Balanced Search Trees (AVL)
Summer 2015

Goal: To implement a balanced search tree using the AVL algorithm.

General:

Binary search trees (BST) are very useful in maintaining dynamic sorted data. If we knew all the data ahead of time, we could put it in an array, sort it once, and then find things efficiently using binary search in O(log n) time. If we don't know all the data ahead of time, then we need a way to dynamically add new data, yet not have to sort it each time we add something.

BST's provide a way to do this, by basically encoding the binary search algorithm into their structure. The problem we run into though, is with a plain BST, if the data is already sorted it becomes a O(n) linked list. To combat this, we need a way to "balance" the tree.

There are many algorithms out there that you may have heard of like: Red-Black, Splay, AVL, Treaps, B-trees, 2-4 trees, etc. The one we will implement for this homework is the AVL tree. AVL trees are one of the earliest of the balanced structures and they maintain a strict height-balanced tree. The cost of this though is that we have to compute some information upon insertion or deletion.

This information is called the balance factor, and remember from class that it is defined as the left subtree height - right subtree height. The only valid balance factors are -1, 0, and 1. Any other value means the tree is out of balance. We restore the balance through rotations. There are four basic rotations: left, right, left-right and right-left. In class we discussed how to detect which of these you need based on the sign changes of the balance factors.

Implementation:

Your homework is to code an AVL tree. You may not refer to AVL implementations on the net, however you may refer to the provided pseudo-code and the code we wrote in class for a basic tree. You may also reuse any code that you wrote for HW3. You will implement the BSTree interface that is provided. I have also included some tests in AVLTreeTest.java.

You will create AVLTree.java that defines a class AVLTree that implements the provided BSTree interface. You may create as many private helper functions as you like (I had a lot to support rotations and recursive traversals).

As in HW3, for removal, use the in-order successor.

You are allowed to use the List classes (ArrayList and LinkedList as well as the List and Queue interfaces) to support your traversals and iterator as in homework 3. You may not however use any API classes to implement the tree itself.

Your AVLNode class should be an inner class of the AVLTree.

If you choose to write your own iterator from scratch rather than use the ones in List, then that class would be an inner class also.

Provided Files:
BSTree.java the interface for your project
AVLTreeTest.java partial junit tests for the public methods of the interface

Required Submission:
AVLTree.java your implementation of the interface with any required inner classes.

Grading Notes:

1. Non-compiling code will receive a zero.
2. Points will be awarded per the Criteria below.
3. Point deductions will be made for not meeting correct running times.
4. Use of Java Collections code is only allowed in the traversals/iterators.

Criteria:

isEmpty ……02
size …….. 02
add
   a. single left rotate……10
   b. single right rotate…….10
   c. double LR……..10
   d. double RL……..10
   e. multiple rotations……..10
   f. no rotate ……05
max ……..03
min……03
contains……05
remove
   a. simple no rotate…….05
   b. single rotate……10
   c. multiple rotates……..10
iterator……..02
getPostOrder……..02
getPreOrder……02
getLevelOrder……02
clear……..02