Esau Hervert
eah170630

Assignment 2

CSE 5343
June 21, 2019

Problem

We have discussed Binary Search Trees (BST). Write a program implementing a delete operation from BST. You will have to write the program to insert nodes in the BST also (we already did the algorithm in detail in the class for insert.)

Insert the following nodes in the order mentioned here.

40, 60, 20, 80, 50, 10, 30, 15, 5, 35, 25, 45, 55, 70, 90, 32, 33, 48, 46

Now:

1. Do an in-order traversal.
2. Make screen shot.
3. Delete 40 (you decide predecessor or successor).
4. Do in order traversal again.
5. make screenshot.
6. Now delete 20.
7. Do in-order traversal.
8. Make screen shot.

Submit the code and submit the screenshots.

Code

Code 1: Binary_Search_Tree_Functions

```
#ifndef BST_Functions
#define BST_Functions

#include <iostream>
#include <vector>

using namespace std;

// This is the structure which will represent an individual
// Node of our binary tree.
struct Node {
        int val;
        Node *parent;
        Node *left;
        Node *right;
};

// This is our binary tree.
class Binary_Search_Tree {
public:
        // Define an empty tree to be one with the root being NULL.
        Node *root = NULL;

        // The functions that we will use.
        void add(int); // Adds an individual integer to the tree.
        void addVector(vector<int>); // Adds a vector of integers.
        void In_Order_Traversal(); // Traverses the tree In-Order.

        // Finds the predecessor of the node with int value.
        Node* Predecessor(int);
        // Finds the successor of the node with int value.
        Node* Successor(int);

        // Deletes using the predecessor.
        void Delete_Predecessor(Node*);
        // Deletes using the successor.
        void Delete_Successor(Node*);
};

// This function adds a new done according to whether it is larger or
// smaller than the potential parent and traverses the tree until it
// finds a spot to place the node.
void addNode(Node *Child, Node *Temp) {
        // The direction we traverse the tree is dependent on the value
        // of the node we will add with the value of the current node
        // we are at.
        if (Temp->val > Child->val) {
                // If there is an open spot, then place the new node.
                if (Temp->left == NULL) {
                        Child->parent = Temp;
                        Temp->left = Child;
                }
                // Otherwise traverse the tree.
                else {
                        addNode(Child, Temp->left);
                }
        }
        else {
                // If there is an open spot, then place the new node.
                if (Temp->right == NULL) {
                        Child->parent = Temp;
                        Temp->right = Child;
                }
                // Otherwise traverse the tree.
                else {
                        addNode(Child, Temp->right);
                }
        }
}

void Binary_Search_Tree::add(int v) {
        // Create the new node to add.
        Node *New = NULL;
        New = new Node();
        New->val = v;

        // If the tree is empty, create the root.
        if (root == NULL) {
                root = New;
        }
        // If the tree is not empty, then traverse the tree to
        // find a spot.
        else {
                addNode(New, root);
        }
}

// This function will add a vector of integers to the tree
// in the order that they appear in the vector.
void Binary_Search_Tree::addVector(vector<int> a) {
        int i;
        for (i = 0; i < size(a); i++) {
                add(a[i]);
        }
}

// This function will traverse the tree In-Order.
void BST_IOT(Node *N) {
        if (N == NULL) {
                return;
        }

        BST_IOT(N->left);
        cout << N->val << " ";
        BST_IOT(N->right);
}

// Get rid of the argument in the call to traversal.
void Binary_Search_Tree::In_Order_Traversal() {
        BST_IOT(root);
}

// This function will find the node with value v.
```

```
Node* FindNode(int v, Node *root) {
        // Base case.
        if (root->val == v) {
                return root;
        }
        // Else traverse the node according to how the
        // values are arranged.
        else if (root->val < v) {
                return FindNode(v, root->right);
        }
        else {
                return FindNode(v, root->left);
        }
}

// We will find the predecessor of the given node N.
Node* FindPredecessor(Node *Temp, Node *N) {
        // This is one base case.
        // This is when we have that the node we are at
        // is a leaf node thus we cannot travel further.
        if (Temp->left == NULL && Temp->right == NULL) {

                // If this node is the predecessor.
                if (Temp->val < N->val) {
                        return Temp;
                }
                // If it is not the predecessor, then we must
                // Go through the ancestors until we find the
                // predecessor.
                else {
                        Temp = Temp->parent;
                        while (Temp != NULL) {
                                if (Temp->val >= N->val) {
                                        Temp = Temp->parent;
                                }
                                else if (Temp->val < N->val) {
                                        return Temp;
                                }
                        }
                        // If there is no predecessor, then
                        // return the Node.
                        if (Temp == NULL) {
                                return N;
                        }
                }
        }
        // If we are at a node but its value is smaller.
        else if (Temp->val < N->val) {
                // If we can traverse more, then we traverse.
                if (Temp->right != NULL) {
```

```
                FindPredecessor(Temp->right, N);
            }
            // Otherwise we have found the predecessor.
            else {
                return Temp;
            }
        }
        // Same logic as above but for the case of the node
        // that we are at being bigger.
        else {
            if (Temp->left != NULL) {
                FindPredecessor(Temp->left, N);
            }
            else {
                Temp = Temp->parent;
                while (Temp != NULL) {
                    if (Temp->val >= N->val) {
                        Temp = Temp->parent;
                    }
                    else if (Temp->val < N->val) {
                        return Temp;
                    }
                }
                if (Temp == NULL) {
                    return N;
                }
            }
        }
}

// This function finds the predecessor of a node by just
// Knowing what the node's value is.
Node* Binary_Search_Tree::Predecessor(int v)
{
        return FindPredecessor(root, FindNode(v, root));
}

// Same logic as the predecessor case, just that some
// Inequalities are flipped.
Node* FindSuccessor(Node *Temp, Node *N) {
        if (Temp->left == NULL && Temp->right == NULL) {
            if (Temp->val > N->val) {
                return Temp;
            }
            else {
                Temp = Temp->parent;

                while (Temp != NULL) {
                    if (Temp->val <= N->val) {
                        Temp = Temp->parent;
```

```
                                }
                                else if (Temp->val > N->val) {
                                        return Temp;
                                }
                        }
                        if (Temp == NULL) {
                                return N;
                        }
                }
        }
        else if (Temp->val > N->val) {
                if (Temp->left != NULL) {
                        FindSuccessor(Temp->left, N);
                }
                else {
                        return Temp;
                }
        }
        else {
                if (Temp->right != NULL) {
                        FindSuccessor(Temp->right, N);
                }
                else {
                        Temp = Temp->parent;

                        while (Temp != NULL) {
                                if (Temp->val <= N->val) {
                                        Temp = Temp->parent;
                                }
                                else if (Temp->val > N->val) {
                                        return Temp;
                                }
                        }
                        if (Temp == NULL) {
                                return N;
                        }
                }
        }
}

// Same as for Predecessor(int v)
Node* Binary_Search_Tree::Successor(int v) {
        return FindSuccessor(root, FindNode(v, root));
}

// Now we delete a node and use the predecessor to replace
// it, if it is not a leaf node.
void Binary_Search_Tree::Delete_Predecessor(Node *D) {

        // If the node is a leaf node, then we just delete.
        if (D->left == NULL && D->right == NULL) {
                // Free up the side of the parent.
                Node *Parent = D->parent;
                if (Parent->left == D) {
                        Parent->left = NULL;
                }
                else if (Parent->right == D) {
                        Parent->right = NULL;
                }
                if (Parent != NULL) {
                        delete D; // Delete node
                }
                return; // for the base case.
        }

        // Find the predecessor of the node to be deleted.
        Node *NPred = FindPredecessor(root, D);

        // Have the predecessor's value become the current
        // node's value, thus "deleting" the node.
        D->val = NPred->val;

        // Travel down the predecessors until we reach a
        // leaf node (base case).
        Delete_Predecessor(NPred);
}

void Binary_Search_Tree::Delete_Successor(Node *D) {

        // If the node is a leaf node, then we just delete.
        if (D->left == NULL && D->right == NULL) {
                // Free up the side of the parent.
                Node *Parent = D->parent;
                if (Parent->left == D) {
                        Parent->left = NULL;
                }
                else if (Parent->right == D) {
                        Parent->right = NULL;
                }
                if (Parent != NULL)
                        delete D;
                return;
        }

        // Find the successor of the node to be deleted.
        Node *NSucc = FindSuccessor(root, D);

        // Have the successors's value become the current
        // node's value, thus "deleting" the node.
        D->val = NSucc->val;
        // Travel down the successors until we reach a
        // leaf node (base case).
        Delete_Successor(NSucc);
}

#endif BST_Functions
```

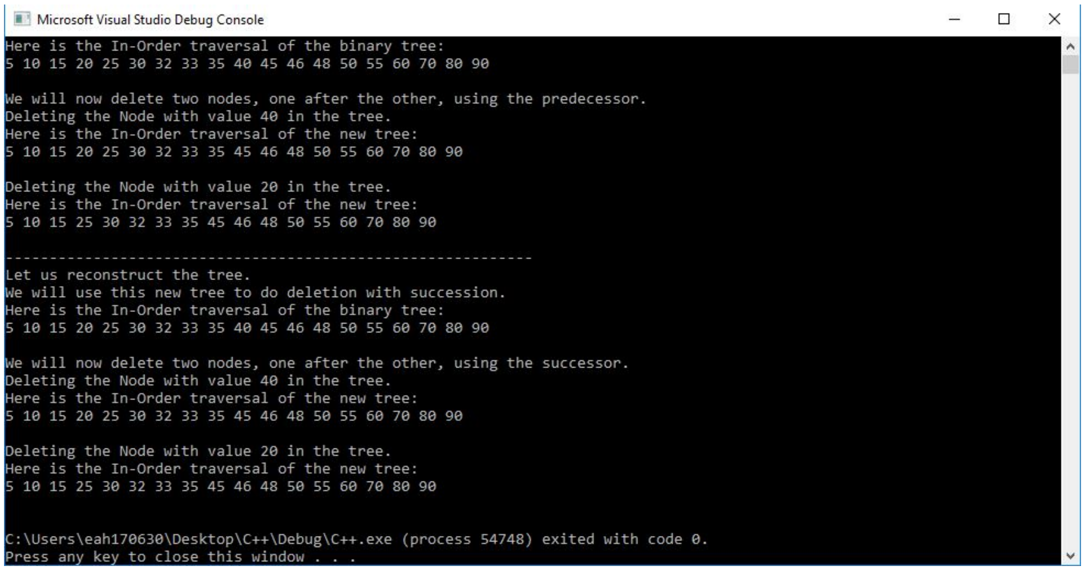
Code 2: Assignment_2_Deleting_BST

```
#include "Binary_Search_Tree_Functions.h"

#include <iostream>
#include <vector>

using namespace std;

int main() {

/***************************************************************************/
        // We initialize a new Binary tree.
        // This Binary tree we will use the predecessor to delete.
        Binary_Search_Tree B1P;

        // This vector holds the values that the binary tree will have.
        // The order is the order that they are to be added to the tree.
        vector<int> A = { 40, 60, 20, 80, 50, 10, 30, 15,
                5, 35, 25, 45, 55, 70, 90, 32, 33, 48, 46 };

        // We add this vector to this function that will add the individual
        // numbers in the order of the vector.
        B1P.addVector(A);

        // Traverse the tree in order.
        cout << "Here is the In-Order traversal of the binary tree: ";
        cout << endl;
        B1P.In_Order_Traversal();
        cout << endl << endl;
/***************************************************************************/

/***************************************************************************/
        // Now to delete using the predecessor.
        cout << "We will now delete two nodes, one after the other, ";
        cout << "using the predecessor." << endl;

        // Find the node that contains the value 40.
        Node *D1P = FindNode(40, B1P.root);

        // Delete this node and replace it with its predecessor.
        cout << "Deleting the Node with value 40 in the tree." << endl;;
        B1P.Delete_Predecessor(D1P);
        // Now traverse the tree.
        cout << "Here is the In-Order traversal of the new tree: " << endl;
        B1P.In_Order_Traversal();
        cout << endl << endl;

        // Find the node that contains the value 40.
        Node *D2P = FindNode(20, B1P.root);

        // Delete this node and replace it with its predecessor.
        cout << "Deleting the Node with value 20 in the tree." << endl;
        B1P.Delete_Predecessor(D2P);

        // Now traverse the tree.
        cout << "Here is the In-Order traversal of the new tree: " << endl;
        B1P.In_Order_Traversal();
        cout << endl << endl;
/***************************************************************************/

        cout << "------------------------------------------------------------------------";
        cout << endl;

/***************************************************************************/
        // We initialize a new Binary tree.
        // This Binary tree we will use the predecessor to delete.
        Binary_Search_Tree B1S;

        // We add the previous vector to this function that will add the
        // individual numbers in the order of the vector.
        B1S.addVector(A);

        // Traverse the tree in order.
        cout << "Let us reconstruct the tree." << endl;
        cout << "We will use this new tree to do deletion with succession.";
        cout << endl;
        cout << "Here is the In-Order traversal of the binary tree: " << endl;
        B1S.In_Order_Traversal();
        cout << endl << endl;
/***************************************************************************/

/***************************************************************************/
        // Now to delete using the successor.
        cout << "We will now delete two nodes, one after the other, ";
        cout << "using the successor." << endl;

        // Find the node that contains the value 40.
        Node *D1S = FindNode(40, B1S.root);

        // Delete this node and replace it with its predecessor.
        cout << "Deleting the Node with value 40 in the tree." << endl;;
```

```
    B1S.Delete_Successor(D1S);

    // Now traverse the tree.
    cout << "Here is the In-Order traversal of the new tree: " << endl;
    B1S.In_Order_Traversal();
    cout << endl << endl;

    // Find the node that contains the value 40.
    Node *D2S = FindNode(20, B1S.root);

    // Delete this node and replace it with its predecessor.
    cout << "Deleting the Node with value 20 in the tree." << endl;
    B1S.Delete_Successor(D2S);

    // Now traverse the tree.
    cout << "Here is the In-Order traversal of the new tree: " << endl;
    B1S.In_Order_Traversal();
    cout << endl << endl;
}
```

Results

I did both the deletion using the predecessor and successor. I went ahead and did all the steps on one run. Here is the results.



```
Here is the In-Order traversal of the binary tree:
5 10 15 20 25 30 32 33 35 40 45 46 48 50 55 60 70 80 90

We will now delete two nodes, one after the other, using the predecessor.
Deleting the Node with value 40 in the tree.
Here is the In-Order traversal of the new tree:
5 10 15 20 25 30 32 33 35 45 46 48 50 55 60 70 80 90

Deleting the Node with value 20 in the tree.
Here is the In-Order traversal of the new tree:
5 10 15 25 30 32 33 35 45 46 48 50 55 60 70 80 90

------------------------------------------------------------
Let us reconstruct the tree.
We will use this new tree to do deletion with succession.
Here is the In-Order traversal of the binary tree:
5 10 15 20 25 30 32 33 35 40 45 46 48 50 55 60 70 80 90

We will now delete two nodes, one after the other, using the successor.
Deleting the Node with value 40 in the tree.
Here is the In-Order traversal of the new tree:
5 10 15 20 25 30 32 33 35 45 46 48 50 55 60 70 80 90

Deleting the Node with value 20 in the tree.
Here is the In-Order traversal of the new tree:
5 10 15 25 30 32 33 35 45 46 48 50 55 60 70 80 90


C:\Users\eah170630\Desktop\C++\Debug\C++.exe (process 54748) exited with code 0.
Press any key to close this window . . .
```

Figure 1: Results