CS 361 – Lab #4 – AVL Trees

An AVL tree is a binary search tree that tries to keep the tree balanced. For each node in the tree, its children may not differ in height by more than 1. Earlier in class you saw how we maintain this balance in our insert and delete operations. In today's lab, you will be given a simple binary search tree implementation. Modify it as follows so that it will behave more like an AVL tree. Because fancy tree functionality can be tedious to implement, today we will just focus our attention on the *insertion* of key values.

1. Create a new directory called lab04. Copy my code from here: http://cs.furman.edu/~chealy/cs361/lab04/. You will be making extensive changes to Tree.java to support added functionality, and only token changes to Driver.java for testing.

   Before making changes to the program, you may want to run it now to see what it does. I have included 2 versions of a tree toString. The first is "verbose" and prints everything about each node. A second one gives a concise format. You might want to see the verbose output to confirm the entire contents of the tree, including null pointers.

2. In Tree.java, add a new instance method called findHeight(). There is already a short comment in the code showing you where to write this function. It will take a Node parameter, and return this node's height in the tree. This calculation will become useful for us later when we implement the AVL insertion algorithm.

3. After you get findHeight() working, it would be good to test it on the entire tree. Under findHeight, implement a new function called findTreeHeight() that will use findHeight() to determine the height of the entire tree.

   Once you have it implemented, go to Driver.java, and un-comment the line of code that tests this method. Driver.java creates an array of key values to be inserted in the tree. I encourage you to change these values so that the program will create larger and smaller trees. This will help in testing that the heights are being calculated correctly.

4. Now comes the main course. In Tree.java, there is a method called add() which does a classic insertion into a binary search tree. Near the end of this function there is a comment indicating where you need to continue this implementation so that the insertion conforms to what should happen in an AVL tree. I have provided you some pseudocode based on our notes in class.

   A good test case would be to insert the consecutive values 1 – 16 into an empty tree.

AVL insertion is not an easy algorithm, so let me give you some hints. Here is an outline of a possible implementation:

1. Identify w as the node that was just inserted.

2. We need to find z, the first unbalanced node we encounter when going up the tree from w. (Maybe no such z exists, which is great because no additional work will be necessary.) Write a loop that starts at w and goes upward on each iteration. For each node you encounter in this loop, you need to check its balance. This means determining the heights of its two children. You will need to call the findHeight() method you wrote earlier. Do you remember how much of a height difference is allowed for the tree to be considered balanced? Note that if the heights of the two children differ, you don't know in advance which child is taller: you must plan for both possibilities. If you find that this node is unbalanced, call it z, and exit the loop.

3. There is also the case to consider where an unbalanced node may have only one child.

4. If z exists, then you must restructure the tree so it is still an AVL tree. Next, you would determine nodes y and x, as mentioned in the pseudocode algorithm. Again, you will need to call the findHeight() method. You might also find it handy to write a new method called isAncestor(). Do you see in the pseudocode where this might be useful?

5. You might want to print out some diagnostic output from time to time. For instance, it may be helpful to know the key values at nodes x, y and z.

6. To do a restructuring, you need to create 7 new nodes as per the algorithm: a, b, c, t1, t2, t3, t4. You need to write some logic (if-statements) that determine the numerical relationships among the numbers in x, y and z so that you can determine how to initialize the new nodes. Note that b is the node that will contain the median value in x, y, z, and it will become the root of the subtree. Note that it is quite possible for some of t1-t4 to be null: how would you determine which ones should be null?

7. Next, you need to carefully set pointers within the tree. You need to carefully consider how to set the left, right and parent pointers for the new nodes that you created in the previous step.

8. Finally, the node b, which is the root of the subtree, needs to become "attached" to the rest of the tree. "b" needs to know where its parent is. But might there also be a case where b should not have a parent? Also, the rest of the tree needs to point to the tree. For example, somebody's child is probably going to be b.