# 

# Department of Software Engineering

**CS 250: Data Structures and Algorithms**

**Class: BSCS-9AB**

**Lab 11: Binary Search Tree**

**Date: December 22, 2020**

**Time: 10:00 am -1:00pm, 2:00pm – 5:00pm**

# Instructor: Dr. Yasir Faheem

# Lab 10: Implementation of Binary Search Tree

**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
* Study some statistics 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.
* There must be no duplicate nodes.

In this lab, you will expand implement binary search tree, study some statistical properties of BST and write a simple application using the BST.

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

class BST\_Node{

Template data;

BST\_Node \*LeftChild;

BST\_Node \*RightChild;

};

**Lab Task**

You are required to upload the lab tasks on LMS and the name of that tasks must be in this format

FullName\_reg#\_task#.cpp

Remember to comment your code properly. Inappropriate or no comment will result in the deduction of marks.

**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.

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. |

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.

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

Implement a version of the insertion operation which allows duplication values. The left sub-tree of any node shall contain items that are less than or equal to its value. Items that are greater than a node’s value must go to its right sub-tree. Note that you cannot use the above implemented search function.

1. Implement the following tree traversal methods
   1. **PreOrder traversal**
   2. **InOrer traversal**
   3. **PostOrder traversal**
2. Implement a function that prints the **smallest** value of a BST.
3. Implement a function that prints the **largest** value of a BST.
4. 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 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.

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.
* 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.

**Deliverable**

#include<iostream>

#include <math.h>;

using namespace std;

class Node {

public:

int data;

Node\* leftchild;

Node\* rightchild;

};

class BinarySearchTree {

public:

Node\* root;

Node\* loc=NULL;

Node\* ploc=NULL;

int i =-1;

int max\_size = 0;

bool isEmpty()

{//To check wether tree is empty or not

return(root == NULL);

}

void Search(int value)

{//To search for a value in tree

//As a result loc and ploc will be updated

ploc = NULL;

loc = root;

//seraches until value is found or tree ends

while (loc != NULL && loc->data != value)

{//ploc points to the previous node of loc

ploc = loc;

if (loc->data > value)

{ //In BST,left side has small values and right side has larger

loc = loc->leftchild;

}

else { loc = loc->rightchild;}

}

}

void Insertion(int value)

{//To insert values in tree

Node\* nn = new Node();

//New node is created and assigned a value

nn->data = value;

if (isEmpty())

{//If tree is empty newnode is made as root

root = nn;

}

else {

//otherwise loc and ploc finds for the logical position and insert it

ploc = NULL;

loc = root;

while (loc != NULL)

{

if (loc->data > value)

{

ploc = loc;

loc = loc->leftchild;

}

else

{

ploc = loc;

loc = loc->rightchild;

}

}

if (ploc->data>value)

{

ploc->leftchild = nn;

}

else { ploc->rightchild = nn; }

}

}

void Insertwithoutduplicate(int data)

{

//Search is used to find logical position of node

Search(data);

if (loc == NULL)

{

Node\* nn=new Node();

nn->data = data;

if (ploc != NULL)

{

if (ploc->data > data)

{

ploc->leftchild = nn;

}

else

{

ploc->rightchild = nn;

}

}

else

{

root = nn;

}

}

else { cout << "duplicate insertions are not allowed"; }

}

void PreOrderTraversal(Node\* pointer)

{ //In this function tree is traversed first its root than left and right subtree

Node\* temp;

if (pointer == NULL)

{

return;

}

else

{

cout << pointer->data<<endl;

PreOrderTraversal(pointer->leftchild);

PreOrderTraversal(pointer->rightchild);

}

}

void PostOrderTraversal(Node\* pointer)

{ //In this function tree is traversed first its left subtree then root and right subtree

Node\* temp;

if (pointer == NULL)

{

return;

}

else

{ PreOrderTraversal(pointer->leftchild);

cout << pointer->data;

PreOrderTraversal(pointer->rightchild);

}

}

void InOrderTraversal(Node\* pointer)

{//In this function tree is traversed first its left subtree right subtree and then root

Node\* temp;

if (pointer == NULL)

{

return;

}

else

{

PreOrderTraversal(pointer->leftchild);

PreOrderTraversal(pointer->rightchild);

cout << pointer->data;

}

}

void SmallestValue(Node\* pointer)

{//recursively calls until left leaf node is accessed

if (pointer == NULL)

{

cout << "The minimum value is " << ploc->data<<endl;

return;

}

else

{

ploc = pointer;

SmallestValue(pointer->leftchild);

//cout << "The minimum value is " << pointer;

}

}

void LargestValue(Node\* pointer)

{//recursively calls until right leaf node is accessed

if (pointer == NULL)

{

cout << "The largest value is " << ploc->data<<endl;

return;

}

else

{

ploc = pointer;

LargestValue(pointer->rightchild);

//cout << "The largest value is " << pointer->data;

}

}

int Height(Node\* pointer)

{//Function to traverse left and right subtree of node to find the height

if (pointer == NULL)

{

return -1;

}

else

{

return (1 + max(Height(pointer>leftchild),Height(pointer>rightchild)));

}

}

int depth(Node\* pointer, int depthoftree)

{

if (pointer == NULL)

{

return depthoftree;

}

else

{

int depth\_ltree = depth(pointer->leftchild, depthoftree);

int depth\_rtree = depth(pointer->leftchild, depthoftree);

return max(depth\_ltree, depth\_rtree);

}

}

};

int main()

{

BinarySearchTree\* bt = new BinarySearchTree();

bt->Insertion(4);

bt->Insertion(8);

bt->Insertion(2);

bt->Insertion(1);

bt->Insertion(13);

bt->Insertion(7);

bt->Insertion(5);

bt->Insertion(6);

cout << "PreOrderTraversal" << endl;

bt->PreOrderTraversal(bt->root);

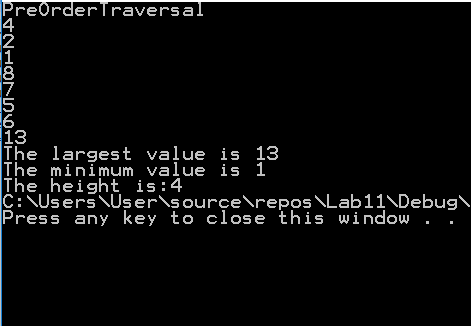
bt->LargestValue(bt->root);

bt->SmallestValue(bt->root);

cout<<bt->Height(bt->root);

}

**Output**



Students are required to upload the lab task on LMS before the deadline.