Module 8 Assignment Part 2: AVL Operations

Marcar esta página

Coding Assignment fecha límite Oct 2, 2023 08:43 -05

AVL Operations

For part two of this assignment, you will be coding the **add()** and **remove()** methods of an AVL. Since trees are naturally recursive structures, each of these methods should be **implemented recursively**.

IMPORTANT:

- **You will be given unlimited attempts on this assignment, with no cooldown between submissions.**
- **Please run your code before each submission to ensure that there are no formatting errors! If there are formatting errors in your code, your code will not be graded and a submission attempt will be logged. For more information, please review the Vocareum overview below.**

AVLNode

An **AVLNode** class is provided to you and will be used to represent the nodes of the tree. This file should be treated as **read-only** and should not be modified in any way. This AVLNode class contains getter and setter methods to access and mutate the structure of the nodes. Please make sure that you understand how this class works, as interaction with this class is crucial for this assignment.

Pointer Reinforcement

Since both the add() and remove() methods may change the structure of the tree, we highly recommend that you use a technique called pointer reinforcement. Although using pointer reinforcement is not required, it will help to make your code cleaner, and it'll also help greatly in future assignments if you end up taking the next course in our series! Below is a video created by our 1332 TAs, timestamped to a section on pointer reinforcement.

Pointer Reinforcement Overview

Balancing

The tree should rotate appropriately to ensure that it is always balanced. A tree is balanced if every node's balance factor is either -1, 0, or 1. Keep in mind that you will have to update the balancing information stored in the nodes on the way back up the tree after modifying the tree; the variables are not updated automatically.

NOTE: If you have completed part one of this assignment, then you should simply **copy and paste** your code for updateHeightAndBF, rotateLeft, rotateRight, and balance, into this assignment. Note that you may have to toggle the method visibilities to private.

Comparable

As stated, the data in the AVL must implement the Comparable interface. As you'll see in the files, the generic typing has been specified to require that it implements the **Comparable** interface. You use the interface by making a method call like **data1.compareTo(data2)**. This will return an **int**, and the value tells you how **data1** and **data2** are in relation to each other.

- If the int is **positive**, then data1 is **larger** than data2.
- If the int is **negative**, then data1 is **smaller** than data2.
- If the int is **zero**, then data1 **equals** data2.

Note that the returned value can be **any** integer in Java's int range, **not** just -1, 0, 1.

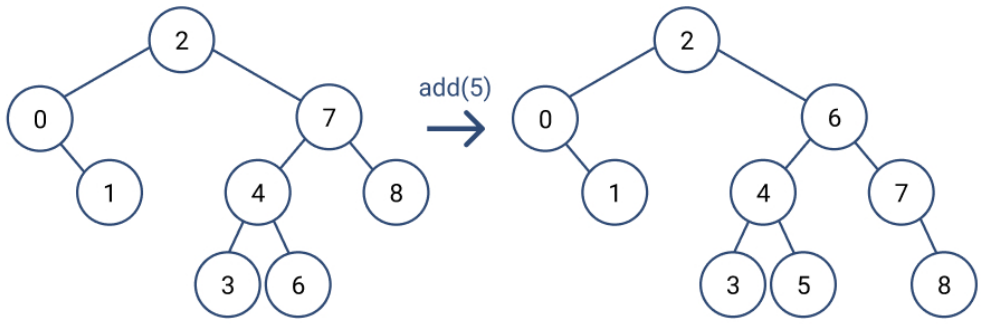
Successor

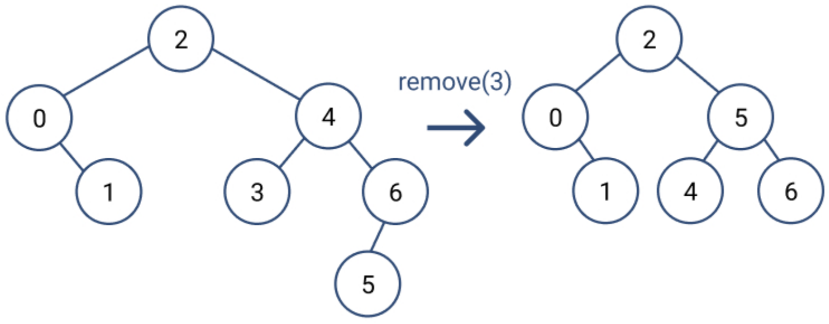
Recall that earlier in the modules you learned about the successor and predecessor of a node in a tree. As a refresher, the successor of a node, n, is the node in the tree that contains the smallest data that is larger than n's data. The predecessor of a node, n, is the node in the tree that contains the largest data that is smaller than n's data. When removing a node from an AVL that has two children, we can choose to replace the removed node with either it's successor or predecessor. For the 2-child case in remove(), you will be replacing the removed node with its **successor node**, **NOT** the predecessor node. For more details, please refer to the javadocs for remove().

Helper Methods

You'll also notice that the public method stubs we've provided do not contain the parameters necessary for recursion to work, so these public methods should act as "wrapper methods" for you to use. You will have to write private recursive helper methods and call them in these wrapper methods. All of these helper methods **must be private**. To reiterate, do **not** change the method headers for the provided methods.

Operation Examples:





General Tips

- Don't forget your base case; this should be the first thing that is checked in your recursive calls. Note that the base case for add() may be different that the base case in remove().

- If you get stuck on remove(), the video above on pointer reinforcement has pseudocode for the remove() method for a BST, which will be very similar to remove() in an AVL! All you'll have to do is balance the node before you return it in your recursive helper method.

- We highly recommend copying the starter code and working in your preferred IDE in order to have access to features such as code completion, auto-formatting, and much more!

Here are general assignment guidelines that should be followed.

- Do not include any package declarations in your classes.
- Do not change any existing class headers, constructors, instance/global variables, or method signatures. For example, do not add throws to the method headers since they are not necessary. Instead, exceptions should be thrown as follows: `throw new InsertExceptionHere("Error: some exception was thrown");`

- All helper methods you choose to write should be made private. Recall the idea of Encapsulation in Object-Oriented Programming!
- Do not use anything that would trivialize the assignment. (e.g. Don't import/use java.util.ArrayList for an ArrayList assignment.)
- Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is extremely rare).
- If applicable, use the generic type of the class; do not use the raw type of the class. For example, use `new LinkedList<Integer>()` instead of `new LinkedList()`.

Use of the following statements should be avoided at all times.

package	System.arraycopy()	clone()
assert()	Arrays class	Array class
Thread class	Collections class	Collection.toArray()
Reflection APIs	Inner or nested classes	Lambda Expressions

The Vocareum (code editor) interface has six main components:

- The **Drop-Down** in the top left. This lets you choose from multiple available files. Note that this drop-down will only be visible in assignments that require multiple files.
- The **Run** button. This will compile your code and run a file scan. Running your code will not count towards your total allowed submission attempts, therefore you are free to run as many times as needed.
- The **Submit** button. This will compile your code, run a file scan, grade your assignment, and output results to console. Note that for most assignments in this class, you will only be allowed a limited number of submissions. A submission is counted when the submit button is clicked, regardless of whether or not your code can compile or if there are any file issues. Therefore, we **highly recommend** that you run your code before submitting to ensure that there are no issues that will prevent your code from being graded and that every submission attempt will generate meaningful results.
- The **Reset** button. This will revert all your changes and reset your code to the default code template.
- The **Code Window**. This is where you will write your code. For large coding assignments, we highly recommend copying the starter code and working in your preferred IDE to have access to features such as code completion, auto-formatting, and much more!
- The **Output Window**. This window will appear whenever you run or submit your code and will display the output for you to view.

For additional help, please visit the Vocareum information page located in the course information module!

< Anterior   Siguiente >