

# Object Oriented Programming - Exercise 4: AVL Trees

## Contents

<b>1</b>	<b>Goals</b>	<b>2</b>
<b>2</b>	<b>Submission Details</b>	<b>2</b>
<b>3</b>	<b>Introduction</b>	<b>2</b>
<b>4</b>	<b>API</b>	<b>2</b>
4.1	Package . . . . .	2
4.2	Methods . . . . .	3
4.3	Constructors . . . . .	3
4.4	Static Methods . . . . .	4
4.5	Comments . . . . .	4
<b>5</b>	<b>Theoretical Questions</b>	<b>5</b>
5.1	Analyzing the AVL tree . . . . .	5
5.2	The complexity of constructing an AVL tree . . . . .	5
5.3	The complexity of copying an AVL tree . . . . .	6
5.4	The complexity of calculating the minimal number of nodes in an AVL tree of a given height . . . . .	6
<b>6</b>	<b>Submission Requirements</b>	<b>6</b>
6.1	README . . . . .	6
6.2	Automatic Testing . . . . .	7
6.3	Submission Guidelines . . . . .	7
<b>7</b>	<b>Technical Issues</b>	<b>8</b>
<b>8</b>	<b>Presubmission Script</b>	<b>8</b>
<b>A</b>	<b>Test Input Files Format</b>	<b>9</b>
A.1	Examples . . . . .	9
A.1.1	. . . . .	9
A.1.2	. . . . .	10
A.2	Some Clarifications . . . . .	10

# 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: **Monday, 9/5/2016, 23:55**
- This exercise will be done alone (**not in pairs**).
- You may use the `java.util.List<T>` interface and two of its implementations – `java.util.LinkedList` 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.

# 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.<sup>1</sup> 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.<sup>2</sup>

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

You are required to submit a class named `AvlTree` that implements `Iterable<Integer>` and the following API.

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.

## 4.1 Package

Both `AvlTree` and any other class you implement in this exercise should be placed in the `oop.ex4.data_structures` package.

---

<sup>1</sup>The height of a tree is defined as the length of the longest downward path from the root to any of the leaves.

<sup>2</sup>See <http://moodle.cs.huji.ac.il/cs13/file.php/67109/tirgul8.pdf>.

## 4.2 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();
```

## 4.3 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 worst-case complexity.*

## 4.4 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);
```

**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

## 4.5 Comments

1. 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.
2. The `add(int newValue)` method will return false and leave the tree unchanged if `newValue` already exists in it.
3. 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.
4. You can assume `findMinNodes()` receives a non-negative integer.

5. **Do not** paste code copied from the PDF file. It contains unwanted characters that will fail compilation.
6. 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.
7. 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?
8. The return type of the `iterator()` method must be `Iterator<Integer>`, and not an iterator of any other type.
9. 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.
10. 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

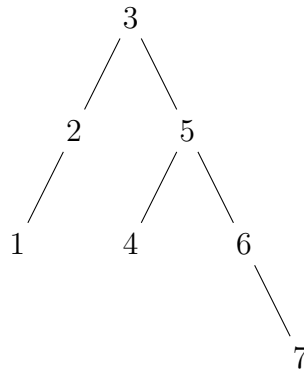
In Figure 1 you can see an AVL tree of height 3. This tree may seem unbalanced, but is in fact a valid AVL tree resulted from a specific insertion order. Notice that this example shows a tree of height 3 with the minimal number of nodes (if you delete any node, the height of the tree will become 2).

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.

### 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?



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

### 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 question from Section 5.

## 6.2 Automatic Testing

Our automatic tests will do the following:

1. Make sure your code compiles and contains a README file.
2. Test each of your methods on simple inputs, in order to test their basic functionality.
3. Test your tree on the four interesting AVL scenarios (i.e., the four rotation cases).<sup>3</sup>
4. Test your code on more complicated scenarios that include large inputs and multiple add/delete operations.
5. As always, we will run unseen tests which weigh 10% of the automatic testers.

We will compile your class with our own tester and run a set of tests. Each such test is defined by an input file, which creates an AVL tree and performs a set of operations on it. The format of the input files is described in Appendix A.

## 6.3 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:

- Files should be submitted in the original directory hierarchy of their packages.
- No `.class` files should be submitted.
- There is no need to submit any testers.
- 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).

---

<sup>3</sup>An explanation in Hebrew of the four cases can be found here: [https://he.wikipedia.org/wiki/%D7%A2%D7%A5\\_AVL](https://he.wikipedia.org/wiki/%D7%A2%D7%A5_AVL).

## 7 Technical Issues

A file named `ex4_files.zip` can be found in the course website. This zip contains the test files on which your program will be tested. When you submit your exercise, we will run your code against these input files (as well as others), and you will receive a response file with the tests you failed (if any). Error messages will contain some information about the problem. For example, you may receive the following error:

```
runTests[17](Ex4Tester):  ex4/tests/020.txt (# simple right-left rotation after delete):  
line number 7:  expected:<true> but was:<false>
```

This means that your program failed the test in file `020.txt`, when trying to execute the delete operation. More specifically, line 7 of the test failed at this stage because the return value of calling `delete()` was true instead of false (details in appendix A).

## 8 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!



# Appendices

## A Test Input Files Format

Each of the automatic tests is defined in a file of the following format:

- The first line starts with a ‘#’ sign, and is a comment describing the test (i.e., this line is ignored).
- The second line creates an `AvlTree` object (denoted `myTree`) using one of the constructors defined in Section 4.3:
  - An empty line calls the default constructor.
  - A list of integers (e.g., `1 5 7 12 4`) calls the data constructor with this list of integers.
- The following lines specify a call to one of the methods of the created tree, and are executed sequentially.

They can be one of the following:

- *add number return\_value*  
call `myTree.add(number)`. Verify that the method’s return value matches *return\_value* (either `true` or `false`).
- *delete number return\_value*  
call `myTree.delete(number)`. Verify that the method’s return value matches *return\_value* (either `true` or `false`).
- *contains number return\_value*  
call `myTree.contains(number)`. Verify that the method’s return value matches *return\_value* (a non-negative integer or -1).
- *size return\_value*  
call `myTree.size()`. Verify that the method’s return value matches *return\_value* (a non-negative integer).
- *copy*  
call `myTree2 = newAvlTree(myTree)`. Verify that the copy constructor functions correctly verifying the trees contain the exact same values .
- *minNode number return\_value*  
call `AvlTree.findMinNodes(number)`. Verify that the method’s return value matches *return\_value*.

### A.1 Examples

#### A.1.1

Consider the following example test file:

```
# a simple creation of a tree and adding the number 5
```

```
add 5 true
size 1
```

In this example, the first line is a comment describing the test. The second line is empty, which means calling the empty constructor. The third line calls `myTree.add(5)`, and then verifies that the command succeeded (by verifying that the return value of the `add` method is `true`). The last line verifies that the tree size is now 1 (after adding a single element).

### A.1.2

Consider another example:

```
# calling the data constructor with the values (1 2 3) , deleting 3, and then searching for 4
1 2 3
delete 3 true
contains 4 -1
```

The first line is, again, a comment describing the test. The second line is a list of numbers, which means calling the data constructor with the integer array (1 2 3). The third line calls `myTree.delete(3)`, and then verifies that the command succeeded. The last line calls `myTree.contains(4)` and verifies that 4 is not found in the tree (return value should be -1).

## A.2 Some Clarifications

Please note the following:

- Your code passes a given test only if your methods return the correct values in **each** of the method calls.
- A test file may include numerous calls to `add`, `delete`, `contains` and `size`.