Object Oriented Programming - Exercise 4: AVL Trees

Contents

1 Goals

1. Implementing a data structure you have learned about in the Data Structures course - the AVL Tree.

2. Experimenting with this data structure.

2 Submission Details

- Submission Deadline: **Wednesday, 23/5/2018, 23:55**
- This exercise will be done **in pairs**.
- You may use the `java.util.List<T>` interface and two of its implementations – `java.util.LinkedList<T>` and `java.util.ArrayList<T>`. You may also use the `java.lang.Iterable<T>` interface and the `java.util.Iterator<T>` class, the `Math` class and any `Exception` classes you want. You **may not** use any other class that you didn't write yourself.
- As always, be sure to be updated on questions asked in the discussion forums.

3 Introduction

An AVL tree is a self-balancing binary search tree. AVL trees maintain the *AVL property* - for each node, the difference between the heights of both of its sub-trees is at most 1.[1] This is done by re-balancing the tree after each insertion and deletion operation. You've recently learned about the theoretical background of AVL trees in the Data Structures (DAST) course.[2]

In this exercise, you will implement an AVL tree of integers based on the pseudo-code shown in the DAST lecture and tirgul. Duplicate keys will not be allowed in the tree.

4 API

4.1 API

You are required to submit a class named `AvlTree` that implements `Iterable<Integer>` and the following API (also found at http://www.cs.huji.ac.il/~oop/ex4/oop/ex4/data_structures/AvlTree.html).

You are allowed (and encouraged!) to write and submit more class(es) as you see fit. You can also add methods of your own but make sure that you don't change the supplied API; you are only allowed to add methods with the `private` and the `default-package` access modifier.

4.2 Package

Both `AvlTree` and any other class you implement in this exercise should be placed in the `oop.ex4.data_structures` package. Make sure the package heirarchy is correct, i.e, create a folder name oop, in it a folder name ex4 and in a folder name data_structures where your files will be placed.

[1] The height of a tree is defined as the length of the longest downward path from the root to any of the leaves.

[2] See https://moodle.cs.huji.ac.il/cs13/file.php/67109/tirgul8.pdf and http://en.wikipedia.org/wiki/Tree_rotation.

4.3 Methods

- ```
  /**
   * Add a new node with the given key to the tree.
   *
   * @param newValue the value of the new node to add.
   * @return true if the value to add is not already in the tree and it was successfully added,
   * false otherwise.
   */
  public boolean add(int newValue);
  ```

- ```
  /**
   * Check whether the tree contains the given input value.
   *
   * @param searchVal the value to search for.
   * @return the depth of the node (0 for the root) with the given value if it was found in
   * the tree, −1 otherwise.
   */
  public int contains(int searchVal);
  ```

- ```
  /**
   * Removes the node with the given value from the tree, if it exists.
   *
   * @param toDelete the value to remove from the tree.
   * @return true if the given value was found and deleted, false otherwise.
   */
  public boolean delete(int toDelete);
  ```

- ```
  /**
   * @return the number of nodes in the tree.
   */
  public int size();
  ```

- ```
  /**
   * @return an iterator for the Avl Tree. The returned iterator iterates over the tree nodes
   * in an ascending order, and does NOT implement the remove() method.
   */
  public Iterator<Integer> iterator();
  ```

4.4 Constructors

- ```
  /**
   * The default constructor.
   */
  public AvlTree();
  ```

- ```
  /**
   * A constructor that builds a new AVL tree containing all unique values in the input
   * array.
   * @param data the values to add to tree.
   */
  public AvlTree(int[] data);
  ```

- ```
  /**
   * A copy constructor that creates a deep copy of the given AvlTree. The new tree
   * contains all the values of the given tree, but not necessarily in the same structure.
   * @param avlTree an AVL tree.
   */
  public AvlTree(AvlTree avlTree);
  ```

  Comment: *A deep copy of the tree means that for every node or any other internal object of the given tree, a new, identical object, is instantiated for the new tree (the internal object is not simply referenced from it; i.e. points to the same internal node object in memory). You are not required to implement this method with the best possible worse-case complexity.*

4.5 Static Methods

- ```
  /**
   * Calculates the minimum number of nodes in an AVL tree of height h.
   *
   * @param h the height of the tree (a non−negative number) in question.
   * @return the minimum number of nodes in an AVL tree of the given height.
   */
  public static int findMinNodes(int h);
  ```

- ```
  /**
   * Calculates the maximum number of nodes in an AVL tree of height h.
   *
   * @param h the height of the tree (a non−negative number) in question.
   * @return the maximum number of nodes in an AVL tree of the given height.
   */
  public static int findMaxNodes(int h);
  ```

  Comment: *You are required to implement this method with a running time complexity that is at most polynomial in h, and uses $O(1)$ memory.*

For example, `findMinNodes(3)`, as shown in Figure 1, should return 7 `findMaxNodes(3)` should return 15.

4.6 Comments

1. There are many types of binary search trees: red-black trees, splay trees, treap, T-trees and more. A good programmer (that likes high grades) should think about these type of things

when planning a design, even if it is not currently demanded in the project. On the other hand, avoid extreme over-engineering and other anti-patterns. A design that can be easily changed to support future reasonable requirements will get better grades

2. The `AvlTree(int[] data)` constructor should construct a tree by creating an empty one and then adding the elements in the input array one by one. Note that if a value appears more than once in the list, only the first appearance is added.

3. The `add(int newValue)` method will return false and leave the tree unchanged if `newValue` already exists in it.

4. The data constructor can receive any array of `int` primitives, which means **you can't assume the array is sorted**. You also can't assume that you won't get a null reference.

5. You can assume `findMinNodes()` and `findMaxNodes()` receive a non-negative integer.

6. You are expected to throw exceptions when needed.

7. **Do not** paste code copied from the PDF file. It contains unwanted characters that will fail compilation.

8. A tree with only a single node is of height zero, or has $h = 0$. A tree with a root that has one or two sons (each of which has no sons) is of height one, or $h = 1$. Etc.

9. When writing a class implementing an interface only partially - opting not to implement some optional methods of the interface - you can throw an `UnsupportedOperationException` from the methods you wish to leave unimplemented. Can you figure out how throwing this specific exception does not violate the interface's API and method signature?

10. The return type of the `iterator()` method must be `Iterator<Integer>`, and not an iterator of any other type.

11. You may have `AvlTree` extend another class, but then you may not add new `public` or `protected` methods to this base class (as `AvlTree` will inherit any such method). In such a case, you may have the base class implement `Iterable<Integer>` instead of `AvlTree` itself.

12. You may also have `AvlTree` implement other interfaces. Make sure, though, that this does not entail extending the API, and that you can explain your design choices.

5 Theoretical Questions

Answer the following questions, and write your full answers in your README file:

5.1 Analyzing the AVL tree

In Figure 1 you can see an AVL tree of height 3. This tree may seem unbalanced, but is in fact a valid AVL tree resulted from a specific insertion order. Notice that this example shows a tree of height 3 with the minimal number of nodes (if you delete any node, the height of the tree will become 2).

1. Find a series of 12 numbers, such that when they are inserted into an empty AVL tree one-by-one, the result will be a tree of height 4 (insertions only, no deletions).

Hint: We suggest you start by figuring out the order in which the nodes were inserted into the example tree, and then continue to create the larger one.

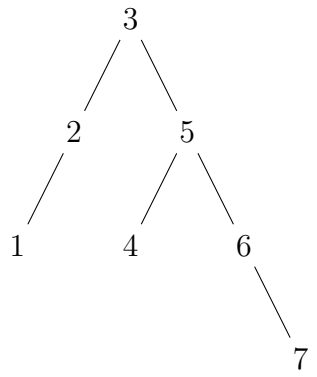

Figure 1: an AVL tree of height 3 with the minimal number of nodes

5.2 The complexity of constructing an AVL tree

1. What is the *asymptotic* running time complexity of the tree construction done in `AvlTree(int[] data)`?

2. **Bonus:** What is the best possible running time complexity for a specific case?

5.3 The complexity of copying an AVL tree

1. What is the *asymptotic* running time complexity of the tree construction done in `AvlTree(AvlTree avlTree)`?

2. What is the best possible running time complexity for a specific case? (this is not a bonus question!)

5.4 The complexity of calculating the minimal number of nodes in an AVL tree of a given height

1. What is the asymptotic running time complexity of *your* implementation of the operation done in `findMinNodes(int h)`?

2. Is that the best possible *asymptotic* running time complexity? If not, what is the best achievable complexity?

6 Submission Requirements

6.1 README

Please address the following points in your README file:

1. Describe which class(es) (if any) you wrote as part of your implementation of an AVL tree, other than `AvlTree`. The description should include the purpose of each class, its important methods and its interaction with the `AvlTree` class.

2. Describe your implementation of the methods `add`() and `delete`(). The description should include the general workflow in each of these methods. You should also indicate which helper methods you implemented for each of them, and which of these helper methods are shared by both of them (if any).

3. Your README file should also answer the theoretical questions.

6.2 Submission Guidelines

You should submit a file named ex4.jar containing all the `.java` files of your program, as well as the README file. Please note the following:

- Only one of you should submit the JAR file. **Make sure both of your details are written in the README file.**

- Files should be submitted in the original directory hierarchy of their packages. **Make sure the names of all folders and the name of the package is correct before submitting.**

- No `.class` files should be submitted.

- Your program must compile without any errors or warnings. IDEA displays most warnings in the editor. In order to also have them displayed when you compile your code, go to `File->Settings->Build, Execution, Deployment->` *Compiler*`->Java Compiler` and enter the following line under `Additional command line parameters`:
  `-Xlint:rawtypes -Xlint:static -Xlint:empty -Xlint:divzero -Xlint:deprecation`
  If you want to check for them using the console, you can use the following customized compilation command:
  `javac -Xlint:rawtypes -Xlint:static -Xlint:empty -Xlint:divzero -Xlint:deprecation file1.java file2.java ...`
  Note: Copy-pasting the command above to the shell might result in some invalid characters.

- `javadoc` should compile without any errors or warnings.

You may use the following unix command to create the jar files:

```
jar -cvf ex4.jar README oop/ex4/data_structures/*.java
```

This command should be run from the main project directory (that is, where the `oop` directory is). Note the directory structures - yours should be exactly the same.

7 Presubmission Script

You can run the presubmission script from the shell using the following command:

`~oop/bin/ex4/presubmission/test.py ex4.jar`

(The same script is run when you upload your submission to Moodle.)

Good Luck!