Contents

1 Basic Test Results

```
1  ******************** OOP pre-submission script for ex4 ********************
2
3  ***** Extracing Jar *****
4  Extracted successfully
5
6  ***** Compiling *****
7  Code complied successfully
8
9  ***** Running Tests *****
10 tests output :
11         Perfect!
12
13 ***** Checking format on QUESTIONS file *****
14 QUESTIONS file has the right format.
```

2 README

```
noa5

============================
=      File description    =
============================

AvlTree - Implements an AVL tree.

TreeNode - Implements a member of a tree (of an AVL tree in our case)

***
JAR file includes also:
README (this file)
QUESTIONS
***

============================
=          Design          =
============================

I Implemented the tree as was defined, and added methods that will 'support'
the main methods (such as 'findNode' and 'findSuccessor'), and keep the tree in
AVL state (such as 'updatetHeight' and 'balanceTree').
Each member of the tree is of AvlNode type.
In addition, I used a nested/inner class in order to implement 'iterator'.


* Remark - some of the helper methods i created, get a TreeNode x - usually
when I call them, i sent the root of the tree, and not some inner node in the
tree, in order to make the methods easy to change, or more accessible to other
purposes in the future (that way, there will be less code reusing, or several
methods that are different, but act similarly.


====================================
=  Implementation of other classes  =
====================================

* TreeNode - I decided to implement the members of the AVL tree as TreeNode
a class with a Default access modifier - meaning, visible only from the same
package. The reason for this, is because I implemented the nodes with data
members that are connected to binary search trees in general (and probably to
more trees), and not specifically to an AVL tree. Since AvlTree is in
'data_structures' package, I thought it makes sense to create more general
class of the AVL tree members - which can save time in the future when
implementing other data structures that also conclud nodes.
In conclusion, the purpose of TreeNode is to represent a member of an AVL tree
(also shows the interaction).
We can see the direct connection between the two classes, with the 'height'
data member of TreeNode - It helps us to implement the most important character
in AVL tree - the balance; The interaction between the two classes is
direct and strong, when there many cases I use the data members of a tree node
from the AvlTree class.
   ----  TreeNode has only constructor, and no methods ---

* Iterator- I used an inner/nested class to implement the iterator.
purpose, important methods and interaction with AvlTree are clear (since
are defined in the unstructions).

```

```
60
61  ================================================
62  =  Implementation of 'add' and 'delete' methods =
63  ================================================
64
65  * 'add' - First, check whether the root of the tree is null.
66  If so, then I point it to a new TreeNode with the given value, set the height
67  to '0' and the size to '1', and return true. In this phase, there is no need in
68  update of the heights (since there are not other nodes) or balancing.
69  If is not, then we call the helper method 'findNode' (will be detailed later),
70  in order to get the node that one of its sub trees will point to the given
71  value. In case the data of the node we got equals to the given value, then
72  the value is already exists in the tree, thus we return false (there are not
73  duplicate values in the tree).
74  After adding the given value, I use the helper method 'updatetHeight' (will be
75  detailed later), in order to update all of the heights of the nodes in the
76  tree.
77  Now, there is a chance that the tree is not balanced anymore (in terms of AVL),
78  So call the helper method 'balanceTree' (will be detailed later) to fix the
79  violation if is occured.
80  In the end, I update the size by 1, and return true.
81
82  * 'delete' - After checking whether the root of the tree is null (if so, return
83  false), we call the helper method 'findNode' (will be detailed later),
84  in order to get the node with the given value. In case the data of the node we
85  got is different than the given value, then the tree does not contain the given
86  value and I return false.
87  Then, I delete the node with the given value, by calling the helper method
88  'removeNode' (will be detailed later). In case the node we have found has two
89  subtrees, we find its successor by calling the helper method 'findSuccessor'
90  (will be detailed later), and then call 'removeNode' again (we have seen in
91  'Dast' that the successor has only one child/ sub tree).
92  After deleting, I use the helper method 'updatetHeight' (will be detailed
93  later), in order to update all of the heights of the nodes in the tree.
94  Again, there is a chance that the tree is not balanced anymore (in terms of
95  AVL), So call the helper method 'balanceTree' (will be detailed later) to fix
96  the violation if is occured.
97  In the end, I update the size (subtract by 1), and return true.
98
99
100 ***************
101 helper methods:
102
103 'findNode' - Returns a node with an equal value to the given one, or a node
104 that should get the given value in one of its sub trees.
105 This method is shared by both 'add' and 'delete'.
106
107
108 'updateHeight' - Updates all of the heights of the nodes that are 'under' the
109 given node, including its height.
110 This method is shared by both 'add' and 'delete', and the 'given node' the
111 method gets from 'add' and 'delete', is the root of the tree.
112
113
114 'balanceTree' - Checks if balancing is needed - if so, calls the rotations
115 methods.
116 This method is shared by both 'add' and 'delete'.
117
118
119 'removeNode' - Returns true if succeded removing the wanted node, false
120 otherwise (for example if the given node has two sub trees).
121
122
123 'findSuccessor' - Returns the successor of a given node in the tree.
124 ***************
125
126
```

```
127
128 ====================================
129 =  Answer for question in Section 5  =
130 ====================================
131
132 Example for such series: (6,4,5,3,11,8,7,9,10,2,1,12)
133
134 After inserting '5', there will be executed a LR rotation (violation is on '6')
135
136 After inserting '8', there will be executed a RL rotation (violation is on '6')
137
138 After inserting '10', there will be executed a LR rotation
139 (violation is on '11')
140
141 After inserting '2', there will be executed a LL rotation (violation is on '4')
142
143 After inserting '1', tree is of height 3. Adding '12' changes the height to 4.
```

-2/-8 The numbers you supplied don't generate a tree with height of 4 and 12 nodes.' (code='analyzing_problem') very complicated example that requires many rotations, you could fine an example with no rotations at all.

3 QUESTIONS

```
  #Week 6 Questions
#============

#Map<A,Integer> myMap = new HashMap<>();
#class A { }
#class B extends A { }
#Select all lines that will pass compilation
#1: myMap.put(new A(), new Integer(6));
#2: myMap.put(new B(), new Integer(6));
#3: myMap.put(new B(), 6);
#4: myMap.put(new Object(), new Integer(6));

w6Answer1 = (1,2,3)
# The answer should be a python-tuple of the chosen numbers, i.e. (1,7,3,8) or (2,6,1)
# optional explanation.

#public class Number implements Comparable<Number> {}
#ComplexNumber num1 = new ComplexNumber(5,2);
#ComplexNumber num2 = new ComplexNumber(4,3);
#int compareResult = num1.compareTo(num2);
#What are the declarations of the ComplexNumber class for which the above code will compile? (multiple choice)
#1: public class ComplexNumber implements Comparator<ComplexNumber> {}
#2: public class ComplexNumber implements Comparable<ComplexNumber> {}
#3: public class ComplexNumber extends Number<ComplexNumber> {}
#4: public class ComplexNumber extends Number {}

w6Answer2 = (2,4)
# The answer should be a python-tuple of the chosen numbers, i.e. (1,7,3,8) or (2,6,1)
# optional explanation.

#Choose all correct statements:
#1: A class can have only a single Comparator.
#2: The implementation of compareTo(A a) should be consistent with that of equals() in the class A.
#3: Comparators have access to private data members.
#4: Only the implementation of the Comparable<A> can be used to determine the order of elements in a TreeSet<A>.
#5: A Comparator<A> can be used to order objects of the type A, and of types extending A.
#6: Comparators allow us to change the comparison methods of some class in runtime.

w6Answer3 = (2,5,6)
# The answer should be a python-tuple of the chosen numbers, i.e. (1,7,3,8) or (2,6,1)
# optional explanation.


#Week 7 Questions
#============

#public void doAction() {
#    doFoo();
#    doBar();
#}
#
#For every checked exception thrown by the methods doFoo() and doBar(), the above method doAction()
# must either catch it or explicitly declare it as thrown.
#1: Correct
#2: Incorrect

w7Answer1 = 1
# optional explanation.

```

```
# class ExceptionA extends Exception {}
# class ExceptionB extends ExceptionA {}
# class ExceptionC extends ExceptionA {}
#Which of the following catch clauses will NOT catch an exception of type ExceptionB? (single choice)
#1: catch (Exception e) {...
#2: catch (ExceptionA e) {...
#3: catch (ExceptionB | ExceptionC e) {...
#4: catch (ExceptionC e) {...

w7Answer2 = 4
# optional explanation.

#Which of the following is true? (single choice)
#1: An exception cannot include any data members.
#2: Unchecked exceptions cannot be caught by a try-catch clause.
#3: Unchecked exceptions cannot be thrown by the throw key-word.
#4: Methods can declare that they throw unchecked exceptions.

w7Answer3 = 4
# optional explanation.

#Which of the following is NOT true? (single choice)
#1: Packages can be used to group together related classes.
#2: Packages have no effect over access permission between different classes.
#3: A class doesn't have to extend class A to access its protected members.
#4: Packages can be used to solve conflicting class names.

w7Answer4 = 2
# optional explanation.
```

4 oop/ex4/data structures/AvlTree.java

```java
package oop.ex4.data_structures;
import java.util.Iterator;


/**
 * Implementation of an AVL tree - a self-balancing binary search tree.
 *
 * @author noa5
 *
 */
public class AvlTree implements Iterable<Integer> {

    // Represents the root of the AVL tree
    private TreeNode root;
    // Represents the number of AVL nodes the AVL tree has
    private int size;

    /**
     * The default constructor.
     */
    public AvlTree() {
        this.root = null;
        this.size = 0;
    }

    /**
     * A constructor that builds the tree by adding the elements in the input
     * array one by one. If a value appears more than once in the list, only the
     * first appearance is added.
     *
     * @param data the values to add to tree
     */
    public AvlTree(int[] data) {
        if (data == null) {
            return;
        }
        for (int item: data) {
            Integer value = item;
            if (value != null) {
                this.add(value);
            }
        }
    }

    /**
     * A copy constructor that creates a deep copy of the given AvlTree. This
     * 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). The new tree must contain all
     * the values of the given tree, but not necessarily in the same structure.
     *
     * @param avlTree an AVL tree.
     */
    public AvlTree(AvlTree avlTree) {
        for (Integer item: avlTree) {
            this.add(item);
        }

        // ******* //
```

```
60      }
61
62      /**
63       * Add a new node with the given key to the tree.
64       *
65       * @param newValue the value of the new node to add
66       * @return  true if the value to add is not already in the tree and it was
67       * successfully added, false otherwise.
68       */
69      public boolean add(int newValue) {
70          Integer newVal = newValue;
71          if (newVal == null) {
72              return false;
73          }
74          if (this.root == null) {
75              this.root = new TreeNode(newValue);
76              this.root.height = 0;
77              this.size = 1;
78              return true;
79          }
80
81          TreeNode node = findNode(this.root, newValue);
82          if (node.data == newValue) {
83              return false;
84          }
85          if (node.data > newValue) {
86              node.left = new TreeNode(newValue);
87          } else {
88              node.right = new TreeNode(newValue);
89          }
90
91          // Update heights after adding
92          updatetHeight(this.root);
93
94          // Balance tree
95          this.root = balanceTree(this.root);
96
97          this.size++;
98          return true;
99
100
101     }
102
103     /**
104      * Check whether the tree contains the given input value.
105      *
106      * @param searchVal searchVal the value to search for
107      * @return n the depth of the node (0 for the root) with the given value if
108      * it was found in the tree, -1 otherwise.
109      */
110     public int contains(int searchVal) {
111         Integer newVal = searchVal;
112         if (newVal == null || this.root == null) {
113             return -1;
114         }
115         int depth = 0;
116         TreeNode x = this.root;
117         while (x != null) {
118             if (x.data == searchVal) {
119                 break;
120             } else if (x.data > searchVal) {
121                 x = x.left;
122             } else {
123                 x = x.right;
124             }
125             depth++;
126         }
127
```

```
128             if (x == null) {
129                 return -1;
130             }
131             return depth;
132 
133         }
134 
135         /**
136          * Removes the node with the given value from the tree, if it exists.
137          *
138          * @param toDelete e the value to remove from the tree.
139          * @return true if the given value was found and deleted, false otherwise.
140          */
141         public boolean delete(int toDelete) {
142             Integer newVal = toDelete;
143             if (newVal == null || this.root == null) {
144                 return false;
145             }
146 
147             TreeNode node = findNode(this.root, toDelete);
148             if (node == null || node.data != toDelete) {
149                 return false;
150             }
151             if (this.root == node) {
152                 if (this.root.right == null || this.root.left == null) {
153                     if (this.root.right == null && this.root.left == null) {
154                         this.root = null;
155                     } else if (this.root.right == null && this.root.left != null) {
156                         this.root = this.root.left;
157                     } else {
158                         this.root = this.root.right;
159                     }
160                 } else {
161                     TreeNode nodeSuccessor = findSuccessor(this.root);
162                     this.root.data = nodeSuccessor.data;
163                     if (!this.removeNode(nodeSuccessor, toDelete)) {
164                         return false;
165                     }
166                 }
167             } else {
168                 if (!this.removeNode(node, toDelete)) {
169                     TreeNode nodeSuccessor = findSuccessor(node);
170                     node.data = nodeSuccessor.data;
171                     if (!this.removeNode(nodeSuccessor, toDelete)) {
172                         return false;
173                     }
174                 }
175             }
176 
177             // Update height after deleting
178             updatetHeight(this.root);
179 
180             // Balance tree
181             this.root = balanceTree(this.root);
182 
183             this.size--;
184 
185             return true;
186         }
187 
188         /**
189          * @return the number of nodes in the tree.
190          */
191         public int size() {
192             return this.size;
193         }
194 
195         /**
```

```
196         * @return an iterator on the AVL Tree. The returned iterator iterates over
197         * the tree nodes in an ascending order, and does NOT implement the remove()
198         * method.
199         */
200        public Iterator<Integer> iterator() {
201
202            Iterator<Integer> it = new Iterator<Integer>(){
203
204                private TreeNode currentNode = findMin(root);
205                int curVal;
206
207                public boolean hasNext() {
208                    return currentNode != null;
209                }
210
211                public Integer next() {
212                    curVal = currentNode.data;
213                    currentNode = findSuccessor(currentNode);
214                    return curVal;
215                }
216
217                public void remove() {
218                    throw new UnsupportedOperationException();
219                }
220            };
221            return it;
222
223        }
224
225        /**
226         * Calculates the minimum number of nodes in an AVL tree of height h.
227         *
228         * @param h the height of the tree (a non-negative number) in question.
229         * @return the minimum number of nodes in an AVL tree of the given height.
230         */
231        public static int findMinNodes(int h) {
232            if (h == 0) {
233                return 1;
234            } else if (h == 1) {
235                return 2;
236            } else {
237                // According to the formula from 'Dast'
238                return findMinNodes(h-1) + findMinNodes(h-2) + 1;
239            }
240        }
241
242        // Gets a pointer to some member in the tree, and some value, possibly in
243        // one of its sub trees. Returns true if succeeded removing the wanted node
244        // (the one with the given value), false otherwise (for example if the
245        // given node has two sub trees).
246        private boolean removeNode (TreeNode node, int toDelete) {
247            TreeNode parent = findParent(this.root, toDelete);
248            if (node.left == null && node.right == null) {
249                if (node.data < parent.data) {
250                    parent.left = null;
251                    return true;
252                }
253                if (node.data > parent.data) {
254                    parent.right = null;
255                    return true;
256                }
257                return false;
258            }
259            if (node.left == null) {
260                parent.right = node.right;
261                return true;
262            }
263            if (node.right == null) {
```

```
264             node.data = node.left.data;
265             return true;
266         }
267         return false;
268     }
269
270     // Gets a pointer to some member in the tree, and some value. Searches from
271     // the given node down the tree for the parent of a node with the given
272     // value. Returns the parent (if there is no parent returns null).
273     private TreeNode findParent(TreeNode x, int value) {
274         if (x.data <= value && x.right != null) {
275             if (x.right.data == value) {
276                 return x;
277             }
278             return findParent(x.right, value);
279         } else if (x.data > value && x.left != null) {
280             if (x.left.data == value) {
281                 return x;
282             }
283             return findParent(x.left, value);
284         }
285         return null;
286     }
287
288     //  Gets a pointer to some member in the tree, and if balancing is needed
289     // (according to AVL tree definition), calls the rotations methods.
290     // Checks balance on the given node, and also on the nodes 'under' it (in a
291     // deeper level of the tree). Returns the current node.
292     private TreeNode balanceTree(TreeNode x) {
293         if (x == null) {
294             return x;
295         }
296         if (x.left != null) {
297             x.left = balanceTree(x.left);
298         }
299         if (x.right != null) {
300             x.right = balanceTree(x.right);
301         }
302         if (x.right != null && x.left == null && x.right.height > 0) {
303             if (x.right.left != null) {
304                 x = this.rotationRL(x);
305             } else {
306                 x = this.rotationRR(x);
307             }
308             // Update heights after balancing
309             updatetHeight(this.root);
310             return x;
311         }
312         if (x.left != null && x.right == null && x.left.height > 0) {
313             if (x.left.right != null) {
314                 x = this.rotationLR(x);
315             } else {
316                 x = this.rotationLL(x);
317             }
318             // Update heights after balancing
319             updatetHeight(this.root);
320             return x;
321         }
322         if (x.right != null && x.left != null &&
323                 (Math.abs(x.left.height - x.right.height) > 1)) {
324             if (x.right.height > x.left.height) {
325                 if (x.right.left == null ||
326                         (x.right.right.height > x.right.left.height)) {
327                     x = rotationRR(x);
328                 } else {
329                     x = rotationRL(x);
330                 }
331             } else {
```

```
                if (x.left.right == null ||
                        x.left.left.height > x.left.right.height) {
                    x = rotationLL(x);
                } else {
                    x = rotationLR(x);
                }
            }
            // Update heights after balancing
            updatetHeight(this.root);
            return x;
        }
        return x;
    }

    // Right-Left rotation on the given node
    private TreeNode rotationRL(TreeNode x) {
        x.right = rotationLL(x.right);
        return rotationRR(x);
    }

    // Right-Right rotation on the given node
    private TreeNode rotationRR(TreeNode x) {
        TreeNode temp = x.right;
        x.right = temp.left;
        temp.left = x;
        return temp;
    }

    // Left-Right rotation on the given node
    private TreeNode rotationLR(TreeNode x) {
        x.left = rotationRR(x.left);
        return rotationLL(x);
    }

    // Left-Left rotation on the given node
    private TreeNode rotationLL(TreeNode x) {
        TreeNode temp = x.left;
        x.left = temp.right;
        temp.right = x;
        return temp;
    }

    // Gets a pointer to some member in the tree, and some value we would like
    // to add/remove from the tree. Returns a node with an equal value, or the
    // node that should get the given value in one of its sub trees.
    private TreeNode findNode(TreeNode x, int value) {
        if (x == null) {
            return null;
        }
        if (x.data == value) {
            return x;
        }
        if(x.data > value && x.left != null) {
            return findNode(x.left, value);
        }
        if(x.data < value && x.right != null) {
            return findNode(x.right, value);
        }
        return x;
    }

    // Gets a pointer to some member in the tree, and update its height and
    // also the heights of the nodes 'under' (in a deeper level of the tree)
    // the given node.
    private int updatetHeight(TreeNode x) {
        if (x == null) {
            return -1;
        }
```

```
400             if (x.right == null && x.left == null) {
401                 x.height = 0;
402                 return 0;
403             }
404             x.height = Math.max(updatetHeight(x.left), updatetHeight(x.right)) +1;
405             return x.height;
406         }
407 
408         // Returns the successor of a given node in the tree
409         private TreeNode findSuccessor(TreeNode x) {
410             if (x.right != null) {
411                 return findMin(x.right);
412             }
413 
414             TreeNode successorCandidate = null;
415             TreeNode y = this.root;
416             while (y != x) {
417                 if (y.data > x.data) {
418                     successorCandidate = y;
419                     y = y.left;
420                 } else {
421                     y = y.right;
422                 }
423             }
424             return successorCandidate;
425         }
426 
427         // Returns a node with the minimal value in the tree
428         private TreeNode findMin(TreeNode x) {
429             if (x == null) {
430                 return null;
431             }
432             TreeNode y = x;
433             while (y.left != null) {
434                 y = y.left;
435             }
436             return y;
437         }
438 
439     }
```

5 oop/ex4/data structures/TreeNode.java

```java
package oop.ex4.data_structures;

//Implementation of a tree node
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;
    int height;
    TreeNode(int value) {
        this.data = value;
        this.left = null;
        this.right = null;
        this.height = -1;
    }

}
```