Com S 228
Fall 2016
Final Exam

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: _____

ISU NetID (username): _____@iastate.edu

Closed book/notes, no electronic devices, no headphones*.** Time limit ***120 minutes. Partial credit may be given for partially correct solutions.

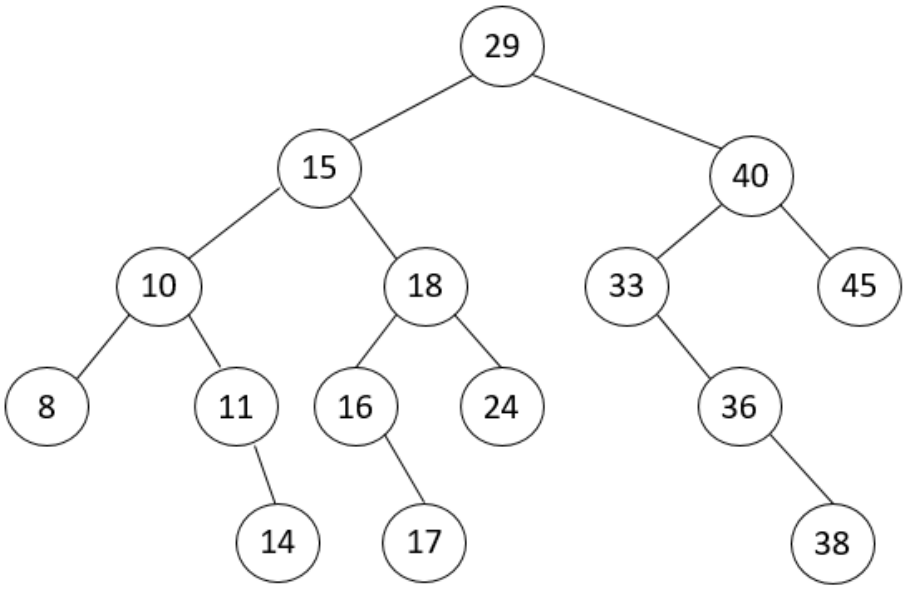
- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

Please *peel off* the **last two** of your exam sheets for scratch purpose.

If you have questions, please ask!

Question	Points	Your Score
1	14	
2	20	
3	5	
4	10	
5	16	
6	13	
7	22	
Total	100	

1. (14 pts) All the questions in this problem concern the ***same*** splay tree storing 15 integer values as shown below. While working on parts g) and h), to reduce the chance for error and to get partial credit from an incorrect final answer, you are suggested to draw the tree after the initial node operation and after each subsequent splaying step or even each tree rotation.



a) (1 pt) The above tree has height _____.

b) (1 pt) The node 17 has height _____.

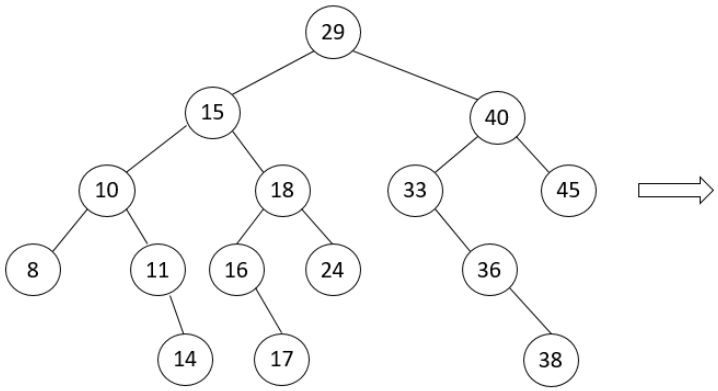
c) (1 pt) The node 11 has depth _____.

d) (1 pt) The node 38 has the successor node _____.

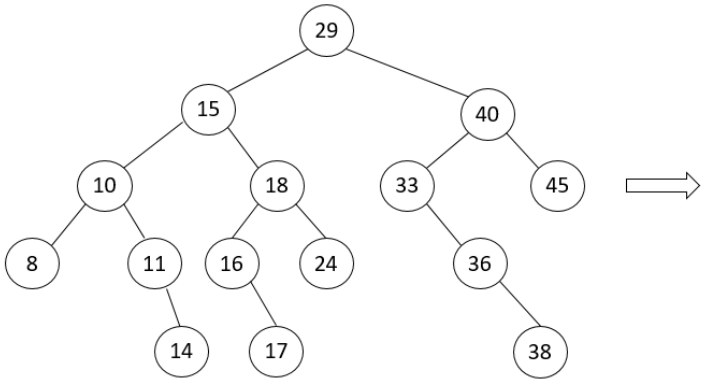
e) (1 pt) The node 29 has the predecessor node _____.

f) (2 pts) Output all the nodes in the post-order. (Do not splay at any node.)

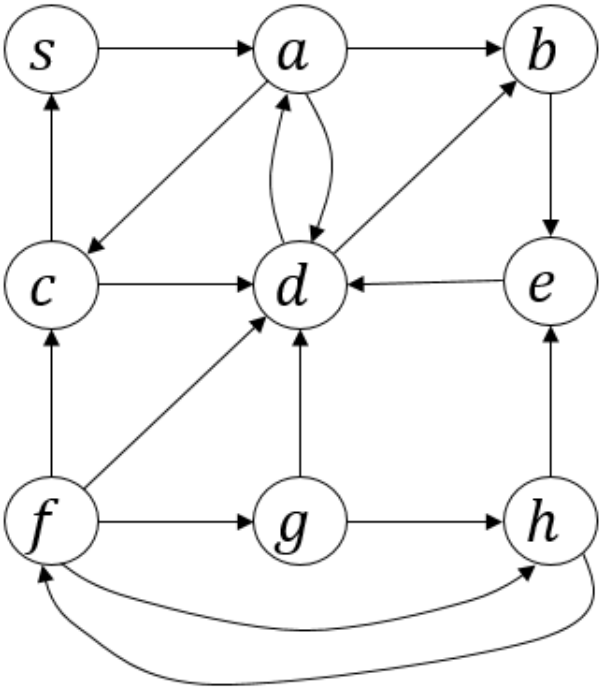
g) (4 pts) Insert 9 into the splay tree. (To save time, you may simply draw a node without the circle.)



h) (3 pts) Delete 18 from the same original splay tree.



2. (20 pts) Consider the simple directed graph below with nine vertices.

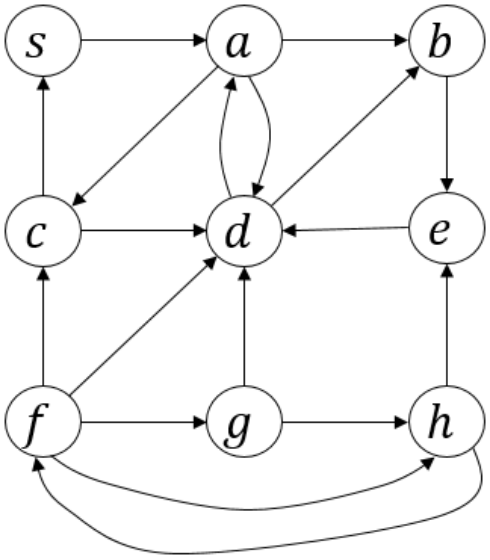


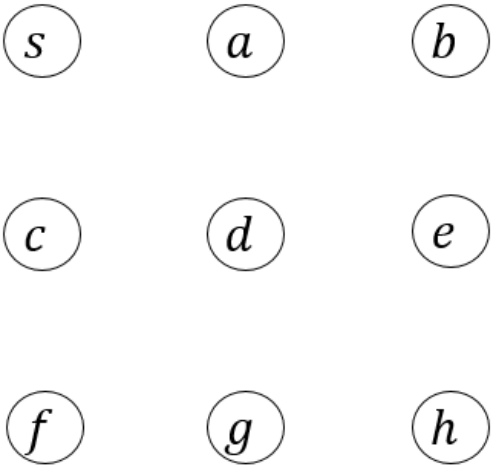
a) (1 pt) The in-degree of the vertex d is _____.

b) (1 pt) The out-degree of the vertex h is _____.

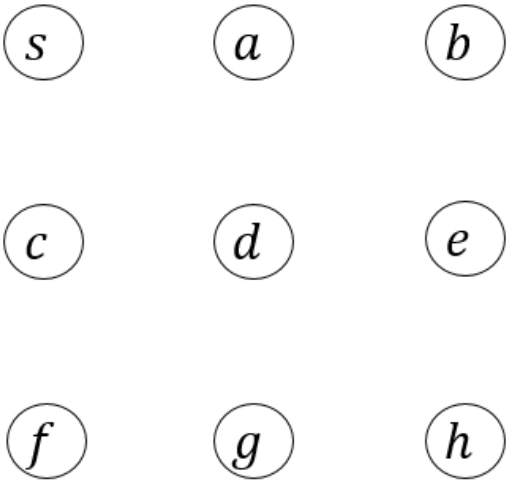
c) (1 pt) Is the directed graph weakly connected? (Yes/No) _____.

d) (6 pts) Perform a depth-first search (DFS) on the graph (copied over below on the left). Recall that the search is carried out by the method **`dfs()`** which iterates over the vertices, and calls a recursive method **`dfsVisit()`** on a vertex when necessary. Suppose that **`dfs()`** processes the vertices in the order $s, a, b, c, d, e, f, g, h$. In the right figure below, draw the DFS forest by adding edges to the vertices. Again, break a tie according to the ***alphabetical order***.

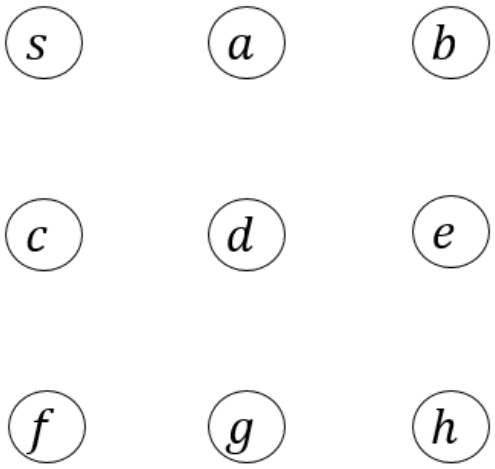




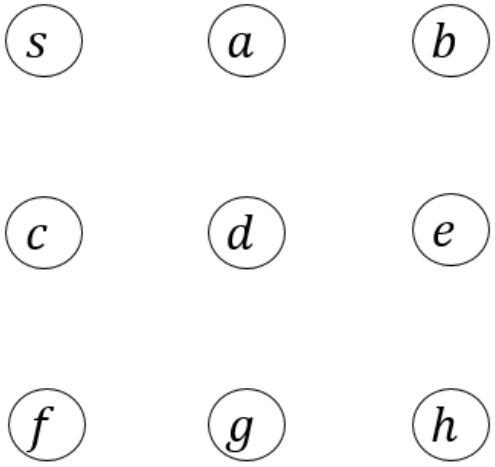
e) (4 pts) Draw all the back edges during the DFS carried out in d).



f) (2 pts) Draw all the forward edges during the same DFS.

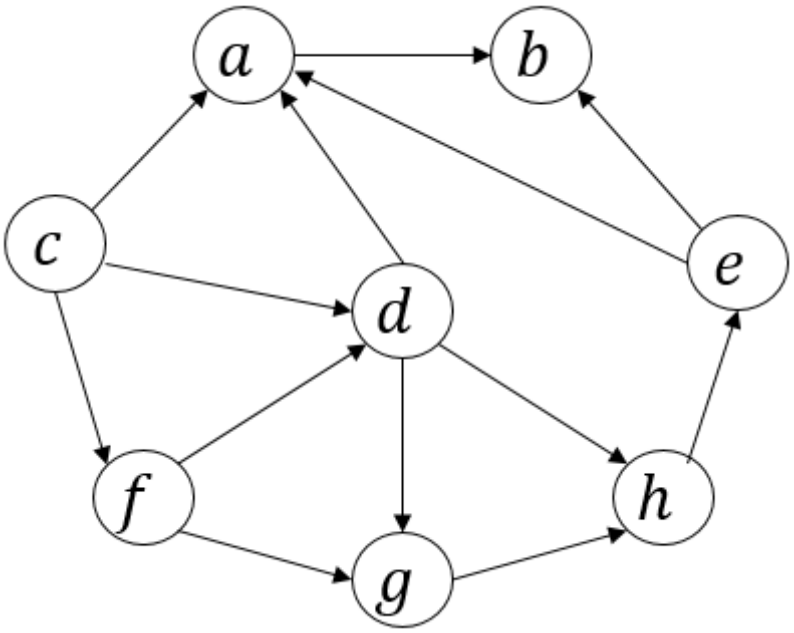


g) (3 pts) Draw all the cross edges during the same DFS.



h) (2 pts) Does the graph contain a cycle? If so, draw one such cycle below by including ***only*** the vertices and edges on the cycle.

3. (5 pts) Consider the following directed acyclic graph *G*.



Give a topological sort of *G* by filling out the table below.

4. (10 pts) Suppose G is a directed graph with V vertices and E edges. In the following questions, you are asked to give the running times of various algorithms on G. You must give your answers as a function of V and E using the big-O notation.

a) (2 pts) Suppose G is represented using an ***adjacency list***. What is the running time of ***breadth-first search*** on G?

b) (2 pts) Suppose G is represented using an ***adjacency list***. What is the running time of ***topological sort*** on G?

The next parts refer to the code snippet below.

```
count = 0
for each vertex v in G
    for each neighbor u of v
        count++
```

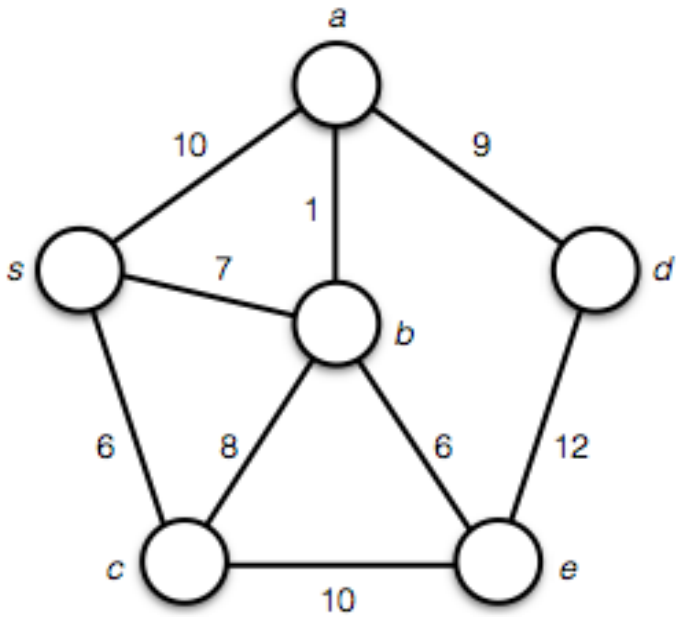
Give the running time for the code snippet above for each of the following representations of G.

c) (2 pts) Adjacency matrix.

d) (2 pts) Adjacency list.

e) (2 pts) HashMap/HashSet.

5. (16 pts) Consider the weighted graph G below.

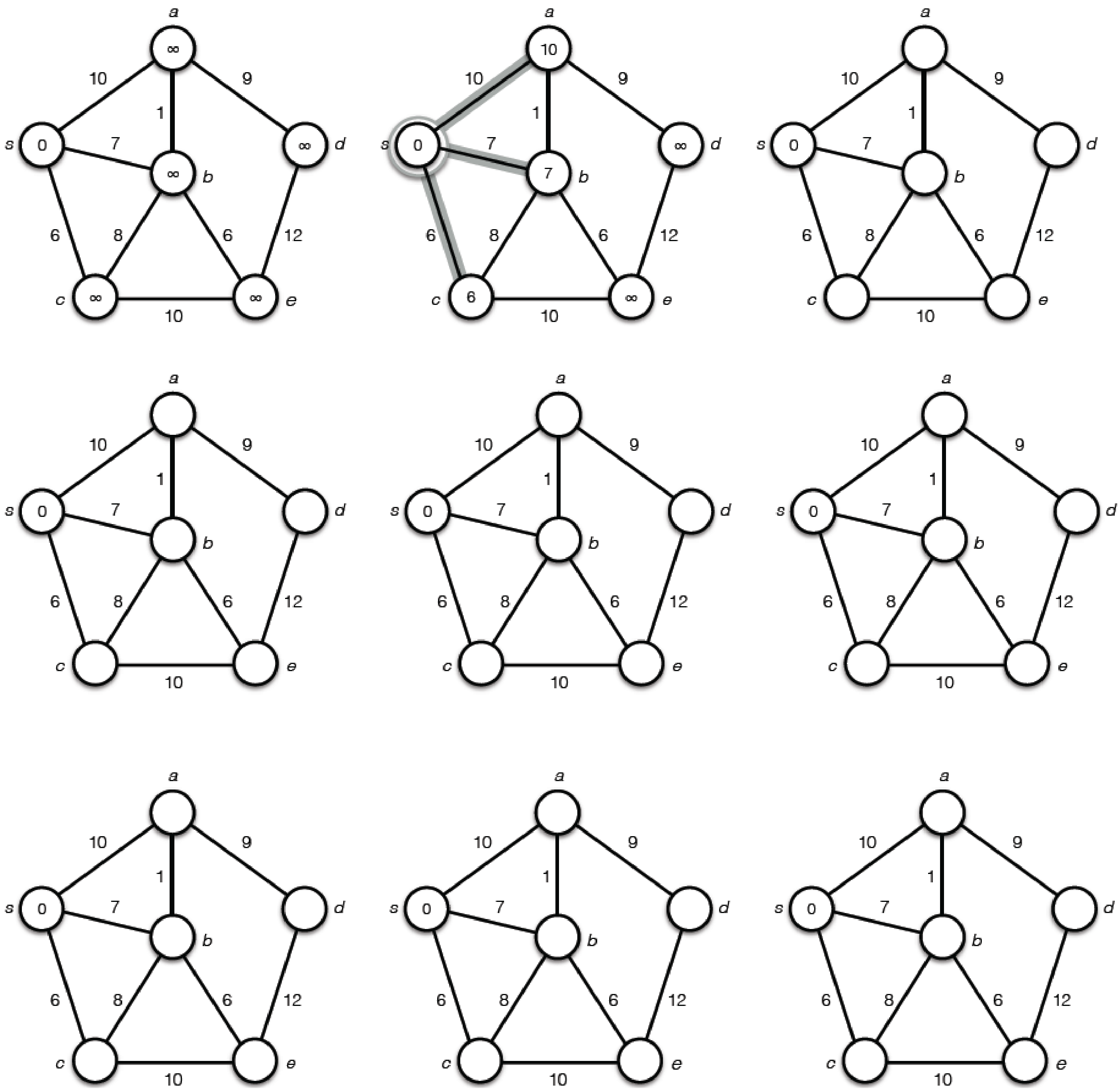


Show the ***complete execution*** of Dijkstra's algorithm for finding all shortest paths from node *s* in G using the pre-drawn templates on the next page. At each step,

- write the distance label (d-value) of each node inside the node,
- circle the node that is selected at that step, and
- for each node v, either mark or darken or double (or draw a line wiggling around) the edge from pred(v) to v.

For your reference, we show the initial d-values, as well as the outcome of the first iteration. In the first step (shown in the second template on the next page), the node s is marked, so are the three edges coming out of this vertex. You need to fill out the remaining templates. (Note that the template for the final step of the execution of Dijkstra's algorithm is not given since it will result in no change of the distance or the predecessor information.)

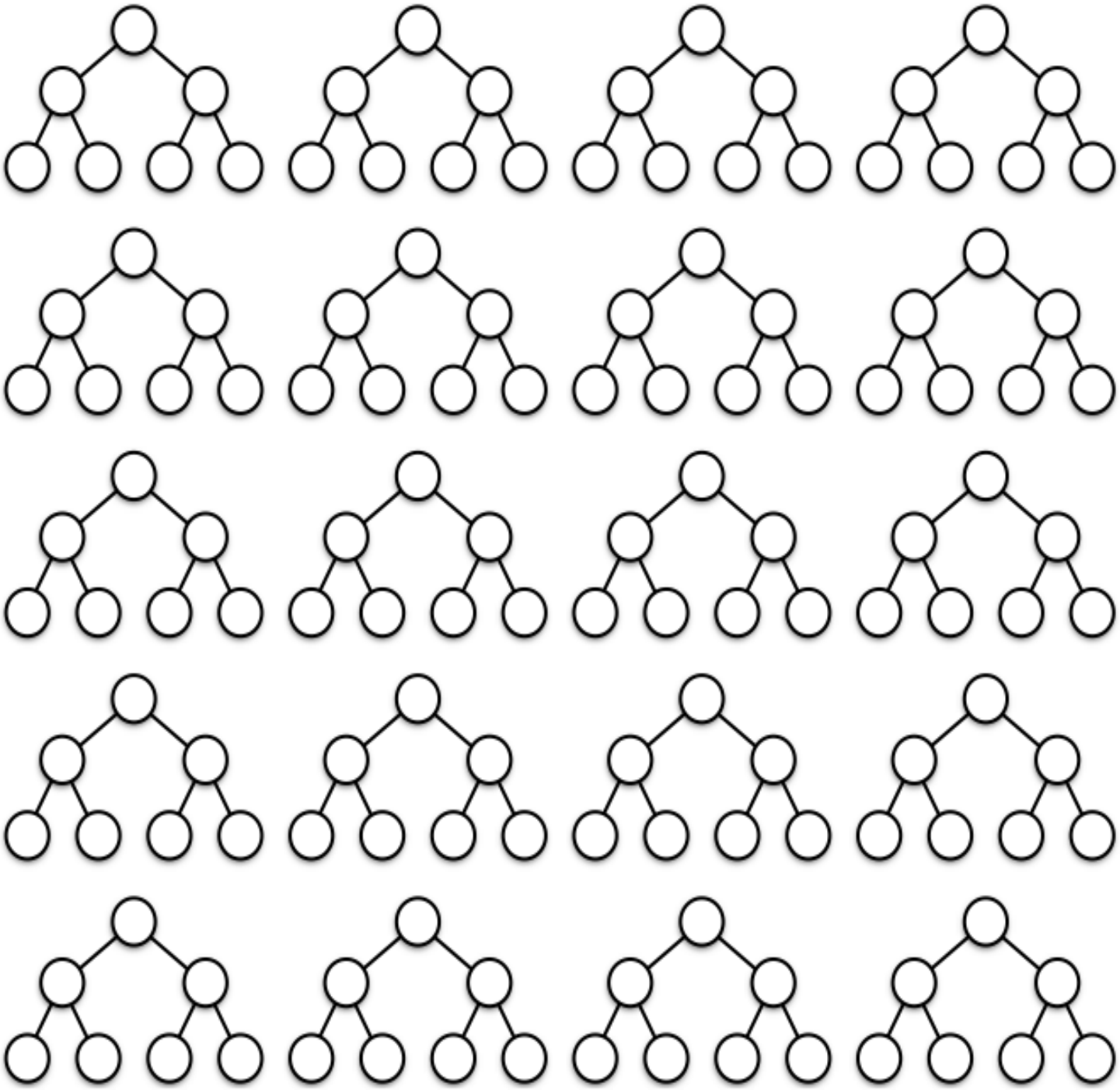
Only partial credit will be awarded if you just give the final answer and/or shortest path tree.



a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
∞
0
∞
∞
∞
∞
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
10
0
7
∞
6
∞
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0
a
s
b
d
c
e
10
9
1
7
6
8
6
12
10
0

6. (13 pts) The following table illustrates the operation of HEAPSORT on the array in row 0. Every subsequent row is calculated by performing a ***single swap*** as determined by HEAPSORT on the preceding line. Note that such a swap may happen during either HEAPIFY or the sorting that follows. Fill in the missing lines, and underline or circle the ***first line that is a heap*** (in other words, demarcate the completion of HEAPIFY). You may use the empty trees on the next page for scratch. ***Only the table is used for grading! The trees are just tools for you and will not be evaluated.*** (There may be more rows than you need to fill in. Just leave the extra ones blank.)

Row				Array			
0	3	4	6	5	2	0	1
1							
2							
3							
4	5	1	3	4	2	0	6
5							
6							
7							
8							
9	0	2	3	1	4	5	6
10							
11							
12							
13							
14							
15							
16							
17							
18							

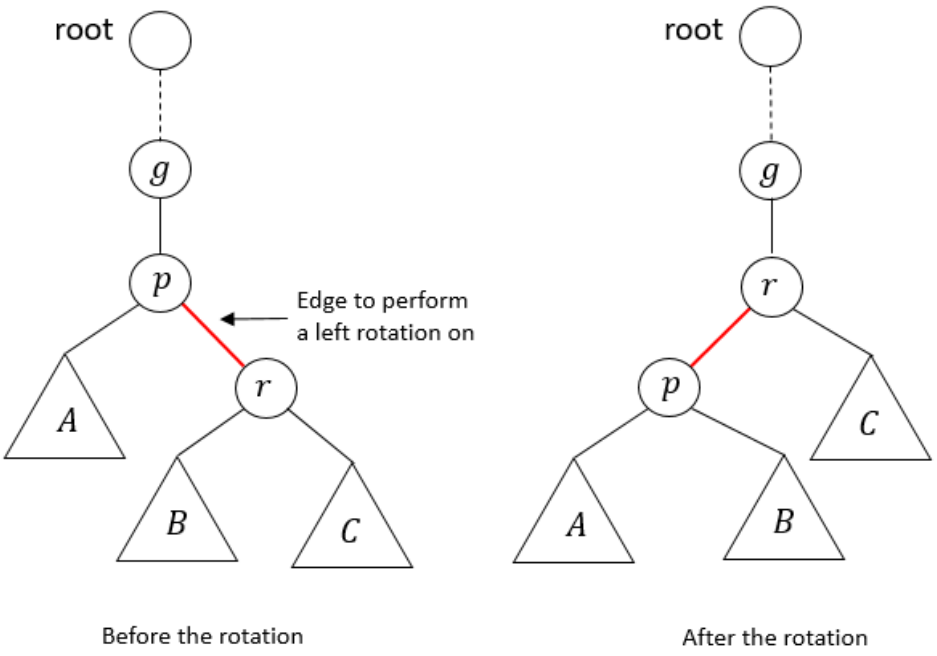


7. (22 pts) In this problem, you are asked to augment an existing generic class **`BinaryTree<E>`** which implements a ***binary tree*** (***not*** a binary search tree). A node in the tree is implemented by the class **`Node<E>`**. Note that if a node n does not have a left or right child, then `n.left == null` or `n.right == null` accordingly. The two classes are displayed below.

```
public class Node<E>
{
    public Node<E> left;
    public Node<E> right;
    public Node<E> parent;
    public E data;


    public Node(E key)
    {
      this(key, null, null, null);
    }


    public Node(E key, Node<E> parent, Node<E> left, Node<E> right)
    {
      this.data = key;
      this.parent = parent;
      this.left = left;
      this.right = right;
    }
}


public class BinaryTree<E>
{
    protected Node<E> root;


    public BinaryTree()
    {
        root = null;
    }


    public BinaryTree(Node<E> rt)
    {
        root = rt;
    }

    // other methods
    // ...

}
```

a) (15 pts) Within the class **BinaryTree<E>**, add a method **`leftRotate()`** which rotates an edge between a parent node p and its right child node r. As illustrated in the figure below, after the rotation the following three conditions will be satisified:

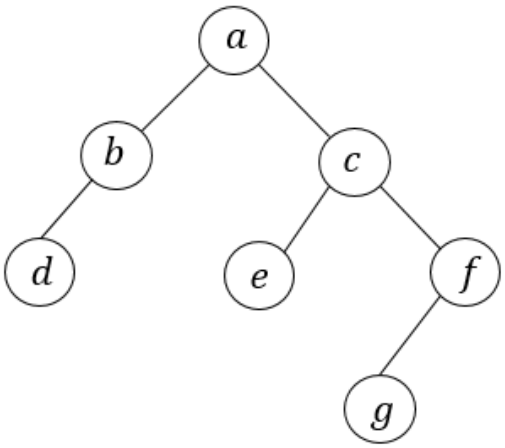
1. The node r becomes the new parent of the node p.
2. The former left subtree B of r, if exists, becomes the right subtree of p.
3. The former parent g of p, if exists, becomes the new parent of r.



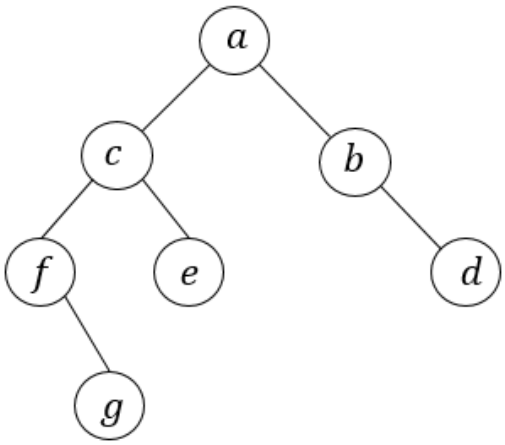

```
/**
 * Perform a right rotation on the edge between the parent node p
 * and its right child node r.
 *
 * @param p  parent node
 * @param r  child node
 * @throws NullPointerException  if either p or r is null
 * @throws IllegalArgumentException  if neither p nor r is null
 *                            but r is not a right child of p
 */
public void leftRotate(Node<E> p, Node<E> r)
            throws NullPointerException, IllegalArgumentException
{
      // handle exceptions.
      //
      // insert code below (4 pts)
```

```
        // make the left subtree of r the new right subtree of p.
        //
        // insert code below (3 pts)









        // establish the relationship between r and the parent of p
        // (if p has one).
        //
        // insert code below (6 pts) by first filling in the blank below

        Node<E> g = _____________________________;  // grandparent of r









        // reverse the parent-child relationship between p and r.
        //
        // insert code below (2 pts)



}
```

b) (7 pts) A **mirror image** of a binary tree is generated by **switching** the left and right children of **every node** in the original tree. For example, the left figure below displays a binary tree while the right figure displays its mirror image.



Original binary tree



Its mirror image

You are asked to add a mutator method **`mirrorImage()`** to the class **`BinaryTree<E>`**. This method modifies the original tree object to its mirror image. The method must reuse the existing nodes in the tree, and must **not** create any new nodes. For this task, you are required to implement a private recursive method **`mirrorImageRec()`** .

```
/**
 * Change the binary tree into its mirror image.
 */
public void mirrorImage()
{
      // insert code below (1 pt)




}


/**
 * Replace the subtree rooted at n with its mirror image. Link
```

```
 * updates only. No creation of a new node.
 *
 * @param n  root of the subtree to be replaced with its mirror
 *           image.
 */
private void mirrorImageRec(Node<E> n)
{
      // handle the case n == null
      //
      // insert code below (1 pt)









      // swap the left and right subtrees and then generate their
      // mirror images.
      //
      // insert code below (5 pts)

}
```

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)