Homework 8: Binary search trees

In this homework you will modify an implementation of binary search trees. A beginning implementation is found in the bst.py file. This implementation is similar to the implementation we created in class, but it is not identical, and you should spend some time making sure you understand how it works.

In the starting implementation, the shape of the tree can be anything, depending on the order of inserts and deletes. However, this can make operations on the tree take a long time. In particular, if elements are inserted in sorted order the tree will be equivalent to a linked list and very inefficient.

We can fix this problem by periodically *balancing* the tree. That is, if at any node the left and right subtrees are substantially uneven in size, we run a balancing operation to fix the problem. In this homework you will add this balancing feature to our implementation of binary search trees.

Step 1: adding a height variable to nodes

We're going to need to keep track of two new variables at each node. The first variable is the height of the node. The height is defined as the distance between the node and the farthest leaf in its subtree. That is, leaves have a height of zero, their parents have a height of one, and so forth. Each node should store its own height, and all operations on the tree need to be modified so that heights are kept accurate. You should also make sure that the worst-case time it takes for each operation remains proportional to the height of the tree.

Step 2: adding a size variable to nodes

We will also need a size variable in each node. This stores the number of nodes in the subtree rooted at that node (including the root node itself). As with height, all operations need to be modified so that size variables stay accurate as the tree is changed.

Step 3: balancing

Now for the hardest part – actually balancing part of the tree. You should add a `balance` method that takes as input a particular node and balances the subtree rooted at that node.

The balancing operation has two parts. The first thing that happens is that the subtree being balanced is converted to a sorted list. Because this data is already in a binary search tree, you should be able to construct a sorted list in $\theta(t)$ time, where t is the size of the subtree being balanced.

Once that has been done, the subtree is rebuilt. This can be done quickly using recursion. Simply make the middle element of the list the new root, then recursively build the left and right subtrees of the root from the two halves of the list. This should also take $\theta(t)$ time.

Step 4: calling on the balancing

Balancing isn't something that users need to do. It should happen automatically when the tree becomes too unbalanced. Modify the tree's operations so that when items are being inserted or deleted, the tree is checked for imbalance at the same time. As the tree is traversed to find where a given value should be inserted or deleted, it should check whether the subtrees rooted at each of the nodes being passed need to be rebalanced, and if so they balance method should be called to do that rebalancing.

We are going to say that a tree is sufficiently unbalanced to need rebalancing if the subtree of one of its children is at least twice the size of the other. We are also going to say that subtrees of height 3 or less aren't big enough to be worth rebalancing. You should check for these conditions and rebalance trees that meet them.

Two things to note: First, it is ok if you're going to insert something into a subtree if you rebalance it first and then insert. (It might feel bad to "mess it up" right away, but it's ok and in some cases even better to do it that way.) Second, if a subtree rooted at x needs to be rebalanced, but also one of its subtrees rooted at y needs to be rebalanced (so x is an ancestor of y), you should make sure that you rebalance at x. The tree at y will be reworked as part of the rebalancing at x, so rebalancing at y first would be a waste of time.

Step 4: finishing

Remember to test what you have done. That will take some effort on your part, but it is worth doing it. When you are sure you are done, upload the modified bst.py file in a new folder in your class directory on AFS.