## Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines you risk receiving a **0** for the entire assignment.

1. All submitted code must compile under **JDK 8**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, or method signatures.
4. Do not add additional public methods.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use `java.util.LinkedList` for a Linked List assignment. Ask if you are unsure.)
6. Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered non-efficient unless that is absolutely required (and that case is extremely rare).
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. After you submit your files redownload them and run them to make sure they are what you intended to submit. You are responsible if you submit the wrong files.

## AVL Tree

You are to code an AVL Tree. An AVL Tree is a self-balancing Binary Search Tree - a data structure consisting of nodes, each having a data item and a reference pointing to the left and right child nodes. For any given node, its value is greater than any of the nodes in its left sub-tree and smaller than any of the nodes in its right sub-tree. You must implement appropriate rotations to ensure that the tree is always balanced.

For this assignment, any attempts to add data already present in the tree should be ignored. All elements added to the tree must implement Java's generic `Comparable` interface. All methods in the AVL tree that are not O(1) must be implemented recursively.

Your AVL tree implementation will implement the AVL interface provided. It will have two constructors: the no-argument constructor (which should initialize an empty tree), and constructor that takes in data to be added to the tree, and initializes the tree with this data.

### Nodes

A class `AVLNode` is provided to you; do not modify it. The AVL consists of nodes having `data` of type `T`, an `AVLNode` reference called `left` for the left child and an `AVLNode` reference called `right` for the right child.

### Methods

You will implement all standard methods for a Java data structure (add, remove, get etc.) and some utility methods. See the interface for more details. Note that some methods are worth more than others. If add is incorrect, then you are likely to fail most tests, as adding is crucial to the usability of a data structure.

## Path

By definition, Path is an alternating sequence of vertices (or nodes, in the case of a tree) and edges, connecting two distinct vertices. For this homework, the Path between two nodes is represented as a list of values between the starting node and the ending node. By the property of a Binary Search Tree (thus, an AVL Tree), there is always a unique Path between two nodes.

## Balancing

Unlike a BST, an AVL Tree must be self-balancing. Each node has two additional variables, `height` and `balanceFactor`. The `height` variable represents the height of the node (see below). The balance factor of a node is equal to its left child's height minus its right child's height. The tree should rotate appropriately to make sure that the tree is always balanced.

In order to determine if a rotation needs to be performed, you need to consider the balance factor of the node. When the absolute value of a balance factor is greater than one, ie. `|balanceFactor| > 1`, rotation should be performed.

There are 4 cases to consider when performing a rotation:

1. The tree is *left-heavy*, and the left subtree is *right-heavy* (the left child has a negative balance factor), then you need to do a left-right rotation.
2. The tree is *left-heavy*, and the left subtree is **not** *right-heavy* (the left child has a zero or positive balance factor), then you need to do a right rotation.
3. The tree is *right-heavy*, and the right subtree is *left-heavy* (the right child has a positive balance factor), then you need to do a right-left rotation.
4. The tree is *right-heavy*, and the right subtree is **not** *left-heavy* (the right child has a zero or negative balance factor), then you need to do a left rotation.

In general, although there may be a few different ways to do rotations, you need to perform as few rotations as possible.

## Height

You will implement a method to calculate the height of the tree. The height of any given node is `max(child nodes' height) + 1`. The height of the tree is the height of the root node. A leaf node has a height of 0. The height variable of each node must be set to be the height of the node (calculated using the above formula).

## Depth

You will also implement a method to calculate the depth of a particular value in the tree. The depth of any given node is `parent node depth + 1`. The root node has a depth of 1.

## A note on JUnits

We have provided a **very basic** set of tests for your code, in `AVLStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor does it guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub. There will be a link to the Georgia Tech GitHub as well as a list of JUnits other students have posted on the class Piazza (when it comes up).

If you need help on running JUnits, there is a guide, available on T-Square under Resources, to help you run JUnits on the command line or in IntelliJ.

## Style and Formatting

It is important that your code is not only functional but is also written clearly and with good style. We will be checking your code against a style checker that we are providing. It is located in T-Square, under Resources, along with instructions on how to use it. We will take off a point for every style error that occurs. If you feel like what you wrote is in accordance with good style but still sets off the style checker please email Joonho Kim (jkim844@gatech.edu) with the subject header of "CheckStyle XML".

### Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs (remember to keep helper methods private). If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs.

### Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong**. "Error", "BAD THING HAPPENED", and "fail" are not good messages. The name of the exception itself is not a good message.

For example:

```
throw new PDFReadException("Did not read PDF, will lose points.");

throw new IllegalArgumentException("Cannot insert null data into data structure.");
```

### Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new AVL<Integer>()` instead of `new AVL()`. Using the raw type of the class will result in a penalty.

## Forbidden Statements

You may not use these in your code at any time in CS 1332. If you use these, we will take off points.

- `break` may only be used in switch-case statements
- `continue`
- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Collections` class
- `Collection.toArray()`

- Reflection APIs
- Inner or nested classes

Debug print statements are fine, but nothing should be printed when we run them. We expect clean runs - printing to the console when we're grading will result in a penalty.

## Provided

The following file(s) have been provided to you. There are several, but you will only edit one of them.

1. `AVLInterface.java`

   This is the interface you will implement. All instructions for what the methods should do are in the javadocs. **Do not alter this file.**

2. `AVL.java`

   This is the class in which you will implement the interface. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

3. `AVLNode.java`

   This class represents a single node in the AVL Tree. It encapsulates the `data`, the `left` reference and the `right` reference. **Do not alter this file.**

4. `AVLStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `AVL` class. It is not intended to be exhaustive and does not guarantee any type of grade. **Write your own tests to ensure you cover all edge cases.**

## Deliverables

You must submit all of the following file(s). Please make sure the filename matches the filename(s) below. Be sure you receive the confirmation email from T-Square, and then download your uploaded files to a new folder, copy over the interfaces, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `AVL.java`