Assignment 8: Binary Search Tree

Description: In this assignment you will create a Binary Search Tree class working with the same struct Data and struct Node from the previous assignments (int for id and string for 'data').

Background

- Watch the recorded lectures on recursion, BST part 1, BST part 2, and this assignment.
- Study the notes on Blackboard regarding recursion.
- Study the notes posted on Blackboard regarding Binary Search Trees. It is in chapter 15 of your text and the notes are on Blackboard. There are also notes in chapters 16 and 17 but we will not cover those explicitly (you may want to read them for your own knowledge).
- Study the code at: https://github.com/alexander-katrompas/binary-search-tree-simple-example to help you understand Binary Search Trees.

Specifications / Requirements

- Follow the Assignment Specific Instructions using this GitHub assignment invite https://classroom.github.com/a/6d-35HuV
- You are given all the files you need for the assignment except .gitignore.
    - Make a proper gitignore and commit it.
    - Place your comment headers in the cpp and h for the tree class and commit.
    - main.cpp and .h are given to you, **do not** modify them. These files already contain test data and a complete and working test suite.
    - You may add functions to functions.h/cpp if you like, but you shouldn't need to.
- Create a Binary Search Tree class as discussed in class and in your text. The class will contain all the data and methods to have a complete working and proper BST object.
- The class must be completely self contained (loosely coupled) and fully functional.
- The tree object may not print to the console or contain any other I/O except parameters and data passed in, and return values passed out, *except* for the traverse functions. Those must print so you can demonstrate traverse.
- Use a "linked list" approach where the tree is a collection of Node pointers (left and right pointers). i.e. The tree is an ordered "list" of pointers to structs.
- Your tree has to be capable of growing to any size.
- Create and delete your nodes inside the tree (just as in previous structures).
- Your class will **only** have the following attributes and should be set to null and 0 in the constructor:
    - **DataNode *root;**
    - **int count;**

- Your Tree must have the following public methods:
    - **BinTree();** your constructor for initializing root to null and count to 0
    - **~BinTree();** your destructor, it must call clearTree()
    - **bool isEmpty();** test for is empty, return T/F
    - **int getCount();** return count
    - **bool getRootData(Data*);** pass (by reference) an "empty" Data struct from main() and fill it with root's data if the tree is not empty, otherwise place -1 and an empty string in the struct. Return T/F based on if data was retrieved.
    - **void displayTree();** display all stats for the tree as shown in the examples and call all display order methods.

    All of the following *public* methods will also have private overloads to implement recursion because you cannot give access to the root pointer outside the tree (see below for the private overloads).

    - **void clear();** deallocate the tree and set it back to "empty."
    - **bool addNode(int, const string*);** pass in and return the same as in previous structures from previous assignments. Don't worry about duplicate ids, just add them anyway.
    - **bool removeNode(int);** pass in and return the same as previous structures
    - **bool getNode(Data*, int);** pass in and return the same as previous structures. You must use binary search search to retrieve the Node.
    - **bool contains(int);** pass in and return the same as previous structures
    - **int getHeight();** dynamically calculate the height of the tree (do not store height, calculate it each time getHeight() is called.
    - **void displayPreOrder();** do a pre-order traversal, printing as you go
    - **void displayPostOrder();** do a post-order traversal, printing as you go
    - **void displayInOrder();** do a ijn-order traversal, printing as you go
- For all of the above where it indicates they also need private overloads, you will need the following private methods (these are all overloads of the public methods):
    - **void clear(DataNode*);**
    - **bool addNode(DataNode*, DataNode**);**
    - **DataNode* removeNode(int, DataNode*);**
    - **bool getNode(Data*, int, DataNode*);**
    - **bool contains(int, DataNode*);**
    - **int getHeight(DataNode*);**
    - **void displayPreOrder(DataNode*);**
    - **void displayPostOrder(DataNode*);**
    - **void displayInOrder(DataNode*);**

    The reason for the private overloads is to be able to recurse, you need to pass the root (new root) back into the function over and over. All the overloaded functions above simply contain an extra DataNode for that purpose. The public function is called by the external caller, and then the public function calls the private version with the root. The private version then recurses until the process completes. This technique is demonstrated in the sample code (link above).

Grading: Your grade will be graded primarily on exactness to detail and specifications, architecture, and coding logic. You will also be graded on your repo and testing with the following guidelines:

- Failure to commit often, small, and smart will result in an **automatic -10% penalty** regardless of your code quality.
- Any stray files in your repo will result in an **automatic -10% penalty** regardless of your code quality.
- If you do not have proper comment headers and/or do not use the Write Submission feature (not the comment section), and/or do not submit your link correctly, it's an **automatic -10%** penalty regardless of your code quality.
- Failure to use the correct branch in your repo (main) will result in an **automatic -5% penalty** regardless of your code quality.

Assignments that do not compile for any reason will not receive a grade above 50%

Submission: When you are ready for grading, use the write submission feature in Blackboard and submit your repo link (this is the URL in the browser, not your SSH or .git link). If you need to fix/change something after you submit but *before* I grade it, just fix/change it and push again. **Do not re-submit the assignment before getting a grade**. Only re-submit after you get a grade and want a *re*-grading.

Hint: You should stub all methods and functions first to get the starter code to compile. You may also want to comment out most of main() and uncomment things one part at a time as you implement features. If you do that, make a back-up of the original main.cpp so you can put it back in place without forgetting something.

Output: Your output will be **exactly** the same as the following. If it is not exactly the same there will be a substantial grade penalty, up to and including a zero.

```
Binary Search Tree created

DISPLAY TREE
==============================================
Tree is empty
Height 0
Node count: 0
```

Pre-Order Traversal

In-Order Traversal

Post-Order Traversal
==

Testing removeNode() on empty tree
==
removing 10... failed

Testing getRootData() on empty tree
==
NOT retrieved -1

Testing contains() and getNode() on empty tree
==
does NOT contain 57
NOT found: 57

Filling Tree
==
adding 60...added
the height of the tree is 1

adding 20...added
the height of the tree is 2

adding 70...added
the height of the tree is 2

adding 40...added
the height of the tree is 3

adding 10...added
the height of the tree is 3

adding 50...added
the height of the tree is 4

adding 30...added
the height of the tree is 4

DISPLAY TREE
===
Tree is NOT empty
Height 4
Node count: 7

Pre-Order Traversal
60 sixty
20 twenty
10 ten
40 forty
30 thirty
50 fifty
70 seventy

In-Order Traversal
10 ten
20 twenty
30 thirty
40 forty
50 fifty
60 sixty
70 seventy

Post-Order Traversal
10 ten
30 thirty
50 fifty
40 forty
20 twenty
70 seventy
60 sixty
===

Testing getRootData() on non-empty tree
===
retrieved 60 sixty

Testing contains() randomly
===
contains 50
contains 60

contains 20
contains 10
contains 10
contains 30
contains 60
does NOT contain 7
does NOT contain 39
does NOT contain 5059
==

Testing getNode() randomly
==
retrieved: 10 ten
retrieved: 60 sixty
retrieved: 20 twenty
NOT found: 1
NOT found: 1000

Testing removeNode() randomly
==
removing 60... removed
removing root 70... removed
removing 50... removed
removing 35... failed

DISPLAY TREE
==
Tree is NOT empty
Height 3
Node count: 4

Pre-Order Traversal
20 twenty
10 ten
40 forty
30 thirty

In-Order Traversal
10 ten
20 twenty
30 thirty
40 forty

Post-Order Traversal
10 ten
30 thirty
40 forty
20 twenty
==

adding 35... added

DISPLAY TREE
==
Tree is NOT empty
Height 4
Node count: 5

Pre-Order Traversal
20 twenty
10 ten
40 forty
30 thirty
35 thirty five

In-Order Traversal
10 ten
20 twenty
30 thirty
35 thirty five
40 forty

Post-Order Traversal
10 ten
35 thirty five
30 thirty
40 forty
20 twenty
==

Clearing tree... Cleared

DISPLAY TREE
==

Tree is empty
Height 0
Node count: 0

Pre-Order Traversal

In-Order Traversal

Post-Order Traversal
===

Filling tree with poorly chosen data
===
adding 5...added
the height of the tree is 1

adding 15...added
the height of the tree is 2

adding 25...added
the height of the tree is 3

adding 35...added
the height of the tree is 4

adding 45...added
the height of the tree is 5

adding 55...added
the height of the tree is 6

DISPLAY TREE
===
Tree is NOT empty
Height 6
Node count: 6

Pre-Order Traversal
5 five
15 fifteen
25 twenty five

35 thirty five
45 forty five
55 fifty five

In-Order Traversal
5 five
15 fifteen
25 twenty five
35 thirty five
45 forty five
55 fifty five

Post-Order Traversal
55 fifty five
45 forty five
35 thirty five
25 twenty five
15 fifteen
5 five
==