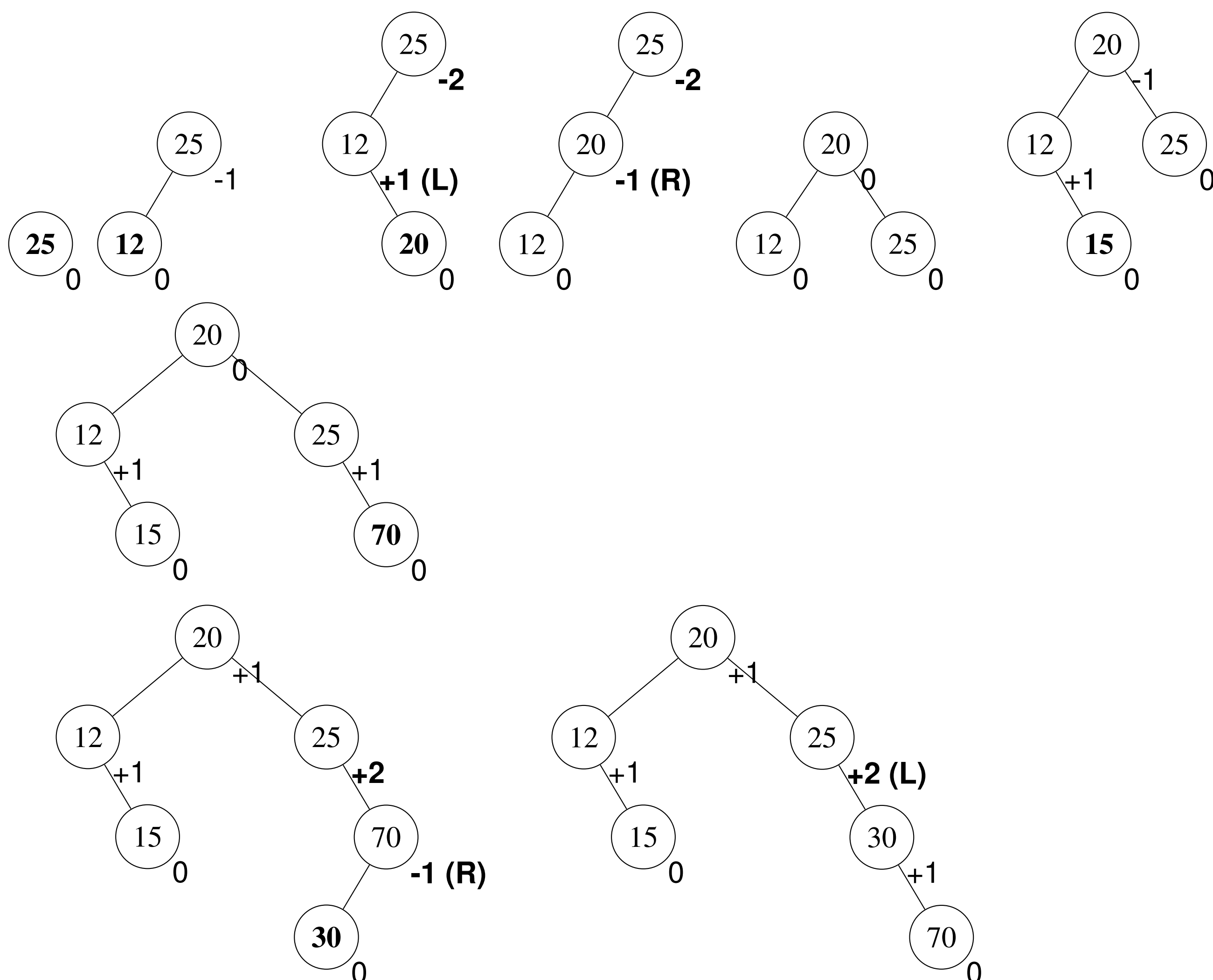
Part A

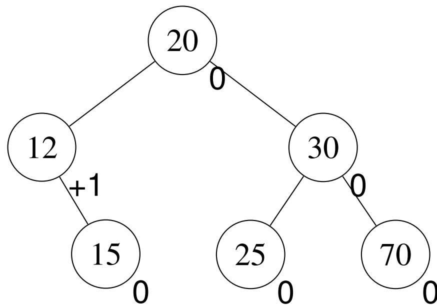
Problem 1. [25 marks] AVL Trees

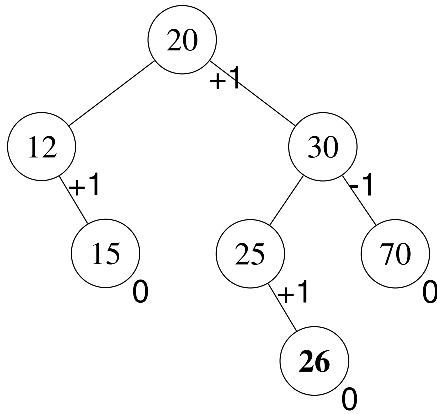
25	12	20	15	70	30	26	13	45

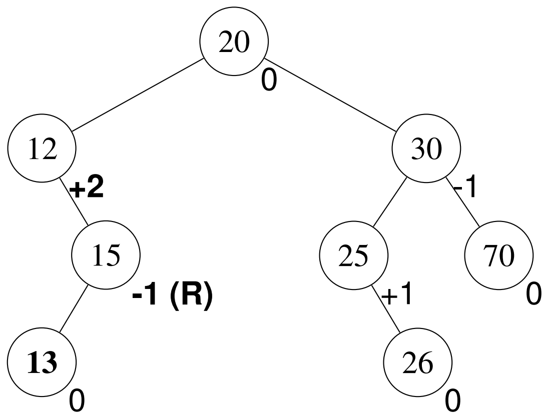
Figure 1

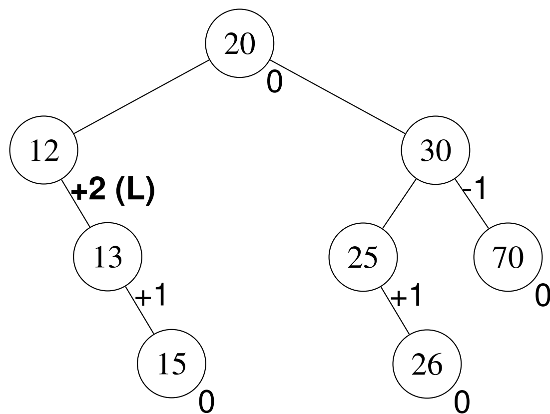
(a) [15 marks] Insert the above values (from left to right) into an empty AVL tree. Show each step and the rotations needed.

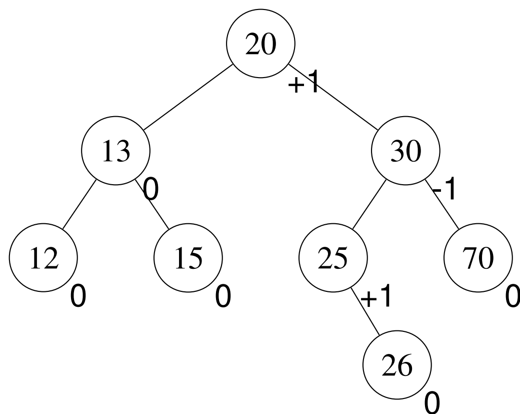


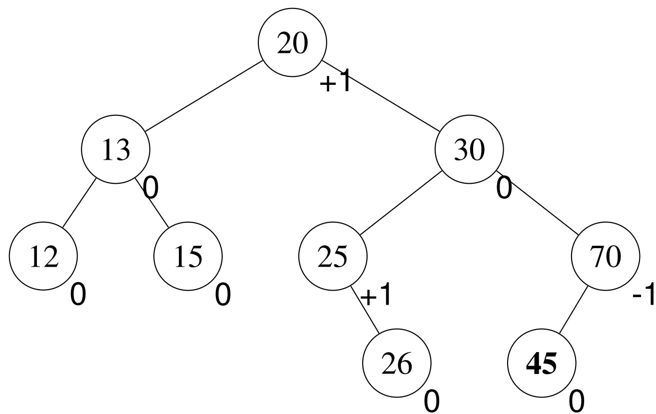

20
0
12
+1
30
0
15
0
25
0
70
0


20
+1
12
+1
30
-1
15
0
25
+1
70
0
26
0

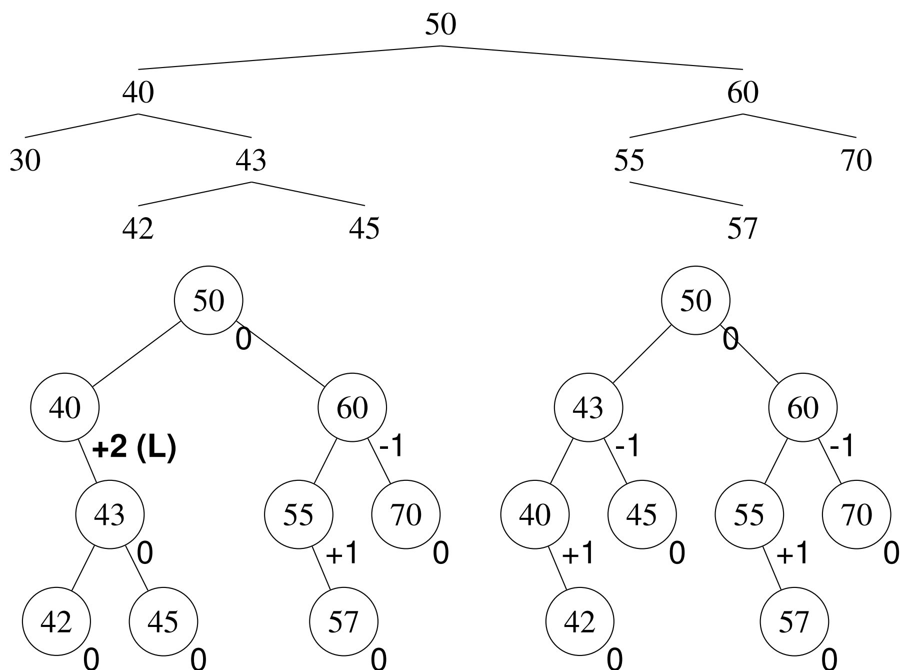

20
0
12
+2
30
-1
15
-1 (R)
25
+1
70
0
13
0
26
0


20
0
12
+2 (L)
30
-1
13
+1
25
+1
70
0
15
0
26
0


20
+1
13
0
30
-1
12
0
15
0
25
+1
70
0
26
0


20
+1
13
0
30
0
12
0
15
0
25
+1
70
-1
26
0
45
0

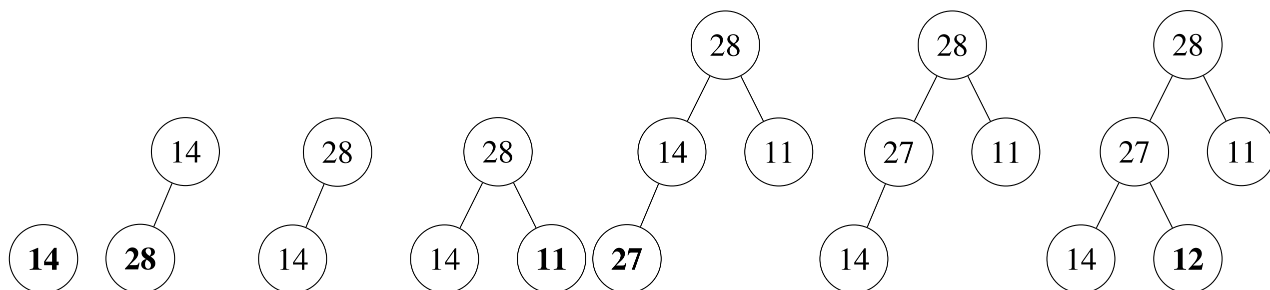
(b) [10 marks] Delete the key 30 from the following AVL tree, showing all your work.

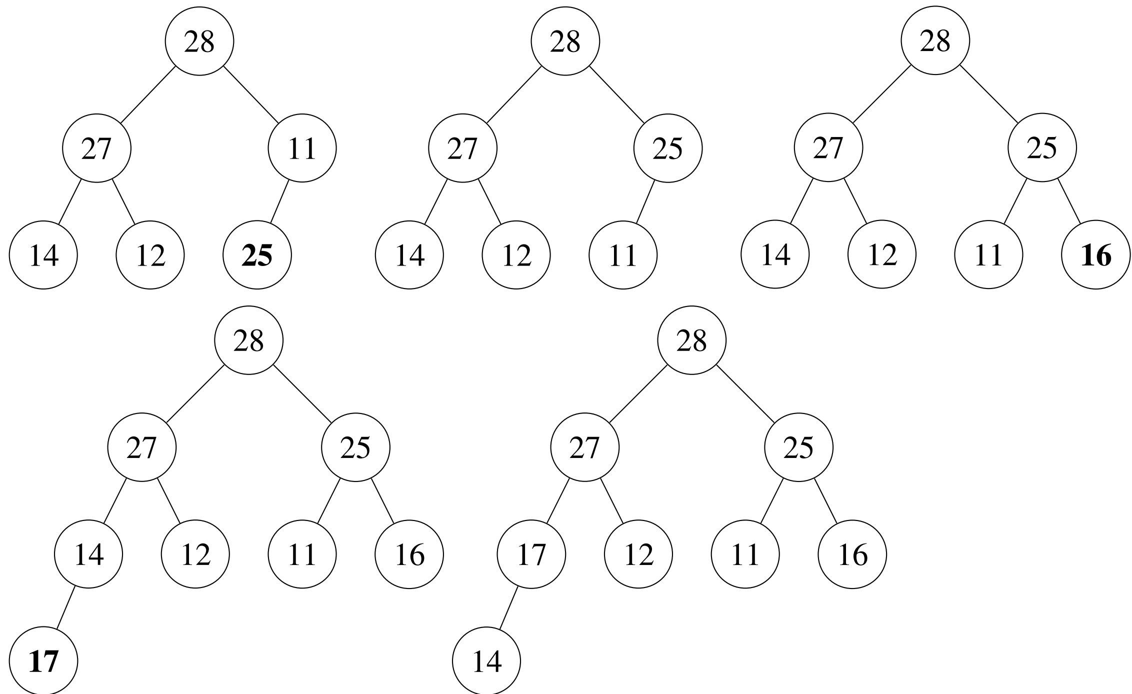


Problem 2. [25] Heaps

14	28	11	27	12	25	16	17

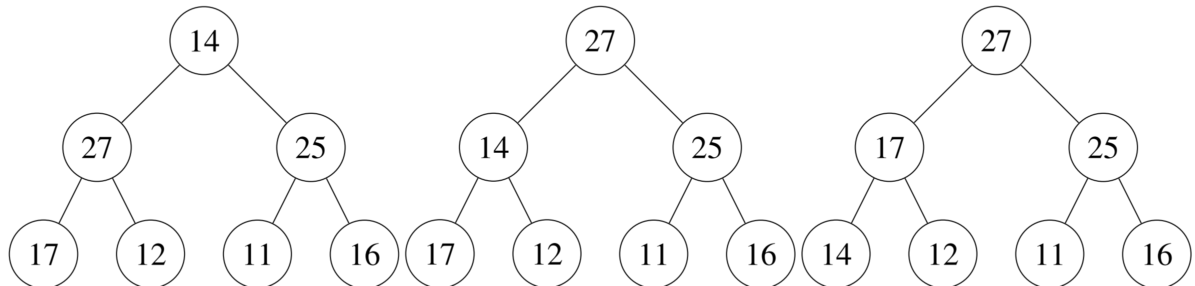
(a) [15 marks] Add the above numbers (from left to right) to an initially empty max heap with top-down approach. Redraw the heap when each value is inserted into the heap and also show the internal nodes change.





(b) [10 marks]

Delete the root of the heap created in the above question. Redraw the heap when the root is deleted from the heap and also show the internal nodes change.



Part B

Problem 3. [5x10] Programming
Your goal is to write a program that accepts some integer-values and create a binary search tree (BST) considering those values sequentially. Finally, your program should report the following:

(a) The count of number of leaves in the BST
(b) The count of only left children in the BST
(c) The count of only right children in the BST
(d) The inorder traversal of the BST
(e) The breadth first traversal of the *mirror* tree of the BST

```
1
2 import java.util.Random;
3
4
5 /****  Queue (used for BreadthFirstSearch)****/
6
7 class Queue<T> {
8 private java.util.LinkedList<T> list = new java.util.LinkedList<T>();
9     public Queue() {
10    }
11    public void clear() {
12        list.clear();
13    }
14    public boolean isEmpty() {
15        return list.isEmpty();
16    }
17    public T firstEl() {
18        return list.getFirst();
19    }
20    public T dequeue() {
21        return list.removeFirst();
22    }
23    public void enqueue(T el) {
24        list.addLast(el);
25    }
26    public String toString() {
27        return list.toString();
28    }
29 }
30
31
32
```

```
33 /**** BSTNode (used in NewBST) ****/
34
35  class BSTNode<T extends Comparable<? super T>> {
36      protected T el;
37      protected BSTNode<T> left, right;
38      public BSTNode() {
39          left = right = null;
40      }
41      public BSTNode(T el) {
42          this(el,null,null);
43      }
44      public BSTNode(T el, BSTNode<T> lt, BSTNode<T> rt) {
45          this.el = el; left = lt; right = rt;
46      }
47 }
48
49 /**** NewBST (A New binary search tree) ****/
50 public class NewBST<T extends Comparable<? super T>> {
51      protected BSTNode<T> root = null;
52      public NewBST() {
53      }
54
55      public void insert(T el) {
56          BSTNode<T> p = root, prev = null;
57          while (p != null) {  // find a place for inserting new node;
58              prev = p;
59              if (el.compareTo(p.el) < 0)
60                  p = p.left;
61              else p = p.right; //CONSIDERS EQUAL VALUES AS WELL
62          }
63          if (root == null)    // tree is empty;
64              root = new BSTNode<T>(el);
65          else if (el.compareTo(prev.el) < 0)
66              prev.left  = new BSTNode<T>(el);
67          else prev.right = new BSTNode<T>(el);
68      }
69
70
71      public void inorder() {
72          inorder(root);
73      }
74      protected void inorder(BSTNode<T> p) {
75          if (p != null) {
76              inorder(p.left);
77              visit(p);
78              inorder(p.right);
79          }
80      }
```

```
81      protected void visit(BSTNode<T> p) {
82          System.out.print(p.el + " ");
83      }
84
85      public void breadthFirst() {
86          BSTNode<T> p = root;
87          Queue<BSTNode<T>> queue = new Queue<BSTNode<T>>();
88          if (p != null) {
89              queue.enqueue(p);
90              while (!queue.isEmpty()) {
91                  p = queue.dequeue();
92                  visit(p);
93                  if (p.left != null)
94                      queue.enqueue(p.left);
95                  if (p.right != null)
96                      queue.enqueue(p.right);
97              }
98          }
99      }
100
101
102     public int countLeftChild()
103      {
104          return countLeftChild(root);
105      }
106
107
108     public int countLeftChild(BSTNode<T> node)
109     {
110         if(node==null){ return 0; }
111         if(node.left == null)
112         {
113             // No Left child.
114             return countLeftChild(node.right);
115         }
116         else
117         {
118             return countLeftChild(node.left) + countLeftChild(node.
    right) + 1;
119         }
120     }
121
122
123     public int countRightChild()
124     {
125         return countRightChild(root);
126     }
127
```

```
128    public int countRightChild(BSTNode<T> node)
129    {
130        if(node==null){ return 0; }
131        if(node.right == null)
132        {
133            // No right child.
134            return countRightChild(node.left);
135        }
136        else
137        {
138            return countRightChild(node.left) + countRightChild(node.
   right) + 1;
139        }
140    }
141
142    public int getLeafCount()
143    {
144        return getLeafCount(root);
145    }
146
147    public int getLeafCount(BSTNode<T> node)
148    {
149        if (node == null)
150            return 0;
151        if (node.left == null && node.right == null)
152            return 1;
153        else
154            return getLeafCount(node.left) + getLeafCount(node.right)
   ;
155    }
156
157
158    public void mirror() {
159        mirror(root);
160      }
161
162    public void mirror(BSTNode<T> node) {
163        if (node != null) {
164          // do the sub-trees
165          mirror(node.left);
166          mirror(node.right);
167
168          // swap the left/right pointers
169          BSTNode<T> temp = node.left;
170          node.left = node.right;
171          node.right = temp;
172        }
173      }
```

```
174   public static void main(String[] args) {
175
176     int MAXIMUM_RUN = 5; //TO REPEAT THE PROCESS 5 TIMES
177
178     for(int run_id=0; run_id<MAXIMUM_RUN; run_id+=1)
179     {
180
181     NewBST<Integer> tree = new NewBST<Integer>();
182
183     //CONSIDERING DIFFERENT AMOUNT OF VALUES IN EACH RUN
184
185     Integer[] data = new Integer[MAXIMUM_RUN+run_id];
186
187       for(int id=0; id<5+run_id; id++)
188       {
189         int min = 0;
190         int max = 100;
191
192         //USING RANDOM VALUES BUT ORIGINALLY WILL BE USING FILE
193         int random_int = (int)Math.floor(Math.random()*(max-min+1)+
      min);
194         data[id] = random_int;
195
196       }
197
198     System.out.print("Input: ");
199
200     for(Integer i = 0 ; i<data.length; i++)
201     {
202       System.out.print(data[i]+" ");
203       //CREATING THE TREE USING CONSIDERED DATA/VALUES
204       tree.insert(data[i]);
205     }
206
207     for(int j=0; j<22-data.length*2;j++) //FOR SPACING
208       System.out.print(" ");
209
210     //PRINTING (a) to (c) INFORMATION
211     System.out.print("\tOutput : "+"(a) "+ tree.getLeafCount()+"\t(b)
      "+tree.countLeftChild()+"\t(c) "+tree.countRightChild()+"\t(d) "
      );
212
213     //PRINTING (d) INFORMATION
214     tree.inorder();
215
216     for(int j=0; j<18-data.length*2;j++) //FOR SPACING
217       System.out.print(" ");
218
```

```
219        //PRINTING (e) INFORMATION
220        System.out.print("\t(e) ");
221        tree.mirror(tree.root);
222        tree.breadthFirst();
223
224        System.out.println();
225
226        }
227
228
229    }
230
231 }
```