BRONX COMMUNITY COLLEGE
of the City University of New York
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

CSI 33 Section E01 **Project 6**
Fall 2017 **November 22, 2017**
Due: Wednesday, December 6, 2017

Programming Project Number 6: Implementing the AVLTree Class

This is similar to Programming Exercise 4 on page 483 of the text, except that the programming language is C++. Other special requirements as listed below.

You are to complete the AVLTree C++ class, starting with the files AVLTree.cpp and AVLTree.h provided with the C++ Tree Supplement. (It is described for Python in Chapter 13, section 13.3.) You also have the files TreeNode.cpp ancd TreeNode.h.

AVLTree.cpp implements a constructor method (`AVLTree()`) and a partially written `insertRec` method which handles rebalancing the tree in the case of inserting an item into the left or right subtrees of the left child of the current root. You must add the code to rebalance the tree after inserting an item into either subtree of the right child of the current root. This must be done whenever the heights of the left and right subtrees, as given by `get_height()`, differ by 2. (Hint: Since this is a mirror-image of the case for the left child, you can take the code for the left child and modify it by replacing each occurrence of "right" with "left" and vice-versa. You may also need to switch `<` and `>` when comparing item and node values.)

A precondition for insert is that the item value does not already exist in the tree. If it does, raise a value exception in the insert method. A precondition for delete is that the value being deleted **does** exist in the tree. If it does not, you must raise a value exception.

Use the provided test program to show that insert, delete, and find will work. This will include code to actually display the structure of the tree, to verify that it remains balanced after an insertion operation (or deletion–see below).

Finally, you must modify the `delete_` method so that it also leaves the tree balanced. That is, after the usual BST deletion, additional code must run to check the heights of the two subtrees, and to rebalance the tree at that node, by single or double rotation, if it is unbalanced. (Hint: it is the same algorithm used to rebalance the subtrees when inserting.)

Raising exceptions when preconditions fail in `_insertRec` and `_deleteRec`

Preconditions should be checked by raising exceptions when they fail. A precondition for `_insertRec` is that a value being inserted does not already exist in the tree. This happens when `value == t->_item`. A similar precondition for `_deleteRec` would be that a value being deleted must be present in the tree. If it is not, raise an `out_of_range` exception. (This should happen for the base case, when `t` is `NULL`). To see the code to do this, you can look at the BST class, or the `_findRec` method.