PROGRAM 2: BST

In this PROGRAM, you will be implementing a binary search tree. You must use a node-based implementation of a tree. Each node in the tree will be represented by a Node class and should have a left and right subtree pointer. Each node in your tree should hold a string (note: a string can hold more than one word) and also contain an integer count. Include the following functions for the tree. For each operation, I have included the function prototype that you must use so that your tree will interface with the main test file. This is by no means an exhaustive list of the functions that you will be writing. These functions must be public functions in your tree class.

Required Public Member Functions

- `void insert(const string &)`: Insert an item into the binary search tree. Be sure to keep the binary search tree properties. When an item is first inserted into the tree the count should be set to 1. When adding a duplicate string (case sensitive), rather than adding another node, the count variable should just be incremented.
- `bool search(const string &) const`: Search for a string in the binary search tree. It should return true if the string is in the tree, and false otherwise.
- `string largest( ) const`: Find and return the largest value in the tree. Return an empty string if the tree is empty.
- `string smallest( ) const`: Find and return the smallest value in the tree. Return an empty string if the tree is empty.
- `int height(const string &) const`: Compute and return the height of a particular string in the tree. The height of a leaf node is 0 (count the number of edges on the longest path). Return -1 if the string does not exist.
- `void remove(const string &)`: Remove a specified string from the tree. Be sure to maintain all binary search tree properties. If removing a node with a count greater than 1, just decrement the count, otherwise, if the count is simply 1, remove the node. You MUST follow the remove algorithm shown in the slides and discussed in class or else your program will not pass the test functions. When removing, if removing a leaf node, simply remove the leaf. Otherwise, if the node to remove has a left child, replace the node to remove with the largest string value that is smaller

than the current string to remove (i.e. find the largest value in the left subtree of the node to remove). If the node has no left child, replace the node to remove with the smallest value larger than the current string to remove (i.e. find the smallest value in the right subtree of the node to remove).

Printing the tree

Print the tree in the following manners. When printing a node, print the string followed by the count in parentheses followed by a , and one space. You must follow these guidelines exactly. For example: goodbye(1), Hello World(3),

- `void preOrder( )`: Traverse and print the tree in preorder notation following the printing guidelines specified above.
- `void inOrder( )`:Traverse and print the tree in inorder notation following the printing guidelines specified above.
- `void postOrder( )`: Traverse and print the tree in postorder notation following the printing guidelines specified above.

Note about recursion

Some of the above functions used to interface with the main test file are not conducive for recursive methodology. However, you must write the inOrder, preOrder, postOrder, search, and remove functions recursively (you will lose points if you do not do these recursively). This may require you to overload 1 or more of the functions. For instance, preOrder is called from main but is not passed any parameter (you should not be able to pass the root from main because it should be a private variable of your tree class). However, you can overload the preOrder function so that it will operate recursively. For example, your preOrder function should look like this:

```
void Tree::preOrder( ) const {
    preOrder(root);
}
```

and you will write the preOrder code within the recursive function that takes a node as a parameter. You may need to do something similar for the inOrder, postOrder, search, and/or remove functions so that you can write these functions recursively. Should these recursive helper functions be public or private?

Provided Files

You can access a skeleton of the BSTree.h plus a very simple test harness (main.cpp) file in my Google drive

You will need to add other private helper functions to this BSTree class and you will need to make your own Node class.

Suggestions to follow while coding

Follow a modular programming style. Write one function for your program and completely test it before moving to the next function. This will isolate your debugging because most of the time the bug will be in the newly created module. However, if you miss or forget a test case in an old module and do not completely test it, a new module can reveal old bugs. Watch out for this.