**Department of Computing**

**School of Electrical Engineering and Computer Science**

**CS250 – Data Structures and Algorithms**



**Lab 8: Binary Search Tree**

**Submission Details**

|  |  |
| --- | --- |
| Name | CMS ID |
| Muhammad Umer | 345834 |
| Group | GP – 1 |
| Lab Engineer | Anum Asif |
| Faculty Member | Bostan Khan |
| Class | BEE12 |
| Date | 01/04/2024 |
| Time | 10:00 am – 12:50 pm |

**Table of Contents**

[2 Binary Search Tree (BST) 3](#_Toc162860841)

[2.1 Introduction 3](#_Toc162860842)

[2.2 Objectives 3](#_Toc162860843)

[2.3 Tools/Software Requirement 3](#_Toc162860844)

[2.4 Deliverables 3](#_Toc162860845)

[3 Lab Task 4](#_Toc162860846)

# Binary Search Tree (BST)

## Introduction

This lab is based on binary search trees and its implementation.

## Objectives

Objective of this lab is to get familiar with the BST and implement it in a programming language.

## Tools/Software Requirement

* Visual Studio C++

## Deliverables

Compile a single word document by filling in the solution parts and submit this file on LMS. The name of word document should follow this format. i.e., YourFullName(reg)\_Lab#. You must show the implementation of the tasks in a complete manner to get your work graded.

# Lab Task

Fill in the missing code sections in the following BST helper template.

#include <iostream>

#include <cstdlib>

using namespace std;

class BinarySearchTree

{

private:

struct tree\_node {

tree\_node\* left;

tree\_node\* right;

int data;

};

public:

tree\_node\* root;

BinarySearchTree() {

root = nullptr;

}

bool isEmpty() const { return root == NULL; }

void print\_inorder();

void inorder(tree\_node\*);

void print\_preorder();

void preorder(tree\_node\*);

void print\_postorder();

void postorder(tree\_node\*);

void insert(int);

tree\_node\* remove(int d, tree\_node\* node) {

// First locate the element, if the element is not found,

simply generate an error message

// Then there will be three cases in removing d. Code accordingly

// 1. Removing a leaf node

// 2. Removing a node with a single child

// 3. Removing a node with 2 children,

then adjust its parents and grand children links

if (node == nullptr)

return nullptr;

else if (d < node->data)

{

node->left = remove(d, node->left);

return node;

}

else if (d > node->data)

{

node->right = remove(d, node->right);

return node;

}

else if (d == node->data)

{

// Write your code here

}

};

void BinarySearchTree::insert(int d) {

tree\_node\* t = new tree\_node;

tree\_node\* parent = root;

t->data = d;

t->left = nullptr;

t->right = nullptr;

if (root == nullptr) {

root = t;

return;

}

// Smaller elements should go to the left,

whereas larger elements should go to the right

else {

// Write your code here

}

}

void BinarySearchTree::print\_inorder() {

inorder(root);

cout << endl;

}

void BinarySearchTree::inorder(tree\_node\* p) {

if (p != nullptr)

{

inorder(p->left);

cout << p->data << "\t";

inorder(p->right);

}

}

void BinarySearchTree::print\_preorder() {

preorder(root);

cout << endl;

}

void BinarySearchTree::preorder(tree\_node\* p) {

// Write your code here

}

void BinarySearchTree::print\_postorder() {

postorder(root);

cout << endl;

}

void BinarySearchTree::postorder(tree\_node\* p) {

// Write your code here

}

void insertIntoTree(BinarySearchTree\* b) {

cout << "How many elements do you want to insert?" << endl;

int n;

cin >> n;

int val;

for (int i = 0; i < n; i++)

{

cout << "Enter element # " << i << ":" << "\t";

cin >> val;

b->insert(val);

}

}

int main()

{

BinarySearchTree b;

while (true) {

int menuOption = 0;

while (menuOption > 6 || menuOption < 1)

{

cout << "\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\t\* Select Option:\t\t\t\*" << endl;

cout << "\t\*\t1. Insert New Data\t\t\*" << endl;

cout << "\t\*\t2. In-Order Traversal\t\t\*" << endl;

cout << "\t\*\t3. Pre-Order Traversal\t\t\*" << endl;

cout << "\t\*\t4. Post-Order Traversal\t\t\*" << endl;

cout << "\t\*\t5. Delete an item\t\t\*" << endl;

cout << "\t\*\t6. Exit\t\t\t\t\*" << endl;

cout << "\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "> ";

cin >> menuOption;

if (menuOption == 6) {

return 0; //Exiting

}

switch (menuOption)

{

case 1:

{

insertIntoTree(&b);

break;

}

case 2:

{

b.print\_inorder();

break;

}

case 3:

{

b.print\_preorder();

break;

}

case 4:

{

b.print\_postorder();

break;

}

case 5:

{

cout << "Enter element to remove:" << endl;

int elem;

cin >> elem;

b.remove(elem, b.root);

}

}

}

}

getchar();

}

**Solution**

*/\**

*BST Help*

*\*/*

#include <cstdlib>

#include <iostream>

using *namespace* std;

*class* BinarySearchTree {

*private:*

*struct* tree\_node {

        tree\_node\* left;

        tree\_node\* right;

*int* data;

    };

*public:*

    tree\_node\* root;

    BinarySearchTree() {

        root = nullptr;

    }

*bool* isEmpty() *const* { return root == NULL; }

*void* print\_inorder();

*void* inorder(tree\_node*\**);

*void* print\_preorder();

*void* preorder(tree\_node*\**);

*void* print\_postorder();

*void* postorder(tree\_node*\**);

*void* insert(*int*);

    tree\_node*\** remove(*int* *d*, tree\_node*\** *node*) {

*// First locate the element, if the element is not found, simply*

*// generate an error message Then there will be three cases in removing*

*// d. Code accordingly*

*// 1. Removing a leaf node*

*// 2. Removing a node with a single child*

*// 3. Removing a node with 2 children, then adjust its parents and grand*

*//    children links*

        if (node == nullptr)

            return nullptr;

        else if (d < node->data) {

            node->left = remove(d, node->left);

            return node;

        }

        else if (d > node->data) {

            node->right = remove(d, node->right);

            return node;

        }

        else if (d == node->data) {

*// No child (leaf node)*

            if (node->left == nullptr && node->right == nullptr) {

                delete node;

                return nullptr;

            }

*// Single child (left or right child)*

            else if (node->left == nullptr) {

                tree\_node\* temp = node->right;

                delete node;

                return temp;

            }

            else if (node->right == nullptr) {

                tree\_node\* temp = node->left;

                delete node;

                return temp;

            }

*// Two children*

            else {

                tree\_node\* temp = node->right;

                while (temp->left != nullptr) {

                    temp = temp->left;

                }

                node->data = temp->data;

                node->right = remove(temp->data, node->right);

                return node;

            }

        }

        else {

            cout << "Element not found" << endl;

            return nullptr;

        }

    }

*void* insertIntoTree(BinarySearchTree*\** *b*) {

        cout << "How many elements do you want to insert?" << endl;

*int* n;

        cin >> n;

*int* val;

        for (*int* i = 0; i < n; i++) {

            cout << "Enter element #" << i << ": ";

            cin >> val;

            b->insert(val);

        }

    }

};

*void* BinarySearchTree::insert(*int* *d*) {

    tree\_node\* t = new tree\_node;

    tree\_node\* parent = root;

    t->data = d;

    t->left = nullptr;

    t->right = nullptr;

    if (root == nullptr) {

        root = t;

        return;

    }

*// Smaller elements should go to the left, whereas larger elements*

*// should go to the right*

    else {

        tree\_node\* curr = root;

        tree\_node\* parent = nullptr;

*// Find the parent of the new node*

        while (curr != nullptr) {

            parent = curr;

            if (t->data > curr->data) {

                curr = curr->right;

            } else {

                curr = curr->left;

            }

        }

*// Insert the new node*

        if (t->data < parent->data) {

            parent->left = t;

        } else {

            parent->right = t;

        }

    }

}

*void* BinarySearchTree::print\_inorder() {

    inorder(root);

    cout << endl;

}

*void* BinarySearchTree::inorder(tree\_node*\** *p*) {

    if (p != nullptr) {

        inorder(p->left);

        cout << p->data << " ";

        inorder(p->right);

    }

}

*void* BinarySearchTree::print\_preorder() {

    preorder(root);

    cout << endl;

}

*void* BinarySearchTree::preorder(tree\_node*\** *p*) {

    if (p != nullptr) {

        cout << p->data << " ";

        preorder(p->left);

        preorder(p->right);

    }

}

*void* BinarySearchTree::print\_postorder() {

    postorder(root);

    cout << endl;

}

*void* BinarySearchTree::postorder(tree\_node*\** *p*) {

    if (p != nullptr) {

        postorder(p->left);

        postorder(p->right);

        cout << p->data << " ";

    }

}

*void* display\_menu() {

    cout << "\*\*== Binary Search Tree ==\*\*" << endl;

    cout << "1) -> Insert New Data" << endl;

    cout << "2) -> In-Order Traversal" << endl;

    cout << "3) -> Pre-Order Traversal" << endl;

    cout << "4) -> Post-Order Traversal" << endl;

    cout << "5) -> Delete an Item" << endl;

    cout << "6) -> Exit" << endl;

}

*int* main() {

    BinarySearchTree b;

    display\_menu();

    while (true) {

        cout << "\nSelect an option:" << endl;

*int* menuOption = 0;

        while (menuOption > 6 || menuOption < 1) *// This loop can exit program*

        {

            cout << "> ";

            cin >> menuOption;

            if (menuOption == 6) {

                return 0; *// Exiting*

            }

            switch (menuOption) {

                case 1: {

                    b.insertIntoTree(&b);

                    break;

                }

                case 2: {

                    b.print\_inorder();

                    break;

                }

                case 3: {

                    b.print\_preorder();

                    break;

                }

                case 4: {

                    b.print\_postorder();

                    break;

                }

                case 5: {

                    cout << "Enter element to remove:" << endl;

*int* elem;

                    cin >> elem;

                    b.remove(elem, b.root);

                }

            }

        }

    }

    getchar();

}

**Output**

root@Zonularity:/home/zonularity/dsa# cd "/home/zonularity/dsa/lab\_8/" && g++ task.cpp -o task && "/home/zonularity/dsa/lab\_8/"task

\*\*== Binary Search Tree ==\*\*

1) -> Insert New Data

2) -> In-Order Traversal

3) -> Pre-Order Traversal

4) -> Post-Order Traversal

5) -> Delete an Item

6) -> Exit

Select an option:

> 1

How many elements do you want to insert?

11

Enter element #0: 20

Enter element #1: 18

Enter element #2: 15

Enter element #3: 19

Enter element #4: 7

Enter element #5: 16

Enter element #6: 43

Enter element #7: 33

Enter element #8: 47

Enter element #9: 50

Enter element #10: 40

Select an option:

> 2

7 15 16 18 19 20 33 40 43 47 50

Select an option:

> 3

20 18 15 7 16 19 43 33 40 47 50

Select an option:

> 4

7 16 15 19 18 40 33 50 47 43 20

Select an option:

> 5

Enter element to remove:

40

Select an option:

> 2

7 15 16 18 19 20 33 43 47 50

Select an option:

> 6