Lab 16 - Binary Trees

Start Assignment

Due Monday by 11:59pm **Points** 20 **Submitting** a file upload
Available until Dec 16 at 11:59pm

Advanced C++ Programming

Module 16 – Binary Trees

Height and Diameter

(20 points)

Perform this lab individually

Summary

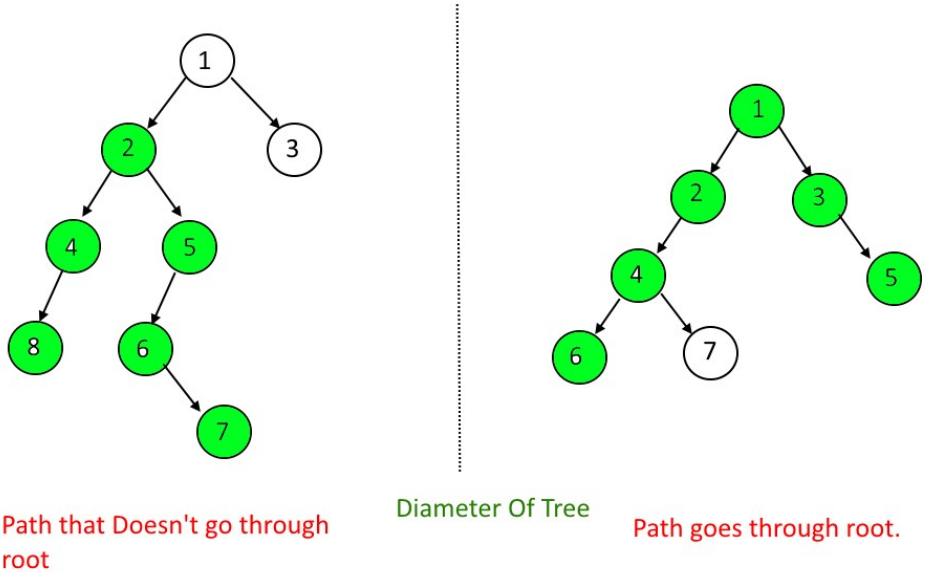
Determine the height and diameter of several binary trees using recursive functions.

Binary Trees

All binary trees need lots of care and feeding to maintain their optimal shapes. In this lab, we'll look at two measures of binary trees, *height* and *diameter*.

The *height* of a node is the number of edges from the node to the deepest leaf. The height of a *tree* is the *height* of the root of that tree.

The *diameter* of a binary tree is the longest path or route between any two nodes in that tree. The path may or may not go through the root. Consider the following diagrams:



Notice that neither tree above is a search tree. Why?

The *diameter* of each tree above is 5 (remember to count edges, not nodes). The *diameter* of any tree can be found by computing the maximum of:

1. The diameter of the left subtree
2. The diameter of the right subtree
3. The longest path (edges) between two nodes which pass through the root

We can compute the first two numbers using recursive techniques.

The longest path between two nodes which pass through the root can be found by calculating:

1 + height of left subtree + height of right subtree.

To perform this lab, download the **BinaryTree.h (https://miracosta.instructure.com/courses/31330/files/7025776?wrap=1)** **(https://miracosta.instructure.com/courses/31330/files/7025776/download?download_frd=1)** file (also available at the bottom of this lab). Rename your class **EBinaryTree** (in the **EBinaryTree.h** file) as you will be making several enhancements to this class.

Next, in the **EBinaryTree** class create a *private* function called **getHeight**. This function will take a node to **EBinaryTree** as a parameter and return the height of the tree starting at that node. Add a prototype for this function inside the **EBinaryTree** class:

```
int getHeight(TreeNode *) ;
```

It's definition (outside the class) should similar to the following:

```
int EBinaryTree::getHeight(TreeNode *tree) {
```

```
    // If at a leaf, then stop recursion
    if (tree == nullptr ||
          (tree->left == nullptr && tree->right == nullptr)) {
       return 0 ;
    }

    return 1 + max(getHeight(tree->left),
                   getHeight(tree->right)) ;
}
```

This function has a multiple recursive nature. (What is its base case? How many times does it call itself? How does this relate to the tree?)

We'll also require a *private* function in the **EBinaryTree** class called **getDiameter** which takes a node of an **EBinaryTree** and returns the diameter of that tree. Again add a prototype inside the **EBinaryTree** class:

```
int getDiameter(TreeNode *) const ;
```

Its implementation should look similar to the following:

```
int EBinaryTree<T>::getDiameter(TreeNode *tree) const {

   // If at a leaf, then stop recursion
   if (tree == nullptr) {
      return 0 ;
   }

   // Calculate the height of the left and right subtrees
   int left_height = getHeight(tree->left) ;
   int right_height = getHeight(tree->right) ;

   // Calculate the diameter of the left and right subtrees
   int left_diameter = getDiameter(tree->left) ;
   int right_diameter = getDiameter(tree->right) ;

  // Find the maximum of the left subtree diameter, the
  // right subtree diameter, and the longest path which
  // goes up to or through the root of this (sub)tree
   int edges = (tree->left != nullptr) + (tree->right != nullptr) ;
   return max(left_height + right_height + edges,
          max(left_diameter, right_diameter)) ;
}
```

Again a multiple recursive function.

To use these functions, add two inline *public* versions. These functions in turn call their *private* counterparts starting at the root of the binary tree. (These are provided because the variable **root** and all nodes are **private**.)

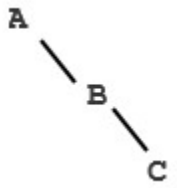
```
int getDiameter() const {
   return getDiameter(root) ;
}
int getHeight() const {
   return getHeight(root) ;
}
```

Binary Search Tree Structure

A binary *search* tree is a binary tree with the following additional properties:

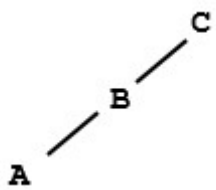
1. All values in the left subtree must be “less than” the value in the root node
2. All values in the right subtree must be “greater than” or equal to the value in the root node
3. This rule is applied recursively to each subtree
4. The base case for the recursion is an empty tree

The structure of a binary search tree depends on the order in which its nodes are entered into the tree, because all new nodes are entered as leaves of the tree. For example, a three-node tree where the nodes are entered in the order “A”, “B”, and “C” will have a structure:



First, “A” is entered, so it becomes the root of the tree. Then “B” is entered. "B" comes after “A”, so it becomes the node on the right side of “A”. Finally, “C” is entered. “C” comes after “A”, so it will appear on the right side of the tree’s node “A”, and it comes after “B”, so it will be placed on the right side of “B”.

What if we enter the nodes in the order “C”, “B”, and finally “A”? Then the tree will have a structure:



First, “C” is entered, so it is now the root of the tree. Then “B” is entered. "B" comes before “C”, so it is entered on the left side of “C”. Finally “A” is entered. “A” comes before “C”, so it will appear on the left side of the tree’s node “C”, and it comes before “B”, so it will be placed on the left side of “B”.

If we traverse either tree in *inorder* sequence, the letters are printed in lexicographic sequence “A”, “B”, “C”. In fact, no matter what order nodes are entered into a binary search tree, the tree's nodes will always appear sequentially ordered when printed *inorder*.

Test Your Functions

Build five binary search trees and determine their heights and diameters.

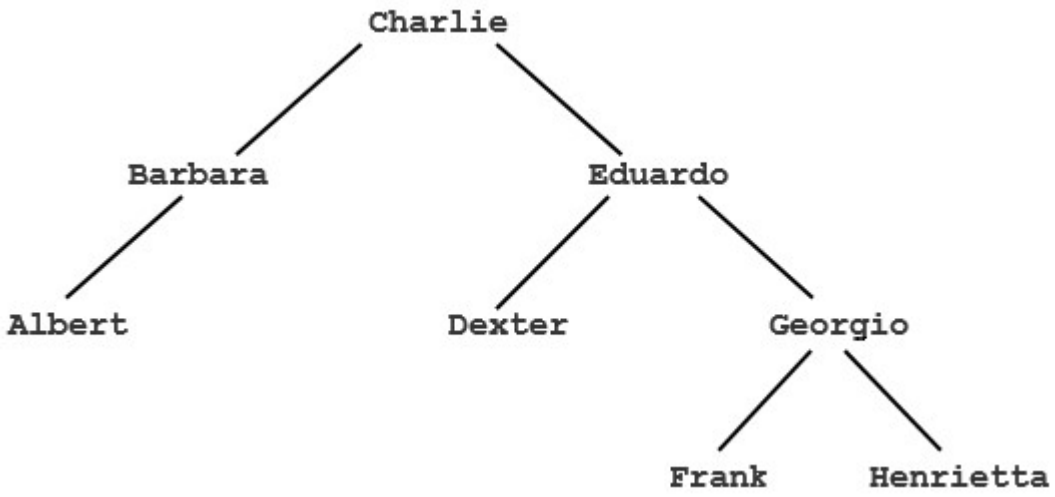
Tree 1: A tree of strings (names) where the nodes are entered in alphabetic (lexicographic) order:

Enter these names into a binary search tree in this order: **Albert, Barbara, Charlie, Dexter, Eduardo, Frank, Georgio, and Henrietta.**

Tree 2: A tree of names where the nodes are entered in reverse alphabetic order:

Enter these names into a binary search tree in this order: **Henrietta, Georgio, Frank, Eduardo, Dexter, Charlie, Barbara, Albert**

Tree 3: The tree of names listed above entered so that resulting tree looks like the following:



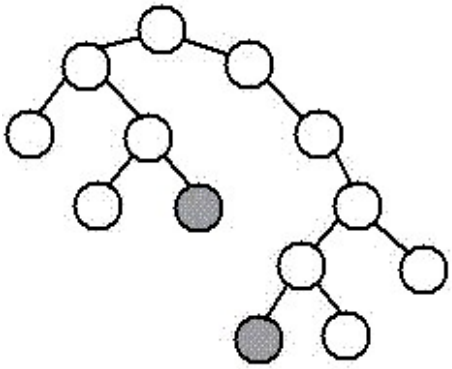
Notice the order of the tree when traversed in the *inorder* sequence – it’s still in lexicographic order!

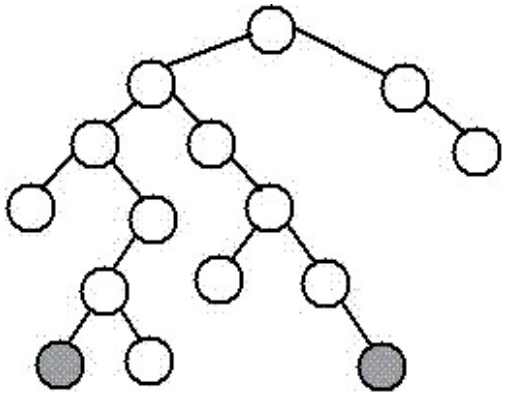
- Process the left subtree (recursively)
- Process the root node
- Process the right subtree (recursively)

To create a tree with the structure above, enter the nodes level-by-level, entering the name in Level 1 first, then the names in level 2, then those in Level 3, and finally the names in Level 4.

Level 1: **Charlie**
Level 2: **Barbara, Eduardo**
Level 3: **Albert, Dexter, Georgio**
Level 4: **Frank, Henrietta**

***Trees 4 and 5*:** Develop two binary search trees of your own, one which duplicates the structure on the left and one duplicating the structure on the right:





Both trees above have a diameter of 8 and a height of 5. The diameter of the tree on the left goes through root, while diameter of the tree on the right does *not* go through root.

To do this, first you must determine the values to assign to each node so that the tree can be used as a binary search tree. Here's a suggestion:

1. Count the total number of nodes in the tree
2. Starting with the letter "A", write down letters in alphabetic order on a piece of paper until you have as many letters as there are nodes in the tree
3. Find the root node in the tree and count the number of nodes to its left and to its right
4. Find the letter in the list of letters with the same number of letters to its left and to its right. Assign that letter to the root node of the tree
5. Now look at the node on the left side of the root node. Count the number of nodes below it which are to its left and to its right
6. From the letters written down starting at the letter "A", find the letter with that many letters to its left. Assign that letter to the node on the left side of the root node. Make sure that there as many letters to the right of the letter just assigned to the left node as there are number of nodes to its right.
7. Look at the node on the right side of the root node. Find out how many nodes are to its left and to its right.
8. From the letters written down starting at the letter immediately after the one assigned to the

root node, find the one that the same number of letters to its left and the same number of letters to its right. Assign that letter to the node on the right side of the root node.

9. Continue dividing the available letters as you sequence down each level in the tree.
10. Once you have all of the letters assigned to nodes, verify that the tree can be used as a binary tree by printing the nodes in *inorder* sequence.
11. Pick names for each letter such as “Albert” for “A”, “Bill” for “B”, etc., then enter these names into a tree level-by-level.

When finished with this lab, submit copies of all program files and snips or printscreens of each of the five trees displayed in *preorder, inorder,* and *postorder* sequence, its height, and its diameter.

Links

Additional Files and Programs

BinaryTree.h (https://miracosta.instructure.com/courses/31330/files/7025777?wrap=1) **(https://miracosta.instructure.com/courses/31330/files/7025777/download?download_frd=1)**

Next Lab

none

Homework Assignment

none

Prior Lab

Lab 15 - Sets (https://miracosta.instructure.com/courses/31330/assignments/842813)