CSCI 241: Data Structures
Assignment 2: BST and AVL

Note about collaboration and cheating. You may chat with another student on a high-level about topics and concepts, but you **cannot** share, disseminate, co-author, or even view, other student's code. Even helping another student debug their program so that their source code is seen should be avoided. Also, you cannot take, in part or in whole, any code from any other outside source, including the internet. Please ask if any of this is unclear.

Warning

Do **not** post your solutions on publicly accessible websites or make them public (e.g., public repository on GitHub). This assignment may be used in future courses. You sharing solutions publicly facilitates cheating and your course grade could be retroactively changed accordingly.

Tip

- Do not start coding right away. First, read the *entire* assignment and read it *carefully*.
- Then write pseudocode in the form of comments and when you are confident about it, implement it.
- Start small, test incrementally, and git commit often.
- It can be helpful to work in brief sessions (1-3 Pomodoros at a time) instead of a single long session.
- If you are stuck with an issue/bug for close to an hour, reach out for help (via email, office hour, CS mentors). Don't waste time spinning your wheels.
- When reaching out for help via email, always make sure your code is committed and pushed so we can look at if required.

1 Overview

In this assignment, you'll implement BST and AVL tree operations. 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.

How to do that? Recall that the BST invariant implies that there are no duplicates in BST and AVL trees. So, to determine unique lines in a file, you'll insert each line in the given file as a node in an AVL tree. In the end, the size of the resulting AVL tree will give you the number of unique elements in the files. 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 $\mathcal{O}(n^2)$ solution is already implemented for you in the `naiveUnique` method. Your job is to implement the `avlUnique` method such that it counts the number of unique lines in the file in $\mathcal{O}(n \log n)$. To do this, you'll need to complete the AVL tree implementation in `AVL.java`.

Two sample text files (one large, `prefixes.txt` and one small, `prefixes_small.txt`) 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 `prefixes.txt`. On my computer, the naive implementation takes around 14 seconds on

the large file and about 1 seconds for the small file (See `BUILD SUCCESSFUL` times in the log below). An efficient AVL implementation takes less than a second on either file, thanks to its $\mathcal{O}(\log n)$ runtime.

```
  $ ./gradlew run --args "naive prefixes.txt"

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

BUILD SUCCESSFUL in 14s
2 actionable tasks: 1 executed, 1 up-to-date
$ ./gradlew run --args "naive prefixes_small.txt"

> Task :app:run
Finding unique lines in prefixes_small.txt
Naive:
24886

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
$ ./gradlew run --args "avl prefixes.txt"

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

BUILD SUCCESSFUL in 528ms
2 actionable tasks: 1 executed, 1 up-to-date
$ ./gradlew run --args "avl prefixes_small.txt"

> Task :app:run
Finding unique lines in prefixes_small.txt
prefixes_small.txt
AVL:
24886

BUILD SUCCESSFUL in 438ms
2 actionable tasks: 1 executed, 1 up-to-date
$
```

4 Your tasks

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

1. First, 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 should 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.
4. Implement `avlInsert` to maintain AVL balance in the tree after insertions using the `rebalance` method.
5. Use your completed AVL tree class to implement the `avlUnique` method in `Unique.java` such that it runs in $\mathcal{O}(n \log n)$ time.
6. For up to 3 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

In the class, in the examples we covered, nodes in trees had integer values. The key (also called as value) in the nodes can also be of a different data type as long as the data type is `comparable`, that is, we can compare the key of two nodes to determine which one is smaller, greater, or if they are equal. Here is the definition of the inner `Node` class from `AVL.java`

```
public class Node {
  public String word;
  public Node parent;
  public Node left;
  public Node right;
  public int height;

  // other code
}
```

Notice that the `Node` key is `word` (of type `String`). Use this to compare two nodes. How to compare two strings? Use the `compareTo` string function. Check javadocs for String on how to use `compareTo` method. Also notice that each node has an additional data, `height`, which represents the height of the subtree rooted at the node. Whenever you insert or remove a node, or rebalance, height of some nodes will change. You'll need to identify those nodes and update their height. Maintaining height at each node will allow you to implement an efficient `height` method.

WARNING
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 $\mathcal{O}(n)$, which means it's not suitable for our purposes. To maintain efficiency, you'll need to update the height of each node along the insertion path from the bottom up. Height of a node is the height of the subtree rooted at that node.
- You are provided with a test suite in `AVLTest.java`. Use `./gradlew test` often and pass tests for each task before moving onto the next.

5 Enhancements (for extra credit)

Enhancements and git The base project will be graded based on the `main` branch of your repository. Before you change your code in the process of completing enhancements, create a new branch in your repository

```
$ 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 `main` and `enhancements` branches to GitHub before the submission deadline.

To work on the `main`, check out the branch with the following command:

```
$ git checkout main  # git checkout <branch-name>
```

To learn more about git branching, check the tutorial here and here.

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

1. (1 pt) Implement `remove`, maintaining AVL balance.
2. (1 pt) 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 pt) 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 your A2 writeup, which you'll commit in your repository (see Section 7)

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.

- BST insertion
- rotations
- rebalance
- AVL insertion and main program behavior
- (optional) Any enhancements
- A2 survey

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 Canvas Discussions if you are stuck.

7 How and what to submit

Submit the assignment by pushing your repository changes to GitHub and by completing the A2 survey on Canvas.

Your repository will be graded based on the **last commit before the deadline**. We will pull commits of your repository right after the deadline. So do not forget to commit and **push your work before the deadline**. Your repository should have these things:

- Code changes for the tasks given above.
- Writeup: You are given a `writeup.md` file in the repository with the following prompts. Update the writeup with your responses.
  - Declare/discuss any aspects of your code that are not working. What are your intuitions about why things are not working? What issues you already tried and ruled out? Given more time, what would you try next? Detailed answers here are critical to getting partial credit for malfunctioning programs.
  - In a few sentences, describe how you tested that your code was working.
  - What was the most challenging aspect of this assignment, and why?
  - What variant/extension of this assignment would you like to try (e.g., a variant that is more powerful, more interesting, etc).
  - If you did the enhancemets, explain what you did, how you did it, and instructions for testing your enhancements.

Rubric

You can earn points for the correctness and efficiency of your program, and can lose points for errors in commenting, style, clarity, and following assignment instructions. A2 is out of a total of 50 points.

> IMPORTANT
> Read this section carefully. There are three new deduction criteria (code compilation, git versioning, and writeup). Not meeting these requirements are grounds for point deductions (see other deductions below).

- **Submission**
  - (1 pt) Code is pushed to github and hours are reported in A2 Survey.
- **Code : Correctness**
  - (33 pts) Unit tests (1.5 pts per test)
  - (6 pts) Main program prints the correct number of unique lines
- **Code : Efficiency**
  - (5 pts) `avlInsert` maintains $\mathcal{O}(\log n)$ performance by keeping track of node heights and updating them as necessary
  - (5 pts) `Unique` processes a document with n words in $\mathcal{O}(n \log n)$ time

- **Clarity deductions** (up to 2 pts each)
  - Include author, date and purpose in a comment at the top of each file in which you write code
  - 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
- **Other deductions** (up to 5 each)
  - Code does not compile.
  - Not enough git versioning (less than 5 commits/member). Your repository should have at least five commits from each member of the team. You should be committing often as you work on the assignment.
  - Incomplete or no writeup.

8 Acknowledgments

This lab is based on materials developed and refined by Tanzima Islam, Brian Hutchinson, Filip Jagodzinski, Qiang Hao, Scott Wehrwein, and several past TAs.