Lab 3

In this lab, you build on the code developed for lab 2 (a BST with nodes that know their height and methods for rotation) toward an implementation of AVL trees. Begin by completing that lab up to the implementations of `rotateLeft` and `rotateRight`.

AVL trees

The first part of implementing AVL trees is determining the necessary rotations. In the last lab, you made each node keep track of its height. Use this information to make each node also store its balance. Remember that this is its right height (i.e. its height where the first step is to the right) minus its left height (i.e. the same but taking the first step to the left). Thus, it can be calculated by subtracting 1 more than the height of the left subtree from 1 more than the height of the right subtree; a special case is needed if either subtree is null, in which case the height on that side is 0. Have the `insert_helper` method perform this update after its recursive call returns. Check that the balance is being properly calculated by print it; you can print this instead of or in addition to the height.

Now we are ready to start actually implementing balancing operations. Decisions about balancing will be made in `insert_helper` after the new balance is calculated. Begin by checking if the balance is +2. If so, check the right child’s balance. If it is +1, then perform a left rotation on the node (the one with balance +2). If it is -1, you’ll need a double rotation, but begin by just having the program print an error message. (Note that these are the only two legal values so it’s probably a good idea to have an error message printed if the child’s balance has any other value.) If a rotation is performed, you’ll need to recalculate the balance.

With only the single case of +2 balance for a node and a +1 balance for the right child, your tree should work if the inserted values are an increasing series (e.g. 1, 2, 3, 4, …). Verify this by making such a series of calls to insert, printing the tree at each step.

Once this works, add the analogous case for right rotations; a node with -2 balance whose left child has -1. Once this is done, verify that your tree stays balanced on the insertion of a decreasing series of values.

When both of those are implemented, you’ll need to handle double rotations. Implement the one for a node with balance +2 whose right child has balance -1. In this case, perform a right rotation on the right child followed by a left rotation on the original node. Verify that this works before implementing the analogous double rotation on the other side.

If you have additional time, see if you can figure out how to handle deletions.