COMP 282 - Project 1

The purpose of this project is to introduce you to building and utilizing trees. It will consist of multiple parts, each corresponding to a lecture. You will be expected to follow the directions for submission to the letter. Failure to do so will result in you receiving no credit for the assignment.

Instructions for Submission

You must provide all source files in a single `.zip` file. This file most contain no directories whatsoever. That is, if I were to unzip it, the contents would be only the `.java` files required for the project. For example, in a *nix environment, I would create a submission by navigating to my source directory issuing a `zip AdamClark *.java`. **Under no circumstances will a submission be accepted if it is the product of you zipping your IDE's project directory!**

There are to be absolutely no packages used in your submissions. The use of the default package is, generally, not advised for enterprise-level work; this, however, is not that.

Additionally, you should not include any `main` functions in your submission, only those files required for each project part are recommended.

The following files **must** be present in your submission:

- `Tree.java`
- `BinaryTree.java`
- `AVLTree.java`

You may include any interfaces described here, but they will not be examined. A successful project need only include the list of files mentioned above – with appropriate implementations, of course.

Part 1 - The Tree Class

In order to build more complex tree-based structures, you need to start with the basics. Namely, you will need to build a `Tree` class to store some arbitrary set of elements. We will extend this class throughout the rest of the project.

The basic interface definition is as follows:

```
public interface ITree<T> {
  public T getItem();
  public ITree<T> find(T item);
  public ITree<T> insert(T item);
}
```

This should be included in your project as `ITree.java`. You must provide an implementation for this interface in form of a `Tree` class – to be defined in `Tree.java`:

```
public class Tree<T> implements ITree<T> {
  // ...
  public Tree(T item) {
    // ...
  }
  // ...
}
```

This is the only file required from this part of the project.

Part 2 - Binary Search Trees

Now it's time to take the general implementation of a tree, and extend from it a binary search tree. To do this, we will need to be able to only accept items that are ordinal:

```
public class BinaryTree<T extends Comparable<T>> extends Tree<Comparable<T>> {
  // ...
}
```

You will want to override the `find` and `insert` methods of your `Tree` class in order to ensure you are using this new tree as efficiently as possible. The `BinaryTree.java` file will be the only required file from this part of the project.

Part 3 - Traversal

In this part, we will supply four methods for traversing our tree structures. The goal is to implement the following interface in your `BinaryTree` class:

```
import java.util.*;

public interface ITraversable<T> {
  public ArrayList<T> nlr(); // Pre-order
  public ArrayList<T> lnr(); // In-order
  public ArrayList<T> lrn(); // Post-order
  public ArrayList<T> bfs(); // Breadth-first
}
```

Each method should return an `ArrayList` of node values, based on the appropriate traversal.

Part 4 - Measurement

In order to implement some more sophisticated trees, we will need an easy way of determining their heights. In order for to do this, we will implement another interface:

```
public interface IMeasurable {
  public int size();
  public int height();
}
```

These should be fairly self descriptive: the `size` method will return the total number of elements in the tree, while the `height` method returns its height. **Remember: the height of a tree is the longest path from the root to any of its leaves.**

Part 5 - Rotation

We have seen where the use of binary search trees can go wrong if we aren't careful about their inputs. In order to be more robust against these bad situations, we will implement left and right rotations in the `BinaryTree` class. In order to accomplish this, you will have to implement a new interface:

```
public interface IRotatable<T> {
  public ITree<T> rotateLeft();
  public ITree<T> rotateRight();
}
```

Each function will operate on the root node of its tree, and perform the appropriate rotation. The return value should be the new root of the rotated tree.

For example:

```
BinaryTree<Integer> t = new BinaryTree<Integer>(1);
```

```
t.insert(2);
t.insert(3);
ITree<Integer> rotated = t.rotateLeft();
rotated.getItem(); // == 2
```

There is no expectation that the old tree be maintained. That is, you are free to change any references you wish without first making a copy.

Part 6 - AVL Trees

Finally, your task will be to implement an `AVLTree` class which derives from your `BinaryTree` class. Your AVL Tree should maintain a balance factor during item insertion, and use it to rebalance the tree automatically whenever the balance factor is determined to be ≤ -1 or ≥ 1. You should also maintain a reference to your tree's parent inside the class:

```
public class AVLTree<T extends Comparable<T>> extends BinaryTree<T> {
  private int balance;
  private AVLTree<T> parent;

  public AVLTree(T item) {
    this(item, null);
  }

  public AVLTree(T item, AVLTree<T> parent) {
    super(item);
    this.balance = 0;
    this.parent = parent;
  }
}
```

In order to properly extend your `BinaryTree` class, you will have to override its `insert`, `rotateLeft`, and `rotateRight` functions. This will be in order to maintain a proper balance factor, for retracing, and for maintaining a proper reference to each subtree's parent during each of these operations. You will have to submit your `AVLTree.java` file for maximum credit. **Note: this is the most complex portion of this project, and is worth twice as many points as any other section toward the project grade.**