CSCI241 Fall 2020: Assignment 2
Due Monday, November 2nd 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 A2 Survey submission. 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 provided with 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 lines in a text document.

Putting elements into a set is one way to find how many unique items are in the set. Though we aren't using the name, you'll notice that our AVL tree is implementing the Set ADT here, storing a set of Strings with no duplicates allowed.

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 Main Program

The main program is found in `Unique.java`. This program takes two command line arguments: a mode and the name of a text file. The mode is either `naive` or `avl` (it defaults to `avl` if anything other than those two is given). The program reads lines from the text file one by one and prints the number of unique lines. A naive $O(n^2)$ solution is already implemented for you in `naiveUnique`. Your job is to implement the `avlUnique` method such that it counts the number of unique lines in the file in $O(n \lg n)$. To do this, you'll complete the AVL tree implementation in `AVL.java`.

Two sample text files are provided to demo the difference between the naive and efficient solutions. `prefixes.txt` contains the first 5 characters of each word in a large list of English words (you can find a similar one on the lab systems at `/usr/share/dict/american-english`). `prefixes_small.txt` contains the first 50,000 lines of that file. On my laptop, the naive implementation takes around 26 seconds on the complete file and about 3 seconds for the small file. The efficient implementation takes a second or less on either file, thanks to its $O(n \log n)$ runtime.

```
$ gradle run --args "avl prefixes.txt"
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE

> Task :run
Finding unique lines in prefixes.txt
prefixes.txt
AVL:
65137

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

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

> Task :run
Finding unique lines in prefixes.txt
Naive:
65137

BUILD SUCCESSFUL in 26s
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 ($O(n \log n)$) implement the `avlUnique` method in `Unique.java`.

6. For up to 5 points of extra credit, you may complete some or all of the enhancements described below.

7. Submit the A2 Survey quiz, including the estimated total number of hours you spent on this assignment.

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 handle nonexistent files in `Unique.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 3 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, setting 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

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.

You can earn up to 3 points of extra credit for completing the following:

1. (1 point) Implement `remove`, maintaining AVL balance.
2. (1 point) The base assignment implements a Set of Strings. In your enhancements branch, modify your code to instead implement a Map from strings to integers. This will allow it to behave something like a `HashMap<String, Integer>` would. In the context of lines of a document, this means you'll keep track of the number of occurrences of each line you've seen. Modify your main program to use this to calculate the most frequently-occurring line in addition to the number of unique lines.
3. (1 point) Modify your tree to fully support the following semantics for removal: removing an element either decrements its count (if count was greater than 1) or removes it from the tree entirely (if the count was 1).

If you complete any of the above, explain what you did, how you did it, and instructions for testing your enhancements 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 A2 Survey. 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:

- By 10/21: BST insertion completed and tested.
- By 10/23: rotations implemented and tested.
- By 10/26: rebalance implemented and tested.
- By 10/28: AVL insertion and main program behavior implemented and tested.
- By 11/2: Any enhancements completed.
- By 10pm on 11/2: Final changes pushed to github, A2 Survey submitted.

7 How and What to Submit

Submit the assignment by pushing your final changes to GitHub before the deadline, then submitting A2 Survey on Canvas. 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 are reported in A2 Survey.	1 point
Code : Correctness	
Unit tests (1.5 points per test)	33
Main program prints the correct number of unique lines	6
Code : Efficiency	
avlInsert maintains O(log n) performance by keeping track of node heights and updating them as necessary	5
Unique processes a document with n words in O(n log n) time	5
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