CS-260	Lab #4: AVL Tree (Spellchecker)

General

For this lab you will further your studies of Abstract Data Types and by coding a pointer-based AVL tree ADT. The basic assignment for this lab is to implement an AVL tree based version of the spell checker. You are to do this with as minimal changes as possible to the existing code by replacing the use of the "TestList" class with your own implementation pointer based AVL tree class. This can be done with only changing the changing the dictionary field from "TestList" to your container class.

As with all of my labs, you can enhance this in any way (maybe add a GUI interface) for extra credit.

Turn in you complete source, project files, directory structure, etc. of you ECLIPSE project directory (use the EXPORT tool in Eclipse to export the entire project directory). This zipped file should be turned in using the same method and done for previous labs.

General lab requirements:

At the beginning of the .java files your comments should include your name, creation date, Project #4, and purpose of this .java file.

Each public class and method should have a JavaDoc formatted comment specifying at least what the method will do. These comments can be fairly short at this point (one sentence). Methods that are implementations of interfaces that are already documented only need the @Override tag as they will get the JavaDoc from their interface definition. You should additionally have inline comments at any point in the code where the complexity is not easily obvious from the code itself.

Your indentation should be consistent. You should indent once for every pair of curly brackets, and reduce the amount of indent used after each closing curly bracket. Use the Eclipse code formatter to clean up your code!

You should use EXCEPTION HANDLING where ever it makes sense to do so.

Concepts

This lab will introduce you to Java programming techniques for a AVL TREE ADT and gives you additional experience with parts of the Java Class Hierarchy. **Take a look at the Java TreeSet class. You are to write a new class that behaves similarly to this class (we won't do the Ordered Set, just the Set).**

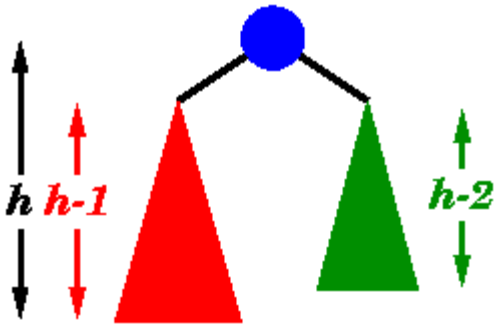
Background

An **AVL tree** is one of several methods for building balanced binary search trees. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(log n)*** search time. Addition and deletion operations also take ***O(log n)*** time. These are known as height-balanced trees.

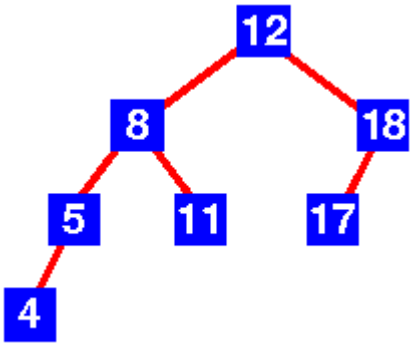
Definition of an AVL tree
An AVL tree is a binary search tree which has the following properties:

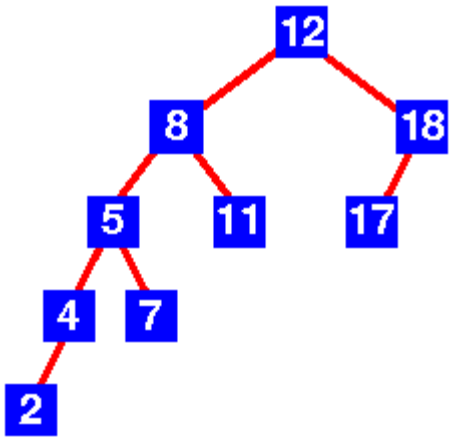
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.



You need to be careful with this definition: it permits some apparently unbalanced trees! The value is the insert and search complexity, not whether each side is equally weighted. For example, here are some trees:

Tree	AVL tree?
	Yes Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree.

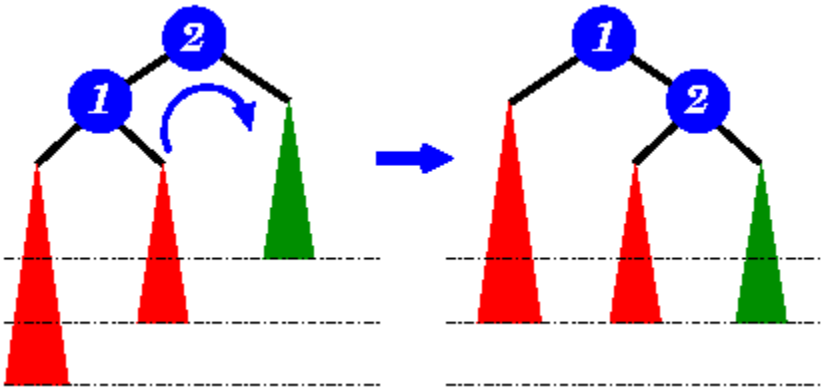


No
Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2

Insertion

As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy* (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.

A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.



Key terms

AVL trees

Trees which remain **balanced** - and thus guarantee **O(log n)** search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in **O(log n)** time.

Assignment

4a.

Write a new **AVL Balanced binary search tree** class (named "AVLTreeSet") that **inherits from your BSTreeSet** class and overrides the "add" methods (and optionally the "remove" methods). The other methods should not require any changes, so they do not need to be overridden. Since we are inheriting from the BSTreeSet class, your AVLTreeSet class will also be a "Set" implementation. However, the only method that we are overriding is add (and optionally remove). All the other methods will work via inheritance and should not be present in you AVLTreeSet class.

Further, your container must be implemented as a generic container that can accommodate any object type that extends the "Comparable" interface. Your implementation must use a pointer-based, binary-search tree ADT.

You will need to write several supporting methods to get AVL balanced "add" to work. Suggested methods are:

```
private int getHeight( Node)
private void fixHeight( Node)
private int getBalanceVal( Node)
private Node singleRotateRight( Node)
private Node singleRotateLeft( Node)
private Node doubleRotateRight( Node)
private Node doubleRotateLeft(  Node)
private Node balance( Node)
```

As long as you have a good start on all of the supporting methods, you will get full credit the #4a lab grade.

Notes:

1. Name your class "AVLTreeSet" so that it can be tested with the jUnit tests that I have provided. It must inherit from your BSTreeSet. If your BSTreeSet properly implements both "Set" and "CompareCount" interfaces, then your AVLTreeSet will inherit all of these behaviors.

2. In order to correctly build the AVLTreeSet when you are comparing the DATA OBJECTS you must use the "Comparable" interface of your data items. Use the compareTo(..) method to determine the ordering for insertion. The class declaration will be something like:

   public class AVLTreeSet<T extends Comparable<T>> extends BSTreeSet<T>

3. Your implementation of "add" and MUST be written using recursive code. Start with copying the "add" and "addHelper" methods from your BSTreeSet class into you AVLTreeSet class so that you can modify this code to do balanced adds. Your add method will call upon several private methods that you will be writing to accomplish this.

4b. Do the following:

#1. Get the AVL add method fully functional and tested. Test your AVL tree class using the jUnit tests that I provide, and also make sure that it works with the SpellChecker project files.

#2. Now that you have a pointer based AVL tree class, edit the Dictionary class and replace the use of the "TestList" class with your AVL Tree ADT. The program should produce the same results as before, except with MUCH better run times and much lower comparison counts. Further, your AVLTreeSet can handle both the sorted and unsorted dictionary, where the BSTreeSet would fail with a stack overflow on the sorted dictionary.

Submission

Turn in your complete source, project files, directory structure, etc. of your ECLIPSE project directory. Please **export** the entire project using the Eclipse export utility. This zipped file should be turned in using the same method and done for previous labs.