CptS 223 PA #2 - AVL Tree Implementation

For this project, we will be implementing functions of an AVL tree. The tree will act as a BST with integer keys (which are Comparable types) and must maintain the AVL nature as you insert and delete nodes. This archive includes a working AVL tree implementation, but does not do several things:

1) Do inserts into the tree with integers
    a) You'll see the vector inserts used in the test code, that's calling your code
2) Maintain the heights of nodes after an insert
3) Check heights of subtrees and do rotations as appropriate to keep the tree balanced
4) Search the tree to see if it contains a given node
5) Return the size (node count) in the tree
6) Remove a node when given a key
7) Return whether the tree is empty or not
8) Print out the tree in pre-, in-, and post- order
9) Implement the makeEmpty function for the destructor

Your assignment is to add these features to the provided code. There are several notes in AvlTree.h about where you need to fill in the stubbed functions or replace/update the ones that are there, notably in insert, remove, contains, height, pre-, in-, post- print functions. Any function that has "TODO" in the comments is one that you'll need to look at.

The starter code compiles and runs on the EECS SIG servers just fine. You can use that as a starting point for you own work. You will need to compile it using the provided Makefile. I have committed the project to your Git repository under the "PA2" branch.

This makefile has one extra target for a big fuzzing test. You'll need to do the full fuzzing test once you have both insert and remove implemented by executing 'make bigtest'. The fuzzing test will do many random inserts and ⅓ as many deletes. This kind of test is called fuzzing because it's trying many random input values to a given algorithm to see if anything breaks. In this case, we'll make a tree of up to 3000 nodes and just seeing if it breaks.

Expected Output

Your code must compile on the EECS servers, though you can do your development on other systems and do final tests on the EECS machines if you like.

Your program will be tested first by doing this series of commands:

1) make
2) make test
3) make bigtest
4) make clean

Then we’ll inspect the source code for quality. Structure, indentation, naming, commenting, cleanliness, consistency.

Deliverables

- Your program will be checked into your Git repository. The only thing uploaded to Blackboard will be the hash of the commit you’d like the grader to look at.

Grading Criteria

Your assignment will be judged by the following criteria:

- [80] Code operational success. Your code compiles, executes, and passes the tests.
- [10] Your code is well documented and generally easy to read.
- [10] Your program intelligently uses classes when appropriate and generally conforms to good OOP design (i.e. everything isn't slapped into main).