```cpp
/*//-----------------------------------------------------------------------------------
Justin Dang
Student ID : 1148267
//-------------------------------------------------------------------------------------
FUNCTION OF THE FOLLOWING CODE >>
- Takes in 15 names and weights.

- Prints the characteristics of the tree(Height, # of leaves, lowest weight, name lowest in the alphabet)

- Contains a search method(not implemented, but working properly) that searches the tree for a name, returning if they
  exist within the tree or not.
//-------------------------------------------------------------------------------------
WORKS CITED >>

Traversal of a tree: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

Height of a tree:    https://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/
*///-----------------------------------------------------------------------------------

#include <iostream>
#include <string>
using namespace std;

// Blueprint for each node in our binary tree
class Node {
public:
    int weight;                         // Weight of each person.
    string name;                        // Name of each person.
    Node* left, * right;                // address of next Name Node.

    // node(int, string, *leftWeightPtr, *rightWeightPtr, *leftNamePtr, *rightNamePtr)
    Node(int info, string info1, Node* lNamePtr = 0, Node* rNamePtr = 0) { // Structure for each node
        weight = info;
        name = info1;
        left = lNamePtr;
        right = rNamePtr;
    }
};
```

```cpp


class BinaryTree {
public:
    Node* root, * lowestWeight, * firstName;


    BinaryTree() { root = 0; }

    /*
    Adds a node to our binary tree.

    Organizes our nodes by data(int) size:
    1. if larger  -> we proceed down the right side of our tree
    2. if smaller -> we proceed down the left side of our tree

    The node is then placed at the end as a leaf of our tree

    (int weight, string name)
    */
    void addNode(int data, string data1) {
        Node* newNode = new Node(data, data1);


        if (root == 0) {                    // Check if the tree is empty(no root), if so simply set the root to our newNode.
            root = newNode;
            lowestWeight = newNode;         // Set our lowestWeight to our root.
            firstName = newNode;            // Set our firstName to our root.
        }

        if(lowestWeight->weight > data)     // If our new user has a lower weight than our current lowest weight user,
            lowestWeight = newNode;         // we track that user with our Node lowestWeight.

        if (firstName->name > data1)        // If our new user has a smaller name than our current first name user,
            firstName = newNode;            // we track that user with our Node firstName.

        Node* temp = root;                  // Since we dont want to alter/manipulate our root, we create a copy to traverse our list.

        while (true) {
            if (data1 < temp->name) {       // Starting at the root, we check if newNode has a smaller int than our root,
                                            // if so we proceed down the left side of our tree.

                if (temp->left == 0) {      // We then check if our root has left node already in place.
```

```
                    temp->left = newNode;  // If not we set our root's left node to our newNode.
                    break;
                }
                else
                    temp = temp->left;     // If our left node is taken, then we continue down the list.
            }
            else if (data1 > temp->name) { // Else if our newNode is larger than the in in our root, we proceed down the
                                           // right side of our tree.

                if (temp->right == 0) {    // Check if our root has a right node in place
                    temp->right = newNode; // if not we set our root's right node to our newNode.
                    break;
                }
                else
                    temp = temp->right;    // If our right node is taken, then we continue down the list.
            }
            else if (data1 == temp->name) {// No duplicates should be entered as written in ref doc(1).
                break;
            }
        }
    } // AddNode End--------------

    /*
    Prints the following:
        1) Height
        2) # of leaves
        3) Lightest person (formatted as:name @ weight)
        4) First name (whoever's name is the lowest in the alphabet)
    */
    void Characteristics() {
        cout << "Characteristics - Height: " << Height(root) - 1 << ", Leaves: " << Leaves(root) << ", Lightest: " <<
        lowestWeight->name << " @ " << lowestWeight->weight << ", First Name: " << firstName->name << endl;
    } // Characteristics End--------------

    int Height(Node* currentNode) {
        if (currentNode == 0)                     // Stops recursion if we find an empty node.
            return 0;
        else{
            int lDepth = Height(currentNode->left); // Traverse left subtree.
            int rDepth = Height(currentNode->right);// Traverse right subtree.

            if (lDepth > rDepth)                  // In short, calculates the
```

```
length of each subtree
                return lDepth + 1;                // taking the depth of the larger of the two(considering
            else                                  // we are using a binary tree that has two subtrees).
                return rDepth + 1;
        }
    } // Height End--------------

    int Leaves(Node* currentNode) {
        int leaves = 0;
        if (currentNode == 0)                     // Returns 0 leaves to our total if we find an empty node   .
            return 0;
        else {
            if (currentNode->left == 0 && currentNode->right == 0)
                leaves++;                         // Adds 1 to our leaf total if we we found a leaf in our tree.

            leaves += Leaves(currentNode->left);    // Used to traverse the tree and determine if each node
            leaves += Leaves(currentNode->right);   // is a leaf, adding to our leaf total after traversing a subtree.
        }
        return leaves;
    } // Leaves End--------------

    /*
    prints tree in Inorder
    */
    void PrintLVR(Node* currentNode) {
        if (currentNode != 0) {
            PrintLVR(currentNode->left);        // traverse left subtree            L
            cout << currentNode->name << " | "; // evaluate (print) current node    V
            PrintLVR(currentNode->right);       // traverse right subtree           R
        }
    } // printLVR End--------------

    /*
    prints tree in PreOrder
    */
    void PrintRVL(Node* currentNode) {
        if (currentNode != 0) {
            cout << currentNode->name << " | "; // evaluate (print) current node    V
            PrintRVL(currentNode->left);        // traverse left subtree            L
            PrintRVL(currentNode->right);       // traverse right subtree           R
```

```
        }
    } // printRVL End--------------

    /*
    prints tree in PostOrder
    */
    void PrintLRV(Node* currentNode) {
        if (currentNode != 0) {
            PrintRVL(currentNode->left);        // traverse left subtree R
            PrintRVL(currentNode->right);       // traverse right subtree L
            cout << currentNode->name << " | "; // evaluate (print) current node V
        }
    } // printLRV End--------------

    void Search(Node* currentNode, string targetName) {
        if (currentNode != 0) {
            while (true) {
                if (targetName == currentNode->name) {      // Stops method if target name is found within tree.
                    cout << endl << currentNode->name << " - " << currentNode->weight << " | exists in tree.";
                    return;
                }
                else if (targetName < currentNode->name) {  // Starting at the root, we check if our target name has a smaller name
                                                            // than our current node, if so we proceed down the left side of our tree.
                    if (currentNode->left == 0)
                        break;
                    currentNode = currentNode->left;        // If our left node exists, then we continue down the tree.
                }
                else if (targetName > currentNode->name) {  // Else if our target name is larger than the name in our root,
                                                            // we proceed down the right side of our tree.
                    if (currentNode->right == 0)
                        break;
                    currentNode = currentNode->right;       // If our right node exists, then we continue down the tree.
                }
            }
        }
        cout << endl << targetName << " does not exist in the tree.";
    }
};

int main()
{
```

```
    int inputWeight;                                      // Stores user's weight here(temp)
    string inputName;                                     // Stores user's name here(temp)
    BinaryTree binTree;

    for (int x = 1; x < 16; x++) {                        // takes in 15 users
        cout << "Please enter user" << x << "'s name: ";
        getline(cin, inputName);
        cout << "\nPlease enter user" << x << "'s weight: ";
        cin >> inputWeight;
        cin.ignore();
        binTree.addNode(inputWeight, inputName);
        cout << "\n\n";
    }

    binTree.Characteristics();
    cout << "\nPreOrder:  | ";
    binTree.PrintRVL(binTree.root);
    cout << "\nInOrder:   | ";
    binTree.PrintLVR(binTree.root);
    cout << "\nPostOrder: | ";
    binTree.PrintLRV(binTree.root);
}
/*//------------------------------------------------------------------------------- case 1:
NOTE>>
Did not use input, instead read data from a seperate function. Hence why no user input is shown for both inputting into the tree
and searching for names.

Characteristics - Height: 6, Leaves: 5, Lightest: Patrick @ 23, First Name: Brandon

PreOrder:  | Mike | Brianna | Brandon | Karl | Chuck | Jack | Finqua | Jill | Jacob | Stephanie | Roger | Patrick | Mof | Parsna | Zelda |
InOrder:   | Brandon | Brianna | Chuck | Finqua | Jack | Jacob | Jill | Karl | Mike | Mof | Parsna | Patrick | Roger | Stephanie | Zelda |
PostOrder: | Brianna | Brandon | Karl | Chuck | Jack | Finqua | Jill | Jacob | Stephanie | Roger | Patrick | Mof | Parsna | Zelda | Mike |

Finqua - 103 | exists in tree.
Chuck - 145 | exists in tree.
test does not exist in the tree.
*///------------------------------------------------------------------------------ case 1:
```