Contents

1 Basic Test Results

```
printing files in /tmp/bodek.c3kzUf/oop/ex5/roigreenberg/testdir/2489
    oop
    README
    submission


compiling with
    javac -cp .:/cs/course/2013/oop/lib/junit4.jar  *.java oop/ex5/data_structures/*.java


tests output :
        Perfect!
```

2 README

```
roigreenberg


##################
File Description
##################

AvlTree - an implement of Avl Tree
README - this file

########
Design
########

I implement all the code for the AvlTree in the master class of the tree

I chose to build the tree as binary tree and for that I create an inner
class of nodes which every node has his parent, 2 sons, his own data, height
and has a methods that calculate the height and balance factor.
In addition, for the Iterator I create another inner class that implement an
iterator. since there was no seen require, it does not implement remove.
other design describe right below as answer to add() and delete().

the add() method -
    after finding the right place to add the new node in the principle of insert
    to BST tree, I add it and update the size.
    Then I go from the node parent to the root and look for
    node that became unbalanced (meaning his balance factor bigger then 1 or
    smaller then -1. (we stop looking after 1 such node is found)
    then I use a seperate method 'balance' the get the node and using
    rotations to rebalance the tree.
    the balance method is using another seperete method to recalculate the heights
    for the nodes that might have changed

the delete() method -
    here again, after finding the node place, I update the size, then check
    the node's sons to decide which way the deletion need to be.
    There are 3 diffrent situations, 2 of them, in cases the node have at most
    1 son, and the third if the node has to sons.
    In the first case, I use seperate method 'deleteFromAvl' that actualy delete
    the node the recalculate the height if nedded the rebalance the tree using
    'balance' method.
    In case of the third option, I first replace the deleted value with the
    succesor value then, use 'deleteFromAvl' to delete the successor node.

Not exactly a helper function, but both method using the contains method
in order to know the correct place to add to/delete from.
Since 'contains' already look for the place in order to know if the value in
the tree there is no reason to look from the same spot again.
Also as I mentioned, both methods are using 'balance' and 'calcHeight' to
    rebalance the tree properly.

######################
Implementation Issues
######################

I don't remember having a serious implementation issue.


```

```
######################
   Answer to Q.5
######################

the number of node in the tree is the number in the left-sub-tree +
right-sub-tree + 1(for the root).
in case we want the minimum number of nodes, it mean one sub tree will be higher
the other by 1 and both sub-trees will be minimum in heights of (h-1) and (h-2)

in order to create such tree I will insert the following numbers (start with 8):
 8, 3, 10, 2, 5, 9, 11, 1, 4, 6, 12, 7
 the values are insert level after level without the need to rebalance
 the order of the insertion make sure the sub-trees of the root are the minimum
 avl tree of height of 3 and 2 then with the root I get minimum tree of height 4

```

3 oop/ex5/data structures/AvlTree.java

```java
/**
 *
 */
package oop.ex5.data_structures;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 *
 * @author roigreenberg
 *
 */
public class AvlTree implements Iterable<Integer>{
    private Node root;
    private Node currentNode;
    private Node parentNode;

    private int size = 0;
    private static final int NOT_CONTAIN = -1;
    /**
     * 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){
        for (int value: data)
            add(value);
    }

    /**
     * A copy constructor -
     * a constructor that builds the tree a copy of an existing tree
     *
     * @param tree an AvlTree
     */
    public AvlTree(AvlTree tree) {
        for (int value: tree)
            add(value);
    }

    /**
     * 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){
        Node newNode = new Node(newValue);
        if (root == null) {
```

```
60              root = newNode;
61              size += 1;
62              return true;
63          }
64          if (contains(newValue) == -1){
65              size += 1;
66              if (newNode.data < parentNode.data){
67                  if (parentNode.left == null){
68                      parentNode.left = newNode;
69                  }
70              } else {
71                  if (parentNode.right == null){
72                      parentNode.right = newNode;
73                  }
74              }
75              newNode.parent = parentNode;
76              if (parentNode.height == 0){
77                  currentNode = parentNode;
78                  while (currentNode != null){
79                      currentNode.height = currentNode.setHeight();
80                      if (Math.abs(currentNode.balanceFactor()) > 1){
81                          balance(currentNode);
82                          break;
83                      }
84                      currentNode = currentNode.parent;
85                  }
86              }
87              return true;
88          }
89          // if the value already in the tree
90          return false;
91      }
92
93
94
95
96
97
98
99      /**
100      * Does tree contain a given input value
101      *
102      * @param searchVal value to search for
103      * @return if searchVal is found in the tree, return the deapth of the node
104      * (where 0 is the root)
105      * Otherwise return -1
106      */
107     public int contains(int searchVal){
108         int depth =0;
109         if (root == null) {
110             return NOT_CONTAIN;
111         }
112         if (root.data == searchVal) {
113             currentNode = root;
114             return depth;
115         }
116         parentNode = root;
117         if (parentNode.data > searchVal){
118             currentNode = parentNode.left;
119         } else {
120             currentNode = parentNode.right;
121         }
122         while (currentNode !=null){
123             depth +=1 ;
124             if (currentNode.data == searchVal)
125                 return depth;
126
127             parentNode = currentNode;
```

```
128             if (currentNode.data > searchVal){
129                 currentNode = currentNode.left;
130             } else {
131                 currentNode = currentNode.right;
132             }
133         }
134         return NOT_CONTAIN;
135     }
136 
137     /**
138      * Remove a node from the tree, if it exists
139      *
140      * @param toDelete value to delete
141      * @return true iff toDelete is found and deleted
142      */
143     public boolean delete(int toDelete){
144         if (contains(toDelete) != -1){
145             size -= 1;
146             if (currentNode.left == null || currentNode.right == null)
147                 deleteFromAvl(currentNode);
148             else {
149                 Node successor = findSuccessor(currentNode);
150                 currentNode.data = successor.data;
151                 deleteFromAvl(successor);
152             }
153             return true;
154         }
155         return false;
156     }
157 
158     /**
159      * doing the actual deletion of the node from the tree
160      * recalculate the heights after the deletion
161      * @param deleteNode - node to delete
162      */
163     private void deleteFromAvl(Node deleteNode){
164         parentNode = deleteNode.parent;
165         if (parentNode != null){
166             if (parentNode.left == deleteNode){
167                 if (deleteNode.left == null){
168                     parentNode.left = deleteNode.right;
169                 } else {
170                     parentNode.left = deleteNode.left;
171                 }
172             } else if (deleteNode.left == null){
173                 parentNode.right = deleteNode.right;
174             } else {
175                 parentNode.right = deleteNode.left;
176             }
177             calcHeights(parentNode);
178         } else {
179             root = null;
180         }
181         while (parentNode != null){
182              if (Math.abs(parentNode.balanceFactor()) > 1){
183                  balance(parentNode);
184              }
185              parentNode = parentNode.parent;
186          }
187     }
188 
189 
190     /**
191      *
192      * @return number of nodes in the tree
193      */
194     public int size(){
195         return size;
```

```
196     }
197
198     /**
199      * @return iterator to the Avl Tree. the return can pass over the tree nodes
200      * in ascending order
201      */
202     public Iterator<Integer> iterator(){
203
204         return new inOrderIteration();
205
206     }
207
208     /**
209      * an inner class to implement the iterator
210      * @author roigreenberg
211      *
212      */
213     private class inOrderIteration implements Iterator<Integer>{
214         Node curNode, nextNode;
215         /**
216          * the constructor
217          * get the first value as the smallest value in tree
218          */
219         private inOrderIteration(){
220             nextNode = findSmallest();
221         }
222         /**
223          * @return true iff there still values in tree
224          */
225         public boolean hasNext() {
226             if (nextNode != null){
227                 return true;
228             } else {
229                 return false;
230             }
231         }
232         /**
233          * @return the next value in ascending order
234          * @throws NoSuchElementException when past the ast value
235          */
236         public Integer next(){
237             if (hasNext()){
238                 curNode = nextNode;
239                 nextNode = findSuccessor(nextNode);
240                 return curNode.data;
241             } else {
242                 throw new NoSuchElementException();
243             }
244
245         }
246         /**
247          * method not support
248          */
249         public void remove(){
250             throw new UnsupportedOperationException();
251
252         }
253     }
254
255     /**
256      * look for the successor of the given node
257      * the successor is the smallest value bigger from the given node
258      * @param currentNode - the node to look for his successor
259      * @return the successor of the given node. null if the node is the maximum
260      */
261     private Node findSuccessor (Node currentNode){
262         if (currentNode.right != null){
263             currentNode = currentNode.right;
```

```
264             while (currentNode.left != null)
265                 currentNode = currentNode.left;
266             return currentNode;
267 
268         } else {
269             while ((currentNode.parent != null) &&
270                     (currentNode.data > currentNode.parent.data)){
271                 currentNode = currentNode.parent;
272             }
273             return currentNode.parent;
274         }
275     }
276 
277     /**
278      *
279      * @return the smallest node in the tree
280      */
281     private Node findSmallest(){
282         currentNode = root;
283         while (currentNode.left != null)
284             currentNode = currentNode.left;
285         return currentNode;
286     }
287 
288 
289     /**
290      * This method calculates the minimum number of nodes in an AVL tree of height h
291      *
292      * @param h height of the tree (a non-negetive number)
293      * @return minimum number of nodes in the tree
294      */
295     public static int findMinNodes(int h){
296         if (h==0)
297             return 1;
298         if (h==1)
299             return 2;
300         return (1 + findMinNodes(h-1) + findMinNodes(h-2));
301     }
302 
303 
304     /**
305      * rebalance the tree using the rotation method
306      * at the end, calculate the heights of the node that might changed
307      * @param curNode the node that might unbalanced the tree
308      */
309     private void balance (Node curNode){
310         Node nodeA, nodeB, nodeC, nodeD;
311         nodeA = curNode;
312         nodeD = nodeA.parent;
313         if (curNode.balanceFactor() == -2){
314             if (curNode.right.balanceFactor() <= 0) {
315                 // RR case
316                 nodeB = nodeA.right;
317                 nodeC = nodeB.right;
318                 nodeA.right = nodeB.left;
319                 if (nodeA.right != null){
320                     nodeA.right.parent = nodeA;
321                 }
322                 nodeB.left = nodeA;
323                 nodeA.parent = nodeB;
324                 setNodes(nodeA, nodeB, nodeD);
325                 calcHeights(nodeA);
326                 calcHeights(nodeB.parent);
327             } else {
328                 // RL case
329                 nodeB = nodeA.right;
330                 nodeC = nodeB.left;
331                 nodeB.left = nodeC.right;
```

-2/-2.5 (code='general_error') very long metho

```
                if (nodeB.left != null){
                    nodeB.left.parent = nodeB;
                }
                nodeA.right = nodeC.left;
                if (nodeA.right != null){
                    nodeA.right.parent = nodeA;
                }
                nodeC.right = nodeB;
                nodeB.parent = nodeC;
                nodeC.left = nodeA;
                nodeA.parent = nodeC;
                setNodes(nodeA, nodeC, nodeD);
                calcHeights(nodeA);
                calcHeights(nodeB);
            }
        } else {
            if (curNode.left.balanceFactor() >= 0) {
                // LL case
                nodeB = nodeA.left;
                nodeC = nodeB.left;
                nodeA.left = nodeB.right;
                if (nodeA.left != null){
                    nodeA.left.parent = nodeA;
                }
                nodeB.right = nodeA;
                nodeA.parent = nodeB;
                setNodes(nodeA, nodeB, nodeD);
                calcHeights(nodeA);
                calcHeights(nodeB.parent);
            } else {
                // LR case
                nodeB = nodeA.left;
                nodeC = nodeB.right;
                nodeA.left = nodeC.right;
                if (nodeA.left != null){
                    nodeA.left.parent = nodeA;
                }
                nodeB.right = nodeC.left;
                if (nodeB.right != null){
                    nodeB.right.parent = nodeB;
                }
                nodeC.left = nodeB;
                nodeB.parent = nodeC;
                nodeC.right = nodeA;
                nodeA.parent = nodeC;
                setNodes(nodeA, nodeC, nodeD);
                calcHeights(nodeA);
                calcHeights(nodeB);
            }
        }
    }

    /**
     * a continue for the balance method
     * set the nodes that changed during the rebalance
     * those change are for every rotation case.
     * @param node1 - node receive from balance method
     * @param node2 - node receive from balance method
     * @param node3 - node receive from balance method
     */
    private void setNodes(Node node1, Node node2, Node node3){
        if (node3 == null){
            this.root = node2;
            this.root.parent = null;
        } else {
            if (node3.right == node1){
                node3.right = node2;
            } else {
```

```
400                 node3.left = node2;
401             }
402         }
403         node2.parent = node3;
404     }
405 
406     /**
407      * recalculating the heights of the nodes that might have
408      *  changes after rebalance or deletion starting from given node
409      *  up to the unchanged node.
410      * @param node - the starting node that need recalculating the height
411      */
412     private void calcHeights(Node node){
413         boolean changed = true;
414         int oldHeights;
415         Node curNode = node;
416         while ((curNode != null) && (changed)){
417             oldHeights = curNode.height;
418             curNode.height = curNode.setHeight();
419             changed = (curNode.height != oldHeights);
420             curNode = curNode.parent;
421         }
422     }
423 
424     /**
425      * static inner class that implement the nodes for the AVL Tree
426      * @author roigreenberg
427      *
428      */
429     private static class Node {
430         private Node left = null;
431         private Node right = null;
432         private Node parent = null;
433         public int data;
434         private int height = 0;
435 
436         /**
437          * A default constructor for empty node
438          */
439         private Node() {
440 
441         }
442 
443         /**
444          * A constructor for node with data
445          * @param data - the data for the new node
446          */
447         private Node(int data) {
448             this.data = data;
449         }
450 
451         /**
452          * calculate the balance factor
453          * negetive in case the right sub-tree is higher
454          * positive in case the left sub-tree is higher
455          * zero in case both sub-trees height are equal
456          * @return the balance factor of the node
457          */
458         private int balanceFactor (){
459             if ((this.left == null) && ( this.right == null)){
460                 return 0;
461             }
462             if (this.left == null){
463                 return -this.right.height - 1;
464             } else if (this.right == null){
465                 return this.left.height + 1;
466             } else {
467                 return this.left.height - this.right.height;
```

```
468             }
469         }
470 
471         /**
472          * calculate the height of the node from the height of the node sons
473          * @return the height of the node
474          */
475         private int setHeight(){
476             if ((this.left == null) && ( this.right == null)){
477                 return 0;
478             } else if (this.left == null){
479                 return this.right.height + 1;
480             } else if (this.right == null){
481                 return this.left.height + 1;
482             } else {
483                 return Math.max(this.left.height, this.right.height) +1;
484             }
485 
486         }
487     }
488 }
```