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.ArrayList` for a 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.

Binary Search Tree

You are to code a binary search tree. A binary search tree is a collection of nodes, each having a data item and a reference pointing to the left and right child nodes. The left child node and all of its children are less than the data. The right child node and all of its children are greater than the data. Therefore, all elements added to the tree must implement Java's generic `Comparable` interface. **This means that ALL comparison logic should be using the compareTo method.**

All methods in the BST that are not $O(1)$ **must be implemented recursively**, except for one specific edge case in `remove()`, see Javadoc for details).

Your binary search tree implementation will implement the BST interface provided. It will have two constructors: the 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. Any attempts to add data that is already in the tree should be ignored (the tree shouldn't be changed, and the duplicate item shouldn't get added).

Nodes

The binary search tree consists of nodes. The `BSTNode` class will be given to you; do not modify it.

Methods

You will implement all standard methods for a Java data structure (add, remove, etc.). See the interface for 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.

Traversals

You will implement 3 different ways of traversing a tree: pre-order traversal, in-order traversal, and post-order traversal. They must be implemented recursively! You may import Java's `LinkedList`/`ArrayList` classes as appropriate for these methods (but they may only be used for these methods and the FindPathBetween method).

FindPathBetween

You will implement a method to calculate the path between two elements in the tree. You will treat the first parameter as your starting point and the second as your ending point. Note that either piece of data could be anywhere in the tree, meaning you will have to at some point make a separate traversal for each piece of data. You may import Java's `LinkedList`/`ArrayList` classes as appropriate for these methods (but they may only be used for this method and the traversal methods). **However**, you are only allowed to use 1 instance of a List (determining which type of List to use is an exercise to you and is subject to efficiency deductions) to store the path. You must only traverse to each element once and any common ancestors may only be traversed once in total. This path must only include the deepest common ancestor only. Including other ancestors will result in 0 points. If the first data parameter is equivalent to the second data parameter, then the path between the data is simply the data in the tree itself (note that the length of the list should be 1 in this case).

Height

You will implement a method to calculate the height of the tree. The height of any given node is `max(left.height, right.height) + 1`. A leaf node has a height of 0.

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	24pts
remove	16pts
get	6pts
contains	6pts
traversals	6pts
path between	7pts
height	3pts
clear	2pts
constructor	5pts
Other:	
Checkstyle	10pts
Efficiency	15pts
Total:	100pts

Keep in mind that add functions are necessary to test other functions, so if an add doesn't work, remove tests might fail as the items to be removed were not added correctly. Additionally, the size function is used many times throughout the tests, so if the size isn't updated correctly or the method itself doesn't work, many tests can fail.

A note on JUnits

We have provided a **very basic** set of tests for your code, in `BSTStudentTests.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 Grayson Bianco (gbianco6@gatech.edu) with the subject header of "CheckStyle XML".

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 will lose points.

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 ArrayList<Integer>()` instead of `new ArrayList()`. 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.

- `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 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 you will only edit one of them.

1. `BSTInterface.java`

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

2. `BST.java`

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

3. `BSTNode.java`

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

4. `BSTStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `BST` 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. T-Square does **not** delete files from old uploads; you must do this manually. Failure to do so may result in a penalty.

After submitting, 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. `BST.java`