!(node == l || node == r))

{

PrintSmall(node->left); //outputing the left most node for smallest value

}

}

1. Implement a function that prints the **largest** value of a BST.

Code:

int PrintLargest(tree\* node)

{

if (node->right == r)

return node->data;

if (!(node == l || node == r))

{

PrintLargest(node->right); // ouputing the right most node

}

}

1. Implement a function that traverses a tree and prints only its leaf nodes.

Code:

void PrintLeafs(tree\* node)

{

if (node->left == l && node->right == r) //the node which has no leaves, then output node

cout << " Node: " << node->data << endl;

if (!(node == l || node == r))

{

PrintLeafs(node->left); //treversing through the tree

PrintLeafs(node->right);//treversing through the tree

}

}

1. Implement a function that counts and returns the number of leaf nodes in a binary tree.

Code:

int CountLeafs(tree\* node)

{

if ((node->left == l && node->right == r))//the node that has no leaves, then increment c

++c;

if (!(node == l || node == r))

{

CountLeafs(node->left); //treversing through the tree

CountLeafs(node->right); //treversing through the tree

}

return c;

}

1. Implement a function that counts and returns the number of internal nodes in a binary tree.

Code:

int CountNodes(tree\* node)

{

if (!(node == l || node == r))

{

if ((node->left != l || node->right != r)) // the node which has leaves

++c\_;

CountNodes(node->left); //treversing through the tree

CountNodes(node->right); //treversing through the tree

}

return c\_;

}

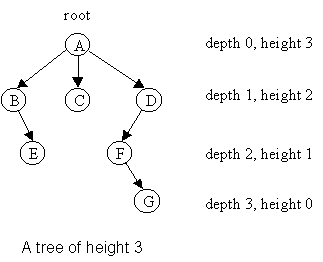


Figure : Height and Depth of Nods in Binary Tree (Source: <https://www.google.com/url?sa=i&url=https%3A%2F%2Fcondor.depaul.edu%2Fntomuro%2Fcourses%2F416%2Fnotes%2Flecture3-fall02.html&psig=AOvVaw2LmCcRw9mH1VJncXqqHfOT&ust=1638900929767000&source=images&cd=vfe&v>)

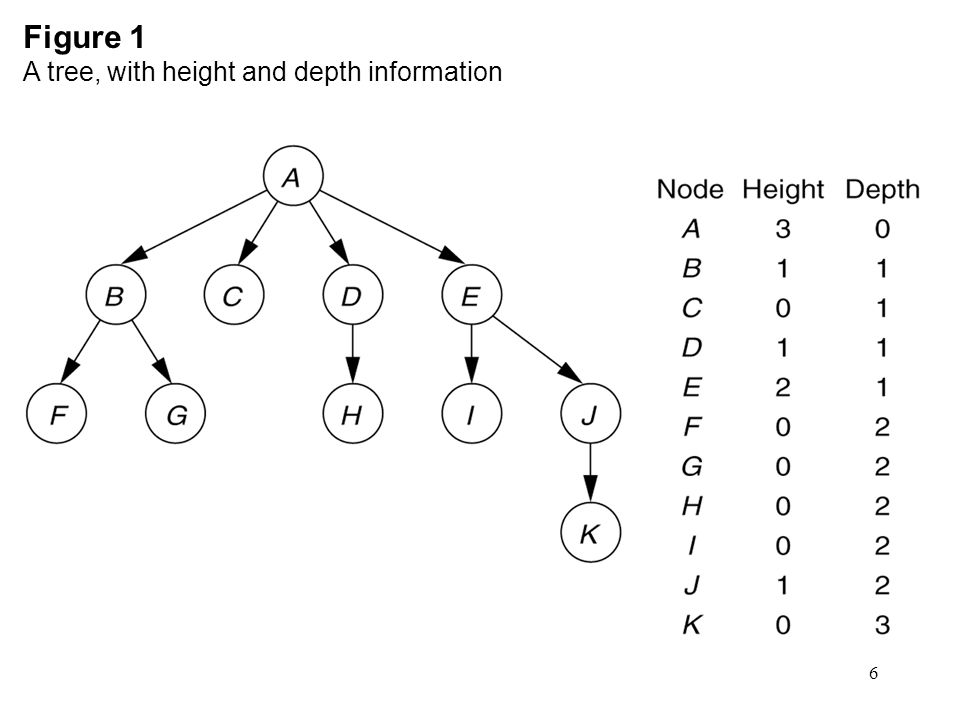


Figure : Height and Depth of Nodes in a BT (Source: https://images.slideplayer.com/42/11530377/slides/slide\_6.jpg)

1. Implement a function to calculate the **height** of a BST.

You ought to first identify the base case and recursive case definitions.

**Recursive Case:** To calculate the height of a node x, one needs to first calculate the height of its left sub-tree and right-subtree. The height of node x is 1+Max(height of x’s left subtree, height of x’s right sub-tree).

**Base Case:** Height of an empty tree is -1.

What is height of a leaf node? The answer is one. How? It is 1+max((height of x’s left subtree, height of x’s right sub-tree). For a leaf node, the height of its left sub-tree and its right sub-tree is -1.

Code:

int TreeHeight(tree\* node)

{

if (!isEmpty())

{

if (!(node == l || node == r))

{

leftD = TreeHeight(node->left);//treversing through the tree

rightD = TreeHeight(node->right);//treversing through the tree

if (leftD > rightD) //the bigger height of left or right branch, gets returned

return(leftD + 1); //returning left height branch

else

return(rightD + 1);//returning right height branch

}

else

return 0;

}

else

return -1; //if empty tree return -1.

}

1. Implement a function that calculates the **depth** of a BST**.**

You ought to first identify the base case and recursive case definitions. Here are some hints:

* Depth of the root node of a binary tree is 0.
* Depth of a node **x** in a binary tree is 1 plus the depth of its parent node.
* Pass the depth of the node or that of its parent for which the depth function is called.
* The **depth of a binary tree** rooted at node **y** is the number of nodes from the root y down to the furthest leaf node which may be in its left or right sub-tree.

Code:

int TreeDepth(tree\* node)

{

if (!isEmpty())

{

if (!(node == l || node == r))

{

leftD = TreeDepth(node->left);//treversing through the tree

rightD = TreeDepth(node->right);//treversing through the tree

if (leftD > rightD)//the bigger height of left or right branch, gets returned

return(leftD + 1);//returning left height branch

else

return(rightD + 1);//returning right height branch

}

else

return 0;

}

else

return -1;

}

**MAIN PROGRAM:**

int main()

{

t\* tre = new t;

tre->Insert(20);

tre->Insert(10);

tre->Insert(30);

tre->Insert(40);

tre->Insert(5);

tre->Insert(11);

tre->Insert(15);

cout << " Length: " << tre->length + 1 << endl;

if (tre->Search(tre->root, 16))

cout << " Search Found." << endl;

else

cout << " Search Not Found." << endl;

cout << " PreOrder: ";

tre->PreOrder(tre->root); cout << endl;

cout << " InOrder: ";

tre->InOrder(tre->root); cout << endl;

cout << " PostOrder: ";

tre->PostOrder(tre->root); cout << endl;

cout << " Smallest Value: " <<

tre->PrintSmall(tre->root); cout << endl;

cout << " Largest Value: " <<

tre->PrintLargest(tre->root); cout << endl;

cout << " Printing Leafs: \n";

tre->PrintLeafs(tre->root); cout << endl;

cout << " Numbers of Leafs: " << endl; cout << " " << tre->CountLeafs(tre->root);

cout << endl;

cout << " Numbers of Internal Nodes: " << endl;

cout << " " << tre->CountNodes(tre->root);

cout << endl;

cout << " Height of the Tree: ";

cout << tre->TreeHeight(tre->root) << endl;

cout << " Depth of the Tree: ";

cout << tre->TreeDepth(tre->root) << endl;

return 0;

}

**Screen Shot:**

Text

Description automatically generated