Writeup for second project of
CMSC 420: "Data Structures"
Section 0101, Spring 2019

Theme: AVL-G trees

Handout date: February 12, 2019
On-time deadline: March 4th

Late deadline (30% penalty): March 6th

1 Overview

Jason is tired of *the man* telling him that AVL trees should rotate whenever we detect an imbalance of **2 (two)** or **-2 (minus two)** in any given subtree! As we have seen in class, AVL trees are **excellent for search** given their height of $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor + 1$. Unfortunately, to achieve this efficient search, they have to use **rotations** during **insertions** and **deletions**, and those **take time**!

In this project, you will implement **AVL-G** trees, a simple modification of AVL Trees that allows for tuning the balance of an AVL Tree based on a constructor parameter. Simply put, an AVL-G tree, where $G = 1, 2, 3, \ldots$, allows for any given subtree to have a balance of **at most G**. This means that our classic AVL trees can be referred to as "AVL-1" trees. Refer to section 2 for examples.

2 AVL-G trees

Remind yourselves of our basic definitions:

- **Height** of a binary tree: The **height** of a binary tree is defined as follows:
  - A `null` (empty) binary tree has a height of -1.
  - A non-empty binary tree has a height equal to the maximum of the height of its subtrees + 1 (plus one). A corollary of this is that a "stub" binary tree (one that consists of a single "leaf" node without any children) has a height of 0 (zero).
- **Balance** of a node: The balance of a node is defined by convention as the height of its **left** subtree minus the height of its **right** subtree.

Figure 1 shows an imbalance detected after an insertion into an AVL-**1** tree ("classic" AVL tree). The imbalance is detected at the node that contains the key 40 and is equal to -2. If instead we had an AVL-**2** tree, this insertion would **not** have triggered a rotation, since -2 is an **admissible** balance metric for AVL-2 trees!

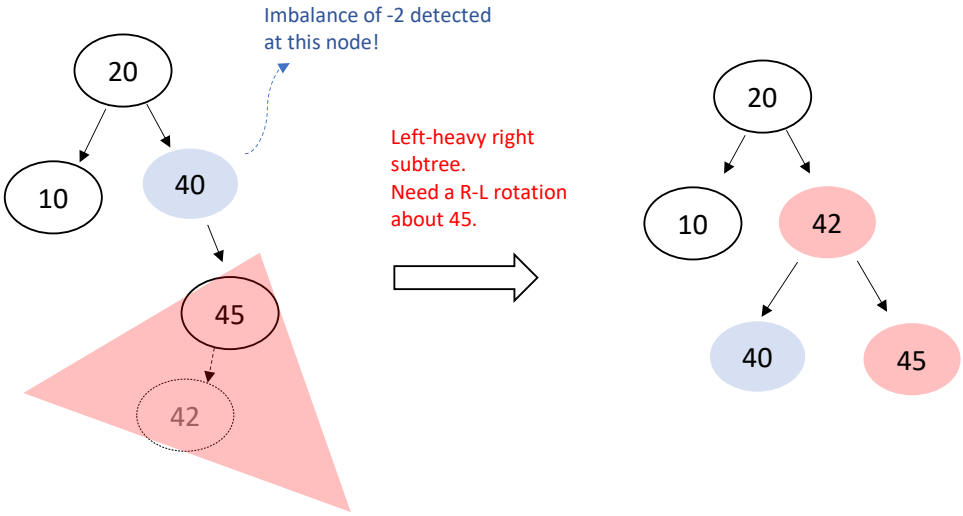

Figure 1: A rotation triggered by an insertion in an AVL-1 ("classic" AVL) tree.

An example of an imbalance and corresponding rotation in an AVL-2 tree is shown in Figure 2. Of course, this insertion would **not** have triggered a rotation in an AVL-**3** tree!

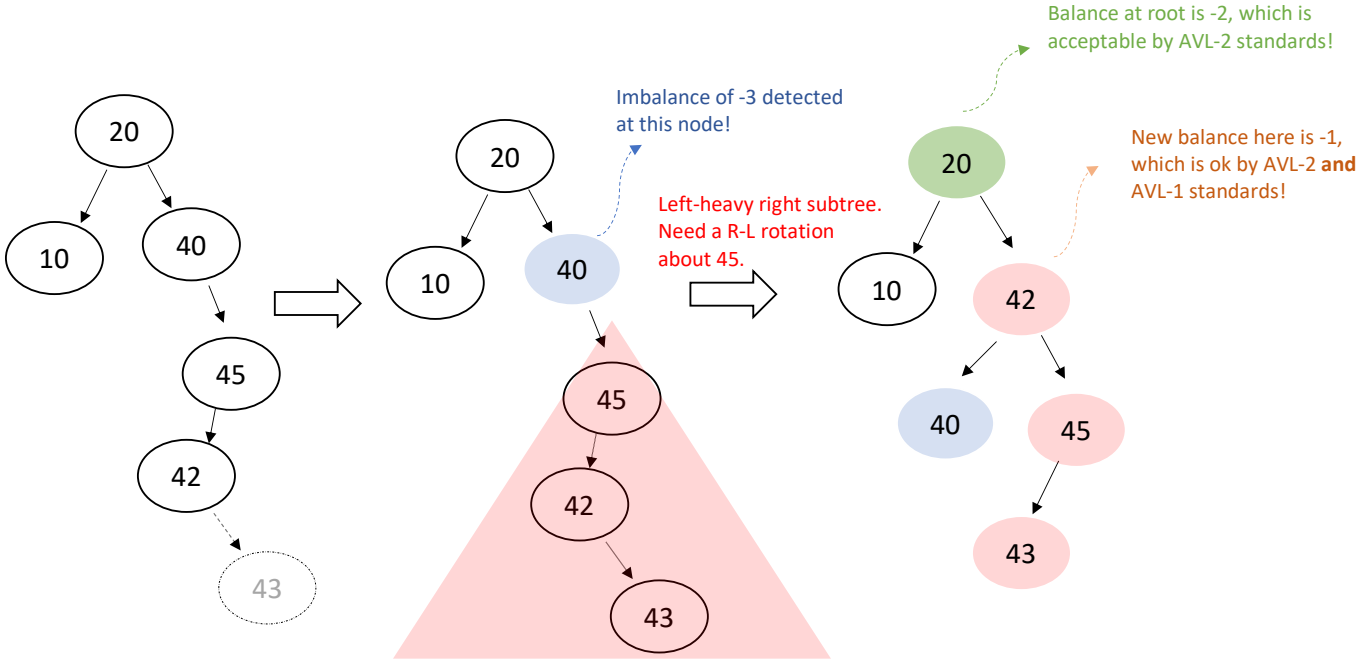

Figure 2: A rotation triggered by an insertion in an AVL-2 tree.

Note that we detect the imbalance at the node that contains 40. In order to rectify this imbalance, we need to perform a **R-L rotation** on the right subtree of this node! Notice something interesting here: in order to perform this R-L rotation, the first thing that we do is, well, a **R**ight rotation about 45. However, this leaves us with a binary tree that has 42 as its root, 45 as its right child and 43 as 45's left child. At first glance, this doesn't look "balanced". But, if we remember that we are in an AVL-**2** tree, **then we can realize that the form of this subtree after the first rotation is completely fine,** and the fact that this operation **would not be ok for an AVL-1** tree **doesn't mean anything** for our current **AVL- 2** tree!

An example of an imbalance detected in an AVL-**3** tree is shown in Figure 3. Note that this example is based on a **deletion**, just to provide you with a variety of examples. Remember than in deletions, when

a certain node detects an imbalance, it is the *opposite* subtree that will have to be mutated so that the entire tree is re-balanced!

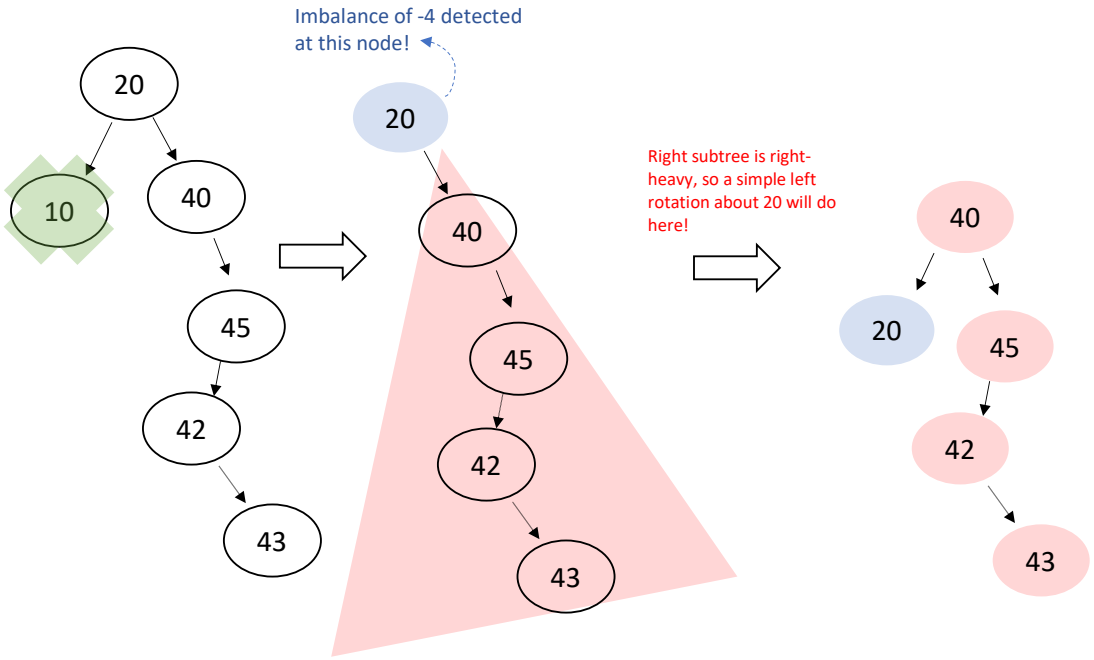

Figure 3: A rotation triggered by a deletion in an AVL-3 tree.

3 Starter Code and Git hints

Everything you need is available in our Github repository. The code for this project is specifically under the sub-package `projects.avlg`. You will only need to implement the class `AVLGTree`. You should read the docs ***very carefully***, and make sure you **do not** erase anything from the sub-package `projects.avlg.exceptions`. Those exceptions are used by the `AVLGTree` class and our tests.

Doing a `git pull` will bring the new version of the remote repository to your local git repository. It might be the case that you would need to resolve **merge conflicts**. You would then want to pull everything from the server **except for the skeleton code we provide for the first project**, since this could override your implementation. However, please do realize that your implementation has already been submitted on the submit server, so there is no way that code you have written will be lost.

The **easiest** way to resolve the merge conflicts would be: do a `git add -A` to put everything that the remote is giving you in your staging area, and then do a small number of `git reset`s, providing your files as arguments. The command `git reset <path_to_file>` will remove the file pointed to by `<path_to_file>` from the staging area, so that a new commit will **not** contain changes to these files. Then do a commit, and you're done. If you don't feel like doing a `git reset`, then you can grab your files from the submit server and just replace the "skeleton" graph project Java source files that the remote gives you under `projects.graph` with your own files, adjusting `import` and `package` statements accordingly.

Note that the submit server does **not care at all** about your implementation of the first project; it only cares about the class `projects.avlg.AVLGTree`. So whether you actually pull the skeleton code for the first project from our GitHub doesn't really matter, but you might want to do it that way so that you have the full course code base when we're done. As with the first project, if you would like to add classes, enums, interfaces, custom exceptions, extra packages, etc., you are **absolutely free** to do so, yet you should **not** move the class `AVLGTree` from its location in the code tree.

4 Advice / Hints

- The first thing you should do after you pull the code and read this writeup is to read the Javadocs for `AVLGTree` to understand its public interface. By "interface" here we do not refer to a Java interface, but to the way the class "interfaces" with its environment; it's API. Once you are certain you understand what the different methods receive as an argument, return to the caller, and **what kinds of Exceptions they throw**, start writing unit tests **before** you implement your code. As with the first project, we think it would be a **terrific** idea to write the tests for a classmate and have **them** write the tests for **you**. Try your best to make your classmate's code break!

- The core of this project is understanding rotations. When should we do a single left or a single right rotation? When is a combination of two rotations appropriate? How can I detect those separate cases in my code? How do I update heights during the rotations in the case of an **insertion** and how in the case of a **deletion**?

- The methods `isBST` and `isAVLGBalanced` that you have to implement are **awesome** for testing. Use them to your advantage! Just make sure that they are implemented correctly by unit-testing **them** as well!

- When you write unit tests for your class, you might find it frustrating that you have to wrap lots of pieces of code within `try - catch` blocks which sometimes seem redundant. For example, the constructor for `AVLGTree` throws an `InvalidBalanceException` if the user provides a balance parameter smaller than 1.[1] Even if you were to write the line of code `AVLGTree<String> mytree = new AVLGTree<>(2);`, the Java compiler will complain that you are not catching an instance of that particular exception, despite the fact that, clearly, $2 \geq 1$. Or, equivalently, if you are searching for a key in a tree that you **know** (or, at least, you **think** you know) **for certain** is **non-** empty, you will still have to wrap this piece of code with a `try - catch` block that attempts to deal with an instance of `EmptyTreeException`. In order to help yourselves out with writing cleaner unit-test code and learn more about how to handle Exceptions, please read the comments in the file `StudentTests.java`, which contains some starter tests which you can use for your implementation. You may also use **multi-catch blocks**, a construct introduced in Java 7. Here's an example from our `ReleaseTests`:

```
@Test
public void testBalancedInsertions(){
   try {
        /*
         * ******************************************************
         * 297 lines of code to which you are not privy...
         * ******************************************************
         */
        } catch(EmptyTreeException | InvalidBalanceException exc){ // Multi-catch block
          fail("Caught a "+ exc.getClass().getSimpleName() + " with message: " +
                       exc.getMessage() + ". How could this possibly happen?");
     }
}
```

[1] In OOP, it is an **excellent** idea to throw exceptions from a constructor, since there is no other way to tell the caller that something went bad during object construction!