P09 Binary Gradebook

Overview

Imagine that you are designing a gradebook feature for a certain website whose name starts with a "C" and has a circular red logo (not to name names or anything). To keep track of the necessary information in the gradebook, each student should have an associated name, email, and current grade. As well, in order to make the gradebook usable for instructors who have a large class size, we should be able to easily add and remove students from the gradebook quickly (say in time $O(\log n)$ at average case), and look up a student's current grade quickly (also in time $O(\log n)$ at average case). This is opposed to the seemingly linear or quadratic time of the un-named website.

In order to accomplish this, you will use a **binary search tree (BST)** to store the data, and implement various recursive algorithms for **searching**, **adding**, **removing**, and **iterating** through the gradebook.

Learning Objectives

The goals of this assignment include:

- **Develop** a Binary Search Tree (NOT necessarily balanced) **from scratch**.
- **Implement** common Binary Search Tree (BST) operations.
- Further **develop** your experience in recursive problem-solving.
- **Contrast** multiple approaches to implementing the `java.util.Iterator` interface, and explain how an Iterator interacts with the `java.util.Iterable` interface to facilitate enhanced-for loops in Java.
- **Improve** your experience in developing unit tests.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P09 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM CT on Sunday April 21**.
+2.5 points	**5% BONUS Points:** Students whose final submission to Gradescope has a timestamp earlier than **4:59PM CT Wed April 24**and passes ALL the immediate tests will receive an additional 2.5 points toward this assignment's grade on gradescope, up to a maximum total of **50** points.
15 points	**Immediate Automated Tests:** Upon every submission of your assignment to Gradescope, you will receive immediate feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Passing all immediate automated tests does NOT guarantee full credit for the assignment.
20 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
10 points	**Manual Grading:** Human graders will review the commenting, style, and organization of your final submission. They will be checking whether it conforms to the requirements of the CS300 Course Style Guide. You will NOT be able to resubmit corrections for extra points, and should therefore carefully review the readability of your code with respect to the course style guide.
50 points	**MAXIMUM Total Score**

Assignment Requirements and Notes

(Please read carefully!)

Pair Programming and Use of External Libraries and Sources

- Pair programming is **NOT ALLOWED** for this assignment.
- Any source code provided in this specification may be included verbatim in your program without attribution.
- All other sources must be cited explicitly in your program comments, in accordance with the Appropriate Academic Conduct guidelines.
- Any use of **ChatGPT** or other large language models (LLM) must be cited AND your submission MUST include a file called `log.txt` containing the full transcript of your

usage of the tool. Failure to cite or include your logs is considered academic misconduct and will be handled accordingly.

- You are only allowed to **import** or **use** the libraries listed below in their respective files **only**.
- **The ONLY external libraries** you may use in your submitted file are:

```
java.util.NoSuchElementException
java.util.Iterator
```

The use of any other packages (outside of `java.lang`) is NOT permitted.

CS300 Assignment Requirements

This section is VALID for ALL the CS300 assignments

- If you need assistance, please check the list of our Resources.
- Read carefully through the specification provided in this write-up. This assignment requires clear understanding of its instructions. Read TWICE the instructions and do not hesitate to ask for clarification on piazza if you find any ambiguity.
- You MUST NOT add any additional fields either instance or static, and any public methods either static or instance to your program, other than those defined in this write-up.
- You CAN define local variables (declared inside a method's body) that you may need to implement the methods defined in this program.
- You MUST NOT add any additional fields either instance or static to your program, and any public methods either static or instance to your program, other than those defined in this write-up.
- You CAN define **private** methods to help implement the different public methods defined in this program, if needed.
- Your assignment submission must conform to the CS300 Course Style Guide. Please review ALL the commenting, naming, and style requirements.
- If starter code files to download are provided, be sure to remove the comments including the `TODO` tags from your last submission to gradescope.

- Avoid submitting code which does not compile. Make sure that ALL of your submitted files ALWAYS compile. A submission which contains compile errors won't pass any of the automated tests on gradescope.
- You can submit your **work in progress (incomplete work) multiple times** on gradescope. Be sure to include method stubs for the incomplete methods (this includes complete method signature and a default return statement if the method return type is not void). Your submission may include methods not implemented or with partial implementation or with a default return statement.
- Run your program locally before you submit to Gradescope. If it doesn't work on your computer, it will not work on Gradescope.
- You are responsible for maintaining secure back-ups of your progress as you work. The OneDrive and GoogleDrive accounts associated with your UW NetID are often convenient and secure places to store such backups. Aspiring students may try their hands at version control.
- Be sure to submit your code (work in progress) of this assignment on Gradescope both early and often. This will 1) give you time before the deadline to fix any defects that are detected by the tests, 2) provide you with an additional backup of your work, and 3) help track your progress through the implementation of the assignment. These tests are designed to detect and provide feedback about only very specific kinds of defects. **It is your responsibility to implement additional testing to verify that the rest of your code is functioning in accordance with this write-up.**

1 Getting Started

1. Create a new project in Eclipse, and in Eclipse, called something like **P09 Binary Gradebook**.
   a. Ensure this project uses Java 17. Select "JavaSE-17" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **not** create a project-specific package; use the default package.
2. Download **one** (1) Java source files from the assignment page on Canvas:
   - `BSTNode.java`
3. Create **five** (5) Java source files within your project's src folder:
   - `StudentRecord.java`
   - `Gradebook.java`
   - `GradebookTester.java`

- `GradebookIterator.java`
- `PassingGradeIterator.java`

Only the `GradebookTester` class should contain a `main()` method.

2 Guidelines

In this project you will implement an application that stores, allows access to, and modifies student grade information in a binary search tree. We will provide only the JavaDocs for all five remaining classes, and you should carefully implement them to follow the specifications in the JavaDocs comments and this write-up. We will give you some visualizations and examples in the writeup, but all the information needed to complete the assignment is contained in the provided JavaDocs . Below is the recommended order that you complete the assignment in:

1. `StudentRecord`
2. `Gradebook, GradebookTester`
   (a) `constructorTester(), Gradebook Constructor`
   (b) `isEmptySizeAddTester(), Gradebook.isEmpty(), Gradebook.size(), addStudent()`
   (c) `toStringTester(), Gradebook.toStringHelper(), Gradebook.toString()`
   (d) `prettyStringTester(), Gradebook.prettyString()`
   (e) `lookupTester(), lookupHelper(), lookup()`
   (f) `getMinTester(), getMinHelper(), getMin()`
   (g) `removeStudentTester(), removeStudentHelper(), removeStudent()`
   (h) `successorTester(), successorHelper(), successor()`
   (i) `iteratorTester(), GradebookIterator constructor, hasNext(), and next()`
   (j) `passingIteratorTester(), PassingGradeIterator constructor, advanceToNextPassingGrade(), hasNext(), and next()`

3 StudentRecord

This is a simple class to store grading information about a student in a class, including their name, email, and grade. **Note that the name and email fields of this class are public final**, and hence do not need to have any accessors or mutators. Make sure to implement the Comparable interface.

4 BSTNode

The provided generic class `BSTNode<T>` represents nodes in the BST, which each store a comparable data object of type T, and have a left and right child reference of type `BSTNode<T>`.

- **The provided `BSTNode.equals()` method can be used to check that two entire trees rooted at the given nodes are exactly the same** (note the recursive call to `BSTNode.equals()`), and similarly the `Gradebook.equalBST()` method should compare the structure of the Gradebook object to a manually created BST of nodes.
- These methods should be used by your testers to verify that the structure of your BSTs are correct, not just that the size or `toString()` of your BSTs are correct!

5 GradebookTester

There is a corresponding tester method in the `GradebookTester` class for each method in the `Gradebook` class. There are **ten** tester methods to be developed in total.

```
constructorTester()   isEmptySizeAddTester()   lookupTester()
toStringTester()      prettyStringTester()
getMinTester()        successorTester()        removeStudentTester()
iteratorTester()      passingIteratorTester()
```

You should work on developing test cases ahead of completing your methods (or at least at the same time) to verify that both you understand what the methods should be doing, and that you have accurately implemented them. When possible, **you should utilize the BSTNode class to construct trees to test on your various helper methods, and use the `BSTNode.equals()` and the `Gradebook.equalBST()` methods to test that the output of the methods is as expected**. This may require you to draw out the examples by hand to determine what the expected output should be, but this will greatly improve your understanding of BSTs!

Be sure to consider **edge cases** (empty BST, matching element found at the root, or no matching element found, for instance) and **normal cases**. When you consider a non-empty tree, try building a tree with at least three levels (whose height counting the number of nodes is at least three). Be sure to consider test scenarios involving recursive cases going on both subtrees (left and right). The `Gradebook.equalBST()` implementation details are provided in the following.

Provided Gradebook.equalBST() method: You can use the below code in verbatim in your `Gradebook` class.

```
/**
 * Returns true if this BST has an identical layout (all subtrees equal) to the given tree.
 *
 * @author Ashley Samuelson
 * @see BSTNode#equals(Object)
 * @param node tree to compare this Gradebook to
 * @return true if the given tree looks identical to the root of this Gradebook
 */
public boolean equalBST(BSTNode<StudentRecord> node) {
  return root == node || (root != null && root.equals(node));
}
```

6 Gradebook

Now for the main course (see what I did there). Implement ALL the methods in the `Gradebook` class and the corresponding tester methods according to the provided JavaDocs and your understanding of BSTs. The order of methods to complete that was provided in Section 2 is optional, but is ordered roughly from easiest to most difficult to slowly build-up your understanding before completing the more difficult methods. Below, we will provide some helpful tips, visualizations, and examples for each of the methods to help you along the way.

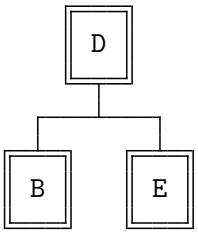
6.1 Gradebook.addStudent()

Note that duplicate StudentRecords are NOT allowed in this gradebook. The `addHelper()` method must call the `addStudentHelper()` method to operate. The helper method `addStudentHelper()` **should be implemented recursively**. Also keep in mind that the helper method should **return a BSTNode object representing the root of the given subtree with the Student added.** This is your recursive problem and subproblems! See Figure 1 for example output for this method.

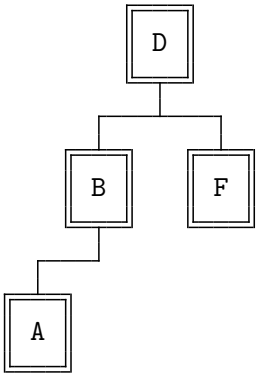
6.2 Gradebook.toString()

This method is critical for debugging your other code, so make sure you test it thoroughly before moving on! This method should return the students in an **in-order traversal** of the BST. (The 2D tree visualizations shown below and in other examples are not part of this assignment. They are provided purely for illustrative purposes.) **The helper method toStringHelper() should be implemented recursively.** See Figure 2 for example output for this method.

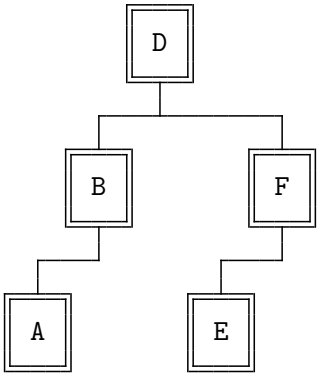
BST:



After calling BST.addStudent(A):



After calling BST.addStudent(E):



After calling BST.addStudent(C):

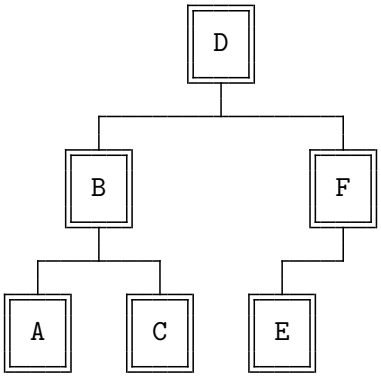


Figure 1: Example output for Gradebook.addStudent()

6.3 Gradebook.prettyString()

This method will also be useful for debugging and for visualizing the structure of your trees, so write adequate testers! This method should return the students in a **backwards in-order traversal (right-current-left)** of the BST, with the indentation (space from the left side of the screen to the student names) **increasing by four (4) spaces at each level of depth** in the tree. This allows you to easily visualize the structure of the tree, but it will be rotated 90 degrees counterclockwise (rotate your head to the left to see the normal top-down view). We provide you in the following with a recursive implementation of the helper method `prettyStringHelper()` that you can use in verbatim in your submitted code. See Figure 3 for example output for the `prettyString()` method.

Hint: `String.repeat()`

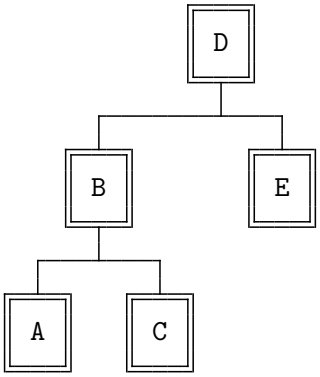
```
/**
   * Returns a decreasing-order String representation of the structure of this subtree,
   * indented by four spaces for each level of depth in the larger tree.
   *
   * @author Ashley Samuelson
   * @param node current subtree within the larger tree
   * @param depth depth of the current node within the larger tree
   * @return a String representation of the structure of this subtree
   */
  public static String prettyStringHelper(BSTNode<StudentRecord> node, int depth) {
    if (node == null) {
      return "";

    }
    String indent = "  ".repeat(depth);
    return prettyStringHelper(node.getRight(), depth + 1) + indent + node.getData().name
        + "\n" + prettyStringHelper(node.getLeft(), depth + 1);
  }
```

6.4 Gradebook.lookup()

The classic use-case for BSTs. This method should return the Student in the tree with a matching email address, or null if none exists. Since Student.compareTo() only compares emails, you can create a dummy student object with a fake name and grade to pass into the helper method. **The helper method `lookupHelper()` should be implemented recursively.** See Figure 4 for example output for this method.

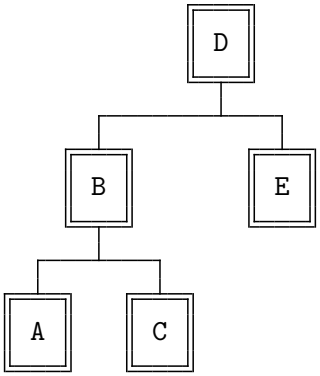
```
BST:
```




```
Expected output of BST.toString():
A
B
C
D
E
```

Figure 2: Example output for Gradebook.toString()

```
BST:
```




```
Expected output of BST.prettyString():
    E
D
        C
    B
        A
```

Figure 3: Example output for Gradebook.prettyString()

6.5 Gradebook.getMin()

This should return the smallest valued (by compareTo()/email) student record in the tree. This method and the associated helper will be useful to use in the iterators and other methods in the Gradebook class. **The helper method getMinHelper() should be implemented recursively.** See Figure 5 for example output for this method.

6.6 Gradebook.removeStudent()

This method is one of the more difficult ones. Make sure to consider all possible cases (e.g. no children, one child, two children), and make a plan for how to handle each case. **The helper method removeStudentHelper() should be implemented recursively.** Also keep in mind that the helper method should **return a BSTNode object representing the root of the given subtree with the Student removed.** In the two-child case, we will choose to **replace the node with its successor.** See Figure 6 for example output for this method.

Hint: Where is the successor of a node in the BST? To find the successor of a node you already have a reference to, you only need to use the getMinHelper() method, rather than the successor() method.

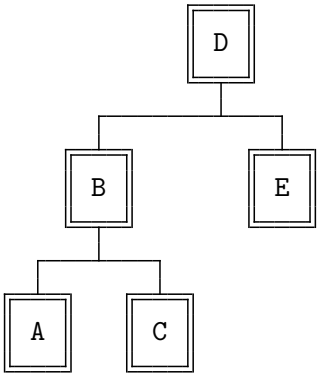
6.7 Gradebook.successor()

This method may be surprisingly difficult to implement correctly. **The helper method successorHelper() should be implemented recursively.** Recall that the successor of a target value x in a BST (where x is not necessarily stored in the BST), is the **smallest value in the BST that is larger than x** (i.e. the next-largest value). See Figure 7 for example output for this method.

Hint: There are three cases to consider.

1. The target data is greater than or equal to the root node (in which subtree must the successor be?)
2. The target data is less than the root node
    a. There is at least one node in the left subtree that is larger than the target (in which subtree must the successor be?)
    b. There are no nodes larger than the target in the left subtree (which node is the successor?)

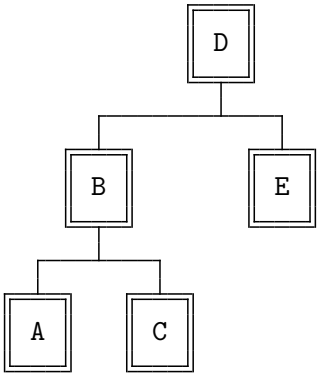
```
BST:
```




```
Expected output of BST.lookup(B):
B
Expected output of BST.lookup(F):
null
```

Figure 4: Example output for Gradebook.lookup()

```
BST:
```




```
Expected output of getMinHelper(D):
A
Expected output of getMinHelper(E):
E
```

Figure 5: Example output for Gradebook.getMinHelper()

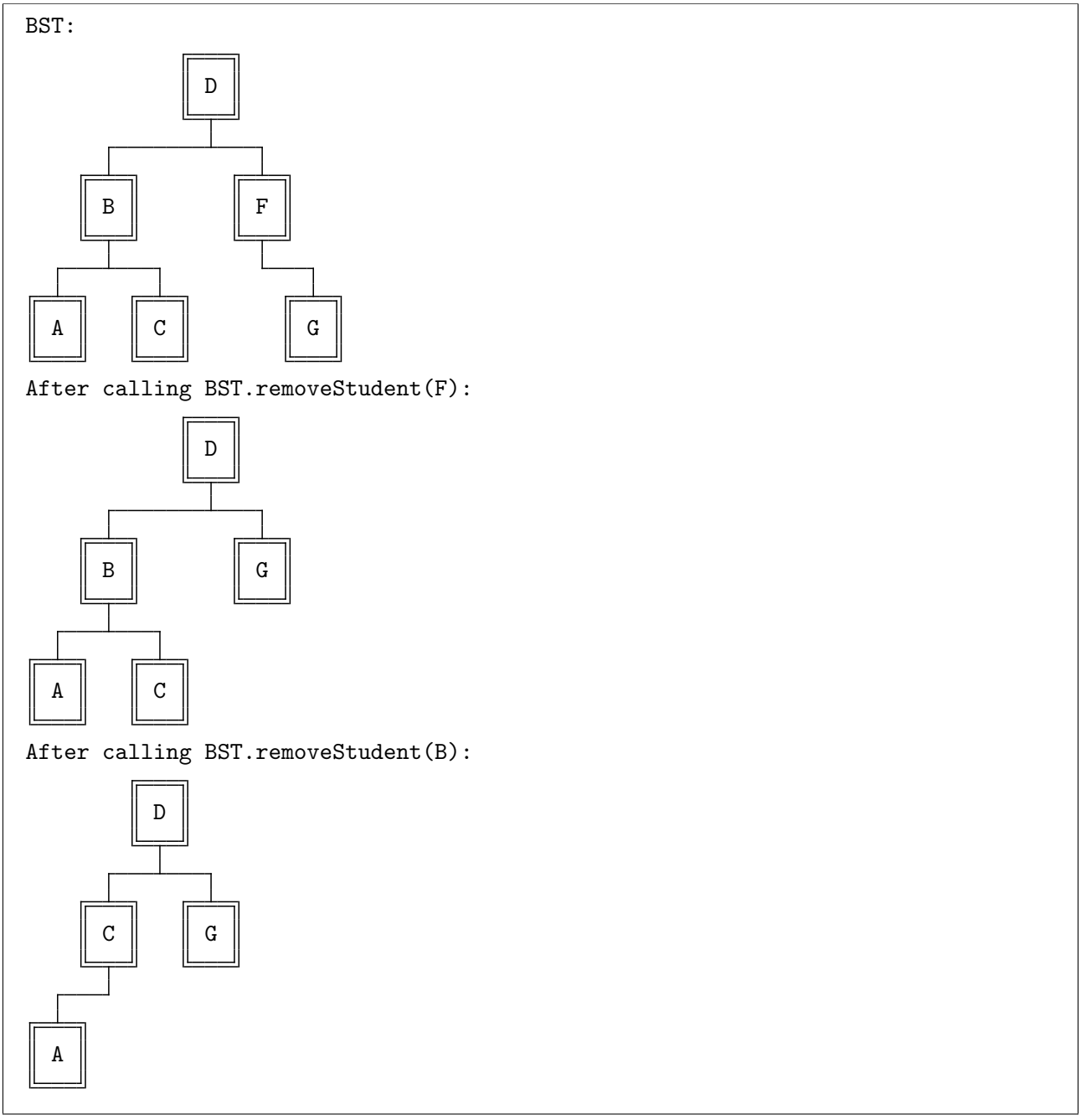
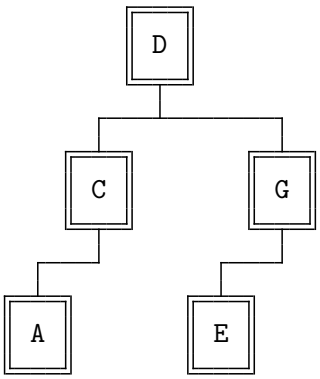


Figure 6: Example output for Gradebook.removeStudent()

```
BST:
```




```
Expected output of BST.successor(B):
C
Expected output of BST.successor(A):
C
Expected output of BST.successor(D):
E
Expected output of BST.successor(G):
null
Expected output of BST.successor(H):
null
```

Figure 7: Example output for Gradebook.successor()

7 Iterators

While a BST gives a structured and efficient hierarchical representation of our data, sometimes it is useful to be able to **progress through the represented data in a linear fashion.** In the `GradebookIterator` and `PassingGradeIterator` classes, you will create iterators that allow you to progress through the nodes of the BST in an in-order fashion, and progress through the nodes that satisfy a certain filtering condition, such as having a passing grade in the course. Note that PassingGradeIterator extends the base class GradebookIterator. Make sure to implement the `Gradebook.iterator()` method using these classes once they are completed.

Hint: `GradebookIterator`: Given a current Student reference, **how do we find the next-largest entry in the BST?**

Hint: For the `PassingGradeIterator`, you can use the super-class methods from `GradebookIterator` to advance the iterator to the next StudentRecord with passing grade in the iteration.

8 Testers

If you haven't already been working on your tester methods (for shame!), here is a list of the **ten** (10) tester methods that you will need to implement. There are no specific requirements for the tester methods; just that they can distinguish a correct from an incorrect implementation of the relevant method.

```
constructorTester()   isEmptySizeAddTester()   lookupTester()
toStringTester()      prettyStringTester()
getMinTester()        successorTester()        removeStudentTester()
iteratorTester()      passingIteratorTester()
```

9 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only five (5) files that you must submit are

- `StudentRecord.java`
- `Gradebook.java`
- `GradebookTester.java`
- `GradebookIterator.java`
- `PassingGradeIterator.java`

Additionally, if you used any `generative AI` at any point during your development, you must include the full transcript of your interaction in a file called

- log.txt

Your score for this assignment will be based on your **"active"** submission made prior to the assignment due date of Due: **9:59PM CT Thu April 25**. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.