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, instance/global variables, or method signatures. For example, do not add `throws` to the method headers since they are not necessary.
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.ArrayList` for an Array 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 inefficient 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.

AVLs

You are required to implement an AVL tree. An AVL is a special type of binary search tree that follows all the same rules: each node has 0-2 children, all data in the left subtree is less than the node's data, and all data in the right subtree is greater than the node's data. The AVL differs from the BST with its own self-balancing rotations, which you must implement.

All methods in the AVL tree that are not O(1) **must be implemented recursively**. Good recursion with simple, focused states is strongly encouraged for this assignment in particular. We highly recommend a technique we call "pointer reinforcement" for this assignment; this is discussed in lecture, recitation, and class resources if you'd like to check it out.

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

Balancing

Each node has two additional instance variables, `height` and `balanceFactor`. The `height` variable should represent the height of the node. If you recall, a node's height is `max(child nodes' heights) + 1` where the height of a null is -1. The balance factor of a node should be equal to its left child's height minus its right child's height. Since we've stored this information in each node, we no longer need to recursively compute them.

The tree should rotate appropriately to make sure it's always balanced. For an AVL tree, a tree is balanced if every node's balance factor is either -1, 0, or 1. Keep in mind that you will have to update the balancing information stored in the nodes on the way back up the tree after modifying the tree; the variables are not updated automatically.

Important Notes

Here are a few notes to keep in mind when switching from BST to AVL trees:

1. For remove, use the **predecessor**, not successor.
2. After every change to the tree, make sure to update height and balance factor fields of all nodes whose subtrees have been modified.
3. Make sure the height method is `O(1)`.

Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in this PDF, and in other various circumstances.

Methods:	
add	25pts
remove	30pts
get	6pts
contains	6pts
clear	3pts
height	2pts
constructor	3pts
Other:	
Checkstyle	10pts
Efficiency	15pts
Total:	100pts

Collaboration Policy

Every student is expected to read, understand and abide by the Georgia Tech Academic Honor Code.

When working on homework assignments, you **may not** directly copy code from any source (other than your own past submissions). You are welcome to collaborate with peers and consult external resources, but you **must** personally write all of the code you submit. **You must list, at the top of each file in your submission, every student with whom you collaborated and every resource you consulted while completing the assignment**.

You may not directly share any files containing assignment code with other students or post your code publicly online. If you wish to store your code online in a personal private repository, you can use Github Enterprise to do this for free.

The only code you may share is JUnit test code on a pinned post on the official course Piazza. Use JUnits from other students at your own risk; **we do not endorse them**. See each assignment's PDF for more details. If you share JUnits, they **must** be shared on the site specified in the Piazza post, and not anywhere else (including a personal GitHub account).

Violators of the collaboration policy for this course will be turned into the Office of Student Integrity.

A note on JUnits

We have provided a **basic** set of tests for your code, in `AVLStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor do they 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 that other students have posted on Piazza from the class.

If you need help on running JUnits, there is a guide, available on Canvas under Files, 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 written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you, and is located on Canvas. It can be found under Files, along with instructions on how to use it. A point is deducted for every style error that occurs. If there is a discrepancy between what you wrote in accordance with good style and the style checker, then address your concerns with the Head TA.

Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs. If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs. Any Javadocs you write must be useful and describe the contract, parameters, and return value of the method; random or useless javadocs added only to appease Checkstyle may lose points.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies not only to comments/javadocs but also things like variable names.

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**. Examples of bad messages are: "Error", "BAD THING HAPPENED", and "fail". The name of the exception itself is, also, not a good message.

For example:

Bad: `throw new IndexOutOfBoundsException("Index is out of bounds.");`

Good: `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 LinkedNode<Integer>()` instead of `new LinkedNode()`. 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.

- `package`
- `System.arraycopy()`
- `clone()`

- `assert()`
- `Arrays` class
- `Array` class
- `Thread` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

If you're not sure on whether you can use something, and it's not mentioned here or anywhere else in the homework files, just ask.

Debug print statements are fine, but nothing should be printed when we run your code. We expect clean runs - printing to the console when we're grading will result in a penalty. If you submit these, we will take off points.

Provided

The following file(s) have been provided to you. There are several, but we've noted the ones to edit.

1. `AVL.java`

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

2. `AVLNode.java`

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

3. `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, and that *only* the following file(s) are present. The only exception is that Canvas will automatically append a `-n` depending on the submission number to the file name. This is expected and will be handled by the TAs when grading as long as the file name before this add-on matches what is shown below. If you resubmit, be sure only one copy of the file is present in the submission. If there are multiple files, do not zip up the files before submitting; submit them all as separate files.

Once submitted, double check that it has uploaded properly on Canvas. To do this, download your uploaded files to a new folder, copy over the support files, recompile, and run. It is your sole responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `AVL.java`