CSCI241 Spring 2020: Assignment 2
Due Monday, May 11th at 9:59pm

Your submission for this and all future homework assignments, must be your own work. You may discuss topics and concepts at a high level and brainstorm using a white board, but you cannot share, disseminate, co-author, or even view, another student's code. Please refer to the academic honesty guidelines on the syllabus for more details. If any of this is unclear, please ask for further clarification.
If you rely on any external resources (e.g., the internet, other textbooks, etc.), you MUST cite those resources in your hours.txt file. Under no circumstances may you cut-and-paste entire blocks of code from the internet, other current or past students, or anywhere else—if you do, you will receive an F in the course and be reported to the Dean of Students.

1 Overview

You are given a partial implementation of a Binary Search tree in AVL.java. Your task will be to complete this implementation and turn the BST into an AVL tree by maintaining the AVL balance property on insertion. You will use this AVL tree to efficiently count the number of unique words in a text document.

2 Getting Started

The Github Classroom invitation link for this assignment is in Assignment 2 on Canvas. Begin by accepting the invitation and cloning a local working copy of your repository as you did in Assignment 1. Make sure to clone it somewhere outside the local working copies for other assignments and labs (e.g., clone to `~/csci241/a2`) to avoid nesting local repositories.

3 Program Behavior: User's perspective

Some helpful skeleton code for the word-counting application is provided in Vocab.java.

Each command-line argument to the Vocab program is a text file. The program reads words from a text file, removing whitespace and punctuation, and normalizing to all lower case - this means "Band" and "band," will both come out as "band" and you won't count the same word as two different ones.

For each text file, your program should then print two numbers on a single line:

1. The number of *unique* words used in the document.
2. The *total* number of words used in the document.

If the program receives no command-line arguments, it should read text from standard in (System.in) until an end-of-file character is reached (in linux you can send the EOF character to a process by pressing Ctrl+d; in Windows, I'm told the shortcut is Ctrl+z). If this doesn't work, try pressing the same shortcut twice in a row.

Some sample invocations of the program appear below. The user typing the EOF character is represented as `^D`. Note that the particulars of the gradle output may differ depending on the particular state of your gradle build, but the inputs and outputs of a working solution should match those shown in the examples below.

```
$ gradle run
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
Words, words, words.
What is the matter, my lord?
Between who?
I mean, the matter that you read, my lord.^D
> Task :run

14 20

BUILD SUCCESSFUL in 18s
2 actionable tasks: 1 executed, 1 up-to-date

$ gradle run --args="a.txt b.txt"
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE

> Task :run
14 20
1955 8060

BUILD SUCCESSFUL in 0s
2 actionable tasks: 1 executed, 1 up-to-date

$ gradle run --args="b.txt"
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE

> Task :run
1955 8060

BUILD SUCCESSFUL in 0s
2 actionable tasks: 1 executed, 1 up-to-date
```

4 Your Tasks

Skeleton code is provided in your repository. The AVL class in `src/main/java/avl/AVL.java` currently implements the `search` functionality for a BST.

1. Implement standard BST (*not* AVL) insert functionality in the provided `bstInsert` method stub. As with `search`, the AVL class has a public `bstInsert(String w)` method that calls a private `bstInsert(Node n, String w)` method that recursively inserts on nodes. Notice that AVL class has a `size` field that should be kept up to date as words are inserted. *Note: `bstInsert` does not need to keep heights up-to-date; this is only necessary in `avlInsert`, and you can assume that `bstInsert` calls are not mixed with avlInsert calls.*

2. Implement `leftRotate` and `rightRotate` helper methods to perform a rotation on a given node. Use the lecture slides as a reference.

3. Implement `rebalance` to fix a violation of the AVL property caused by an insertion. In the process, you'll need to correctly maintain the `height` field of each node. Remember that height needs to be updated any time the tree's structure changes (insertions, rotations).

4. Implement `avlInsert` to maintain AVL balance in the tree after insertions using the `rebalance` method.

5. Use your completed AVL tree class to efficiently ($O(n \log n)$) count the number of unique words found in each document processed by Vocab. Because insertion ignores duplicates, the number of unique words will simply be the size of the tree.

6. For the final 5 points, complete some or all of the series of enhancements described below.

7. Record the total number of hours you spent on this assignment in the provided "hours.txt" file. The file should contain one line with a single integer representing the estimated time you took to complete the project.

4.1 Implementation notes

- **Public method specifications and signatures should not be changed**: if method names, call signatures, or return values change, your code will not compile with the testing system and you'll receive no credit for the correctness portion of your grade.

- You may write and use as many `private` helper methods as you need. You are especially encouraged to use helper methods for things like calculating balance factors, updating heights, etc., in order to keep the code for intricate procedures like `rebalance` easy to read.

- The skeleton code implements a try/catch block to skip nonexistent files in Vocab.java. Error catching beyond this is not required - you may assume well-formed user input and that method preconditions will not be violated.

- Be careful with parent pointers. A recursive tree traversal such as the reverse-in-order traversal used in printTree never follows parent pointers; this means parent pointers can be misplaced and printTree will still look normal.

- Keep in mind that the `height` method from Lab 4 is $O(n)$, which means you can't call it on every node and expect to maintain the $O(n \log n)$ runtime in AVLInsert: instead you need to update the height of each node along the insertion path from the bottom up, updating each node's height field using the `height`s of its children.
- You are provided with a test suite in `src/test/java/avl/AVLTest.java`. Use `gradle test` often and pass tests for each task before moving onto the next.

5 Enhancements

The base assignment is worth 45/50 points. The final 5 points may be earned by completing the following enhancements. You may also come up with your own ideas - you may want to run them by the instructor to make sure they're worthwhile and will result in points awarded if successfully completed. It is highly recommended that you complete the base assignment before attempting any enhancements.

Enhancements and git The base project will be graded based on the master branch of your repository. Before you change your code in the process of completing enhancements, create a new branch in your repository (e.g., `git checkout -b enhancements`). Keep all changes related to enhancements on this branch—this way you can add functionality, without affecting your score on the base project. Make sure you've pushed both master and enhancements branches to GitHub before the submission deadline.

The final five points can be earned as follows:

1. (2 points) Implement `remove` using standard BST removal (without maintaining AVL balance).
2. (1 point) Modify your `remove` method to maintain AVL balance through removals.
3. (1 point) Add a `count` field to the `Node` class and modify `avlInsert` and `remove` to maintain `count` as the net number of additions/removals of a word. In other words, count should store the number of times the word has been added minus the number of times it has been removed. If count gets to zero, remove the node from the tree.
4. (1 point) If a single file is specified as a command line argument, print the number of unique words over a fixed-width sliding window as the text is processed. For example, for a window size of 30, maintain a set of the most recent 30 words; after each word is read, print the number of unique words in the most recent 30, separated by newlines.

If you complete any of the above, explain what you did and any design decisions made in a comment at the top of the corresponding java file.

6 Game Plan

Start small, test incrementally, and git commit often. Please keep track of the number of hours you spend on this assignment, as you will be asked to report it in hours.txt. Hours spent will not affect your grade.

The tasks are best completed in the order presented. Make sure you pass the tests for the current task before moving on to the next. Rotations and rebalancing are the trickiest part. Visit the mentors, come to office hours, or post on Piazza if you are stuck. A suggested timeline for completing the assignment in a stress-free manner is given below:

1. By the end of Sunday 5/3: BST insertion completed and tested.
2. By the end of Tuesday 5/5: rotations implemented and tested.
3. By the end of Thursday, 5/7: rebalance implemented and tested.
4. By Saturday, 5/9 AVL insertion and Vocab behavior implemented and tested.
5. By Monday, 5/11: Any enhancements completed, hours recorded, and final changes pushed to GitHub.

7 How and What to Submit

Submit the assignment by pushing your final changes to GitHub before the deadline. Be sure to fill in `hours.txt` with a single integer representing the estimated number of hours you spent on this assignment. Feel free to tell me how the assignment went on the line below the number of hours. If you completed any enhancements, be sure to push your enhancements branch as well.

Rubric

You can earn points for the correctness and efficiency of your program, and points can be deducted for errors in commenting, style, clarity, and following assignment instructions.

Git Repository	
Code is pushed to github and hours spent appear as a lone integer in hours.txt	1 point
Code : Correctness	
Unit tests (n tests passed earns $\lceil 1.\overline{33}n \rceil$ points)	28
Vocab prints the unique and total counts of words from standard input	3
Vocab prints the unique and total counts of words from each command-line argument	3
Code : Efficiency	
avlInsert maintains O(log n) performance by keeping track of node heights and updating them as necessary	5
Vocab processes a document with n words in O(n log n) time	5
Enhancements	
`remove` correctly removes a node from the tree	2
`remove` maintains AVL balance	1
The tree tracks a count for each node, inserting or incrementing on insertion and decrementing or removing on removal.	1
Sliding window vocabulary tracking is implemented.	1
Clarity deductions (up to 2 points each)	
Include author, date and purpose in a comment comment at the top of each file you write any code in	
Methods you introduce should be accompanied by a precise specification	
Non-obvious code sections should be explained in comments	
Indentation should be consistent	
Methods should be written as concisely and clearly as possible	
Methods should not be too long - use private helper methods	
Code should not be cryptic and terse	
Variable and function names should be informative	
Total	50 points