Exercise 3 – AVL Tree

CLIDS 2013

May 3^{rd}, 2013

1 Goals

1. Implementing a data structure learned in class (AVL Tree)
2. Experimenting with this data structure

2 Submission Details

- Submission Deadline: **Thursday, 16/5/2013, 23:55**
- This exercise will be done alone (**not in pairs**).
- Note: In this exercise you may use classes and interfaces from the *java.lang* package in standard java 1.7 distribution. You may also use the *java.util.List* interface and two of its implementations – *java.util.LinkedList* and *java.util.ArrayList*. 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 property that for each node, the difference between the heights of both its sub-trees is at most 1.[1] This is done by re-balancing the tree after each insertion and deletion operation. You 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, based on the pseudo-code shown in the DaSt lecture.

4 API

You are required to submit a class called *AvlTree* that implements the following API. You are allowed (and encouraged!) to write and submit more class(es) as you see fit.

4.1 Package

Both *AvlTree* class and any other class you implement in this exercise should be placed in the *clids.ex3.data_structures* package.

4.2 Methods

- ```
  /**
   * Add a new node with key newValue into the tree.
   *
   * @param newValue new value to add to the tree.
   * @return false iff newValue already exist in the tree
   *
   */
  public boolean add(int newValue);
  ```

[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 `http://moodle.cs.huji.ac.il/cs12/file.php/67109/tirgul7new.pdf`.
  ```

- ```
  /**
   * Does tree contain a given input value.
   *
   * @param val value to search for.
   * @return if val is found in the tree, return the depth of its node (where 0 is the root).
   * Otherwise – return -1.
   */
  public int contains(int searchVal);
  ```

- ```
  /**
   * Remove a node from the tree, if it exists.
   *
   * @param toDelete value to delete
   * @return true iff toDelete is found and deleted
   */
  public boolean delete(int toDelete);
  ```

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

4.3 Constructors

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

- ```
  /**
   * A data constructor –
   * a constructor that builds the tree by adding the elements in the input array one-by-one.
   * If the same value appears twice (or more) in the list, it is ignored.
   *
   * @param data values to add to tree
   */
  public AvlTree(int[] data);
  ```

5 Analyzing the AVL Tree

AVL trees are considered balanced trees. For each node, it is guaranteed that the heights of both its sub-trees are (almost) the same. However, there are cases where for a specific node, one sub-tree has substantially more nodes than the other sub-tree (although the height difference property still holds).

Find a series of 20 numbers, such that when they are inserted into an empty AVL tree one-by-one, the result is a tree where the left sub-tree of the root has 15 nodes, while the right sub-tree of the root has only 4 nodes (insertions only, no deletions).

Hint: Figure 1 shows an example of an AVL tree where the left sub-tree of the root has 3 nodes while the right sub-tree has only one node. We suggest you start by figuring out in what order were the nodes inserted into this tree, and then continue to create the larger tree.
  ```

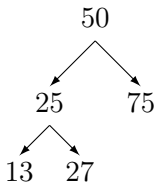

Figure 1: a (not so) balanced AVL tree

5.1 Bonus

In this exercise, you have an opportunity to get a bonus to your exercise grade, by implementing the following method:

```
/**
 * A method that calculates the number of nodes in an AVL tree of height h,
 * with the maximal difference between the number of nodes in the left sub-tree
 * and right sub-tree of the root.
 *
 * @param h height of the tree (a non-negative number).
 * @return maximal difference between number of nodes in the left sub-tree
 * and right sub-tree of the root (a non-negative number).
 */
public static int findMaxDifference(int h);
```

For example, calling *findMaxDifference*(2) should return 2 (as the example in Figure 1 is the AVL tree of height 2 with the maximal difference between the number of nodes in the left and right sub-trees of the root (3 - 1 = 2). Similarly, if you solve the question presented above, you will see that *findMaxDifference*(4) = 15 - 4 = 11.

If you implement this method using less than 20 lines of code, **and** in complexity of $O(h)$, you will get up to 5 points in the exercise grade.

6 Submission Requirements

6.1 README

Please address the following points in your README file:

1. Describe which class(es) (if any) you built in order to implement your AVL tree, other than AvlTree. The description should include the purpose of each class, its main 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 functions did you implement for each of them, and which of these helper functions are shared by these two methods (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. This includes verifying that your *AvlTree* class follows the API defined in Section 4 (including the correct package names).
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 settings (i.e., the four rotations cases).
4. Test your code on complicated settings that include large inputs and multiple add/delete operations.

In order to run our tests, we created a class called *TestEx3*, which uses an external *AvlTree* class. Will will compile this class with each of the *AvlTree* classes submitted by the students. This class runs a set of tests on the *AvlTree* class it is compild with. Each such test if 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 ex3.jar containing all the *.java* files of your program, as well as the README file. Please note the following:

- Files should be submitted along with their original packages.
- No *.class* files should be submitted.
- Your program must compile without any errors or warnings.
- javadoc should accept your program and produce documentation.

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

*jar –cvf ex3.jar README clids/ex3/data_structures/*java*

This command should be run from the main project directory (that is, where the *clids* directory is found).

7 Misc.

7.1 Grading

- Working according to Coding style guidelines – 10%
- Analyzing the AVL tree (see Section 5) – 10%
- Questions 1,2 in the README file section (see Section 6.1) – 10%
- Software design and implementation details – 20%
- Correct implementation – 50%

7.2 Forums and Questions

See previous exercises.

Good Luck!

Appendices

A Test Input File Format

We will run automatic tests on your code. In this appendix, we describe the format of these tests. The general testing framework is very similar to the one used in ex1. Each of the automatic test is incorporated 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 next lines call the methods of the created tree. The lines are called one after the other. 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).

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(5)* and verifies that 5 is 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*. However, only a single tree is constructed in every test.
- The file that reads the input files and follows their instruction, as well as the test files themselves, are supplied by the course staff.

A.3 Technical Issues

A file called ex3_files.zip is found in the course website. The zip contains the test files, on which your program will be tested. If any problem was discovered during the test (i.e., your program returned a different value than expected in one of the method calls, or an exception was thrown and not caught), an error message will be printed to the screen. Otherwise, if you passed successfully all of the tests, "*Perfect!*" will be printed.

As in ex1/2, after you submit your exercise, we will run your code against all the input files, and you will receive an email with the tests you failed on (if any). Error messages will contain the test file name. The format is similar to the one used in ex1.

For your convenience, you may (and are advised to!) run the automatic tests from the terminal by running the following command:
$\sim clids/bin/ex3AutoTests.sh \quad ex3.jar$
(where $ex3.jar$ is your jar file)