Com S 228
Spring 2019
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.

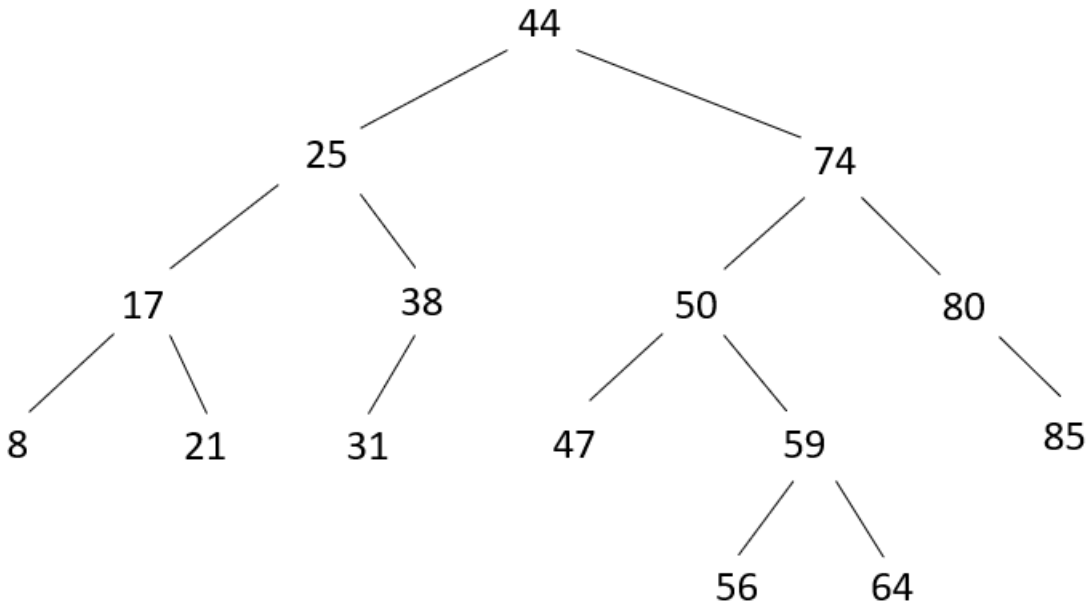
- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

Peel off the last one *of your exam sheets for scratch purpose*.

If you have questions, please ask!

Question	Points	Your Score
1	18	
2	24	
3	14	
4	14	
5	10	
6	20	
Total	100	

1. (18 pts) All the questions in this problem concern the ***same*** splay tree storing 15 integer values as shown below. While working on parts j) and k), to reduce the chance for error and to get partial credit despite 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 pts) The tree has _____ leaves.

b) (1 pts) The tree has _____ internal nodes.

c) (1 pts) The tree has _____ edges.

d) (1 pt) The tree has height _____.

e) (1 pt) The node 50 has height _____.

f) (1 pt) The node 31 has depth _____.

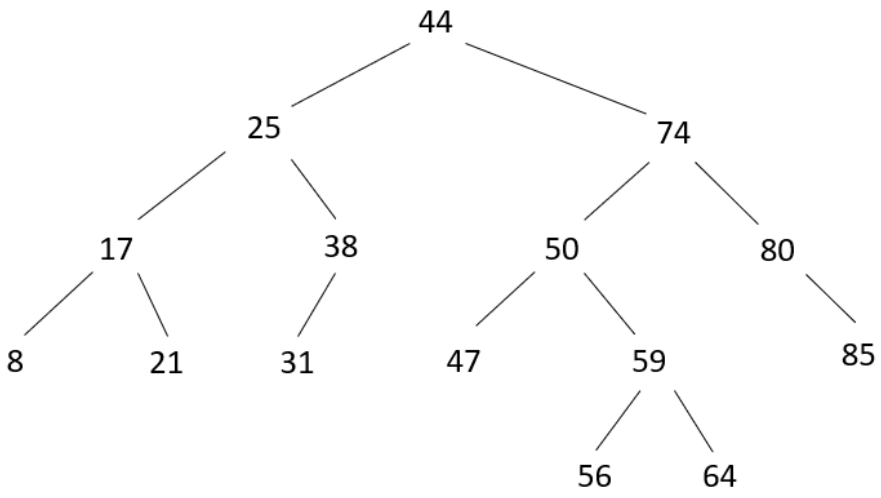
g) (1 pt) The node 38 has the successor node _____.

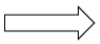
h) (1 pt) The node 74 has the predecessor node _____.

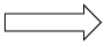
i) (2 pts) Output all the nodes in the ***preorder***. (Do not splay the tree at any node.)

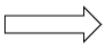
j) (5 pts) Insert 71 into the splay tree. In the area to the right of the first arrow, you may draw the splay tree after the insertion (i.e., the final answer), and use the remaining space for your scratch work. Or, you may draw the intermediate tree just before the first splay step, and then add the remaining steps next to the arrows on the next two pages (if needed), ending with the spay tree after the insertion.

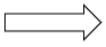
If you include intermediate steps, in such a step you may draw only part of the tree that is being changed. Mark the answer tree if intermediate steps are drawn.



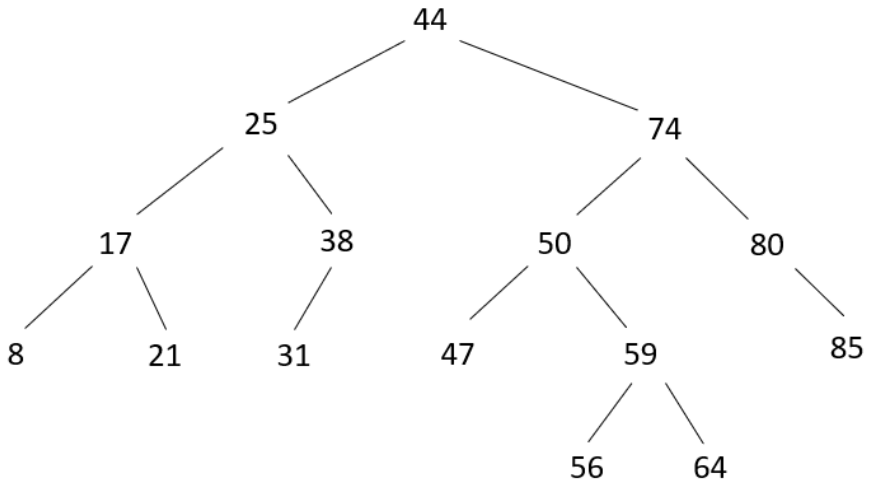








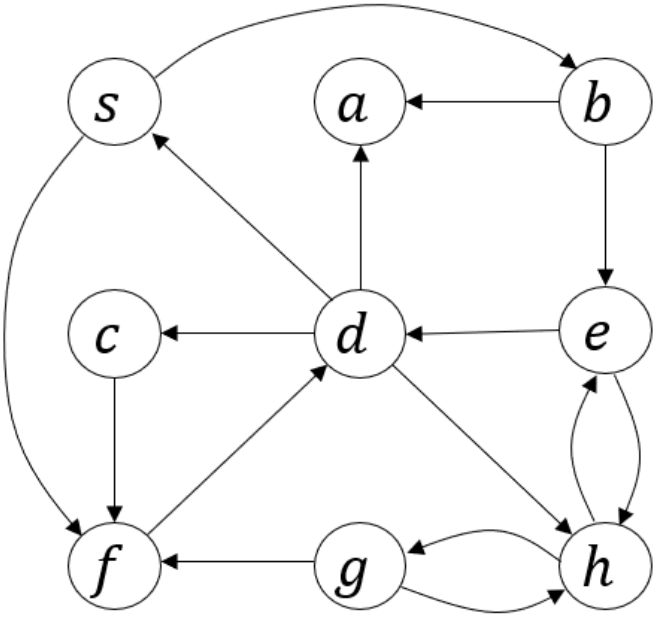
k) (3 pts) Delete 74 from the original splay tree (displayed on the next page). In the area to the right of the first arrow, you may draw the splay tree after the deletion (i.e., the final answer), and use the remaining space for your scratch work. Or, you may draw intermediate trees (step by step) next to the arrows, ending with the splay tree after the deletion.


44
25
74
17
38
50
80
8
21
31
47
59
85
56
64

⇒

⇒

2. (24 pts) Consider the simple directed graph below with nine vertices.



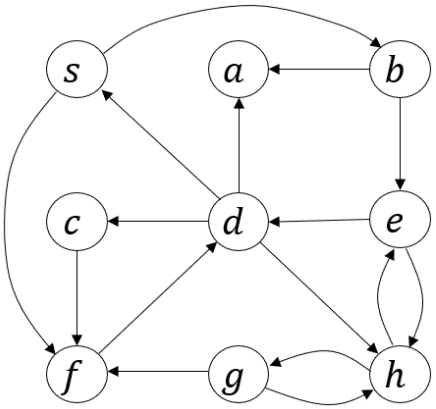
a) (1 pt) The in-degree of the vertex h is _____.

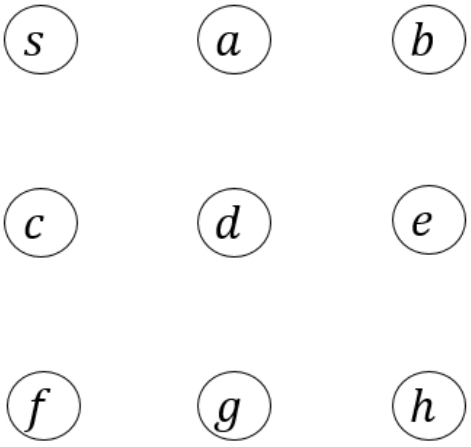
b) (1 pt) The out-degree of the vertex d is _____.

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

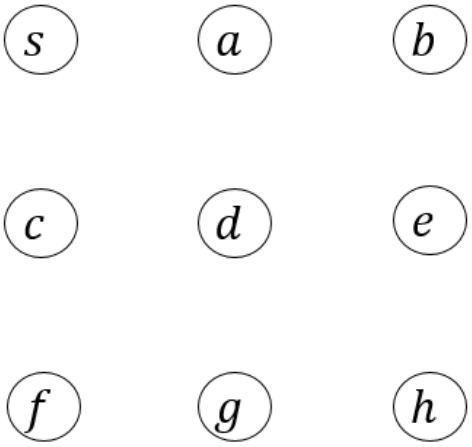
d) (1 pt) Is the graph weakly connected? (Yes/No) _____.

e) (13 pts) Perform a depth-first search (DFS) on the same graph (copied over to 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 $s, a, b, c, d, e, f, g, h$. Below are four figures (in a 2 x 2 arrangement) that show vertices only. In the first figure, draw the ***DFS tree*** or ***forest*** by adding edges to the vertices. Break a tie according to the ***alphabetical order***. Add back edges, forward edges, and cross edges to the remaining three figures, respectively.

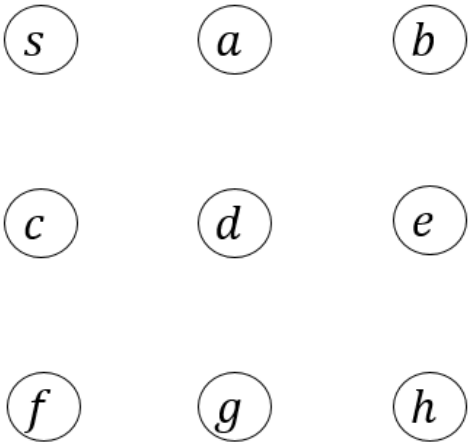




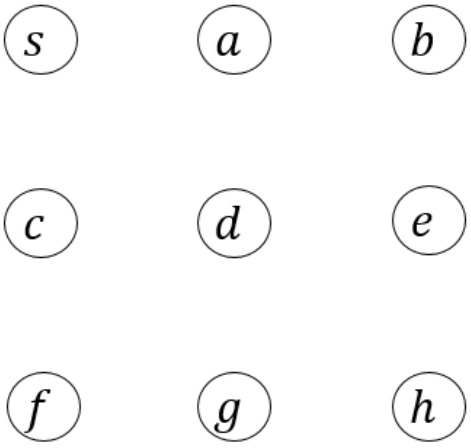
(6 pts) Add all the ***tree edges*** in the above.



(3 pts) Add all the ***back edges*** in the above.

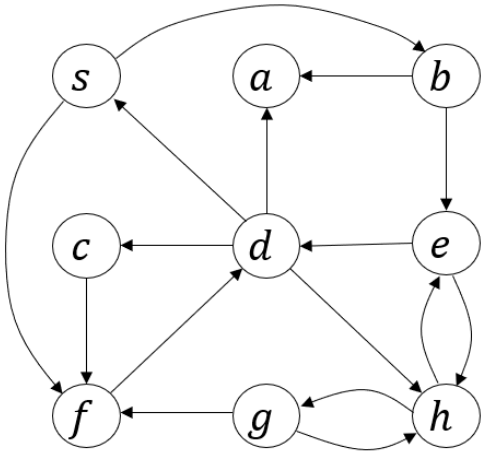


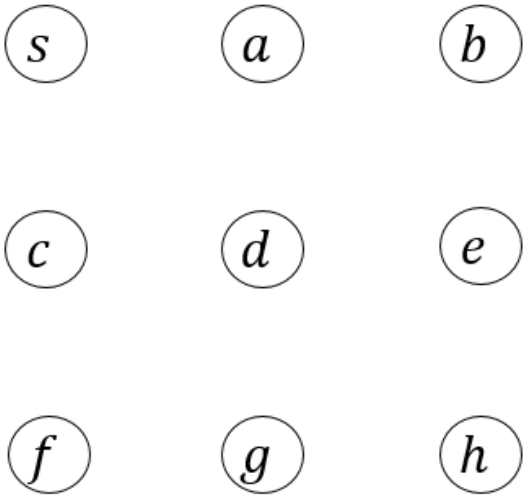
(2 pts) Add all the ***forward edges*** in the above.



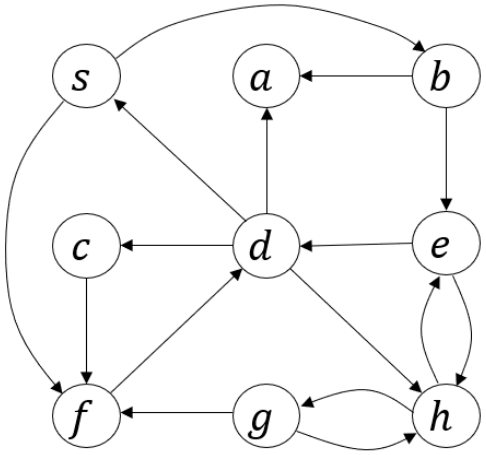
(2 pts) Add all the ***cross edges*** in the above.

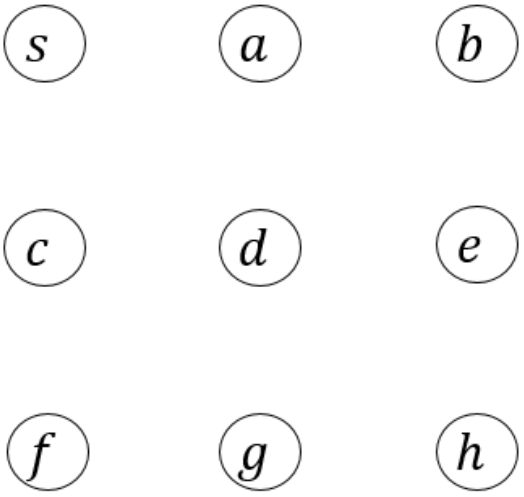
f) (2 pts) Find a ***simple*** cycle in the graph that consists of ***four*** edges. The graph is again copied over below on the left. Draw the cycle in the right figure below by including ***only the edges*** on the cycle.



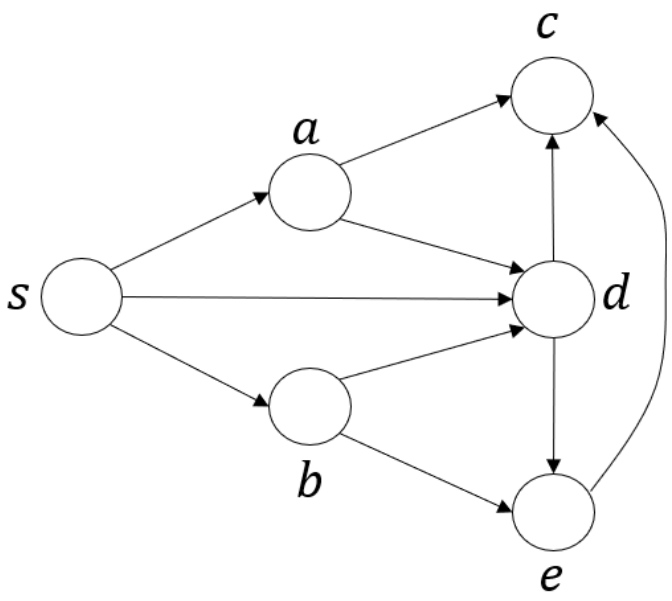


g) (5 pts) Perform a breadth-first search (BFS) on the same graph (shown below on the left) starting at the vertex s. Break a tie according to the ***alphabetical order***: a, b, c, d, e, f, g, h. In the right figure below, draw the ***BFS tree*** by adding only the edges each of which, when scanned, led to the discovery of a new vertex.



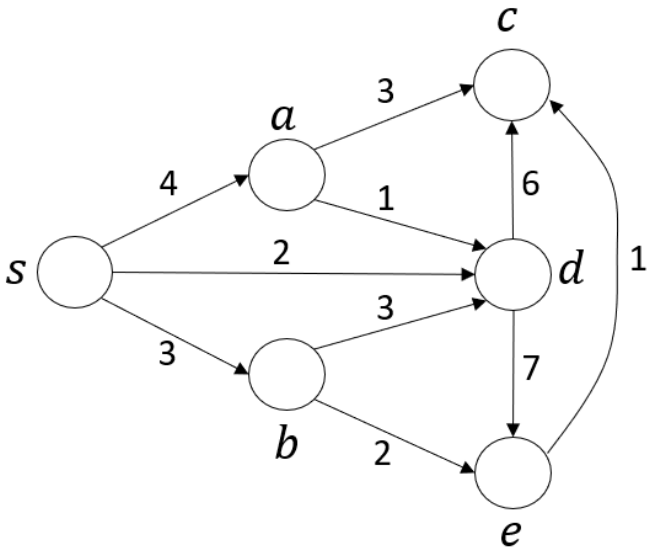


3. (14 pts) Consider the following directed acyclic graph *G*.



a) (4 pts) Give a topological sort of *G* by filling out the following table.

b) (10 pts) Now we add a weight to every edge in *G* as shown below.



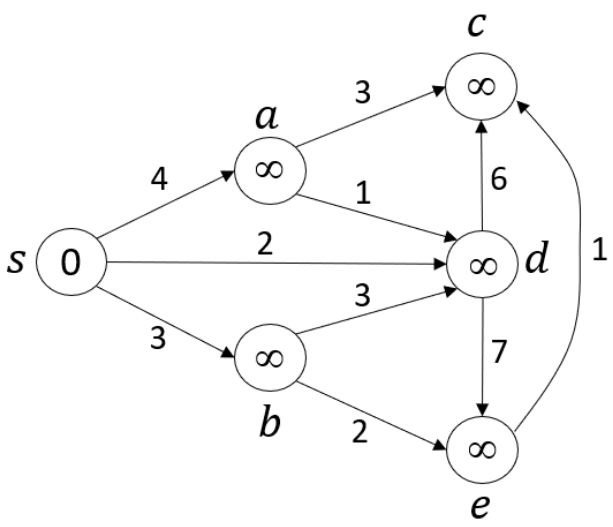
Use Dijkstra's algorithm to find ***all shortest paths*** from the source node s in G using the seven pre-drawn templates starting at the bottom of this page. At each step,

- write the distance label (d-value) of every node that is not s inside the node,
- ***circle*** the node v that is selected at that step, and
- either mark or darken or double (or draw a line wiggling around) ***every*** edge starting at v, and, if necessary, update the distance label (d-value) of the edge's destination node.

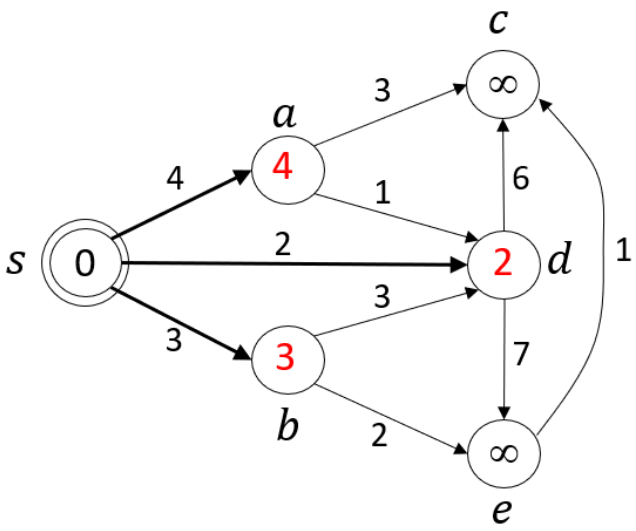
For your reference, we show the initial d-values in the first template, as well as the outcome of the first iteration in the second template. In the second template, the node s is marked and so are the three edges coming out of this vertex. The d-values of their destination nodes a, b, d are also updated.

You need to fill out the remaining five templates. (Note that the final step of the algorithm will result in no change of the distance.)

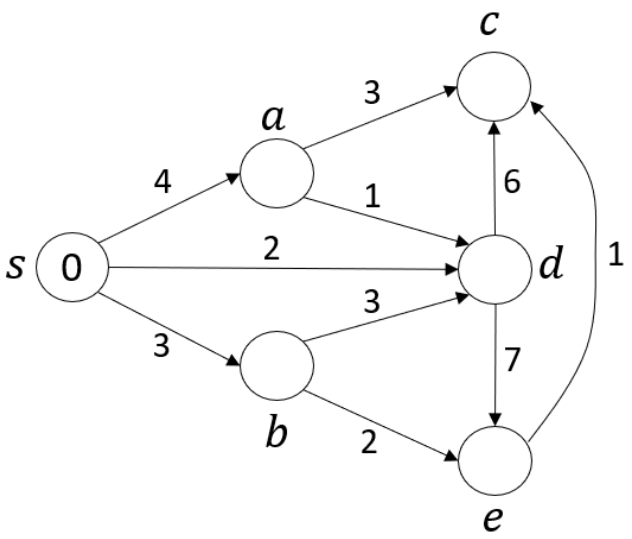
Only partial credit will be awarded if you just give the final answer and/or shortest path tree.



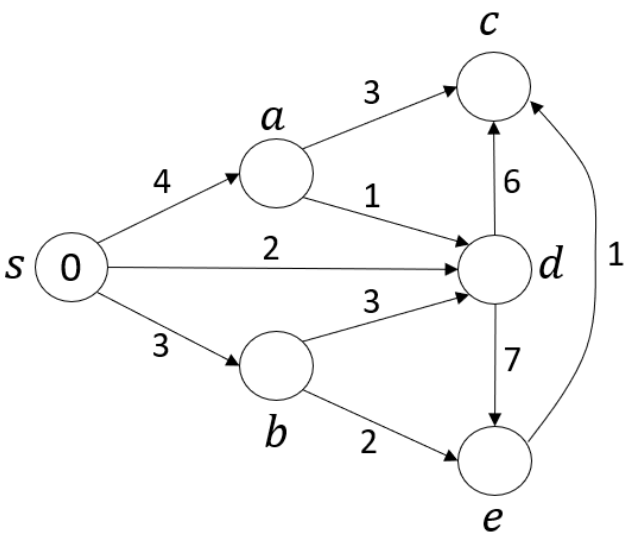
(1)



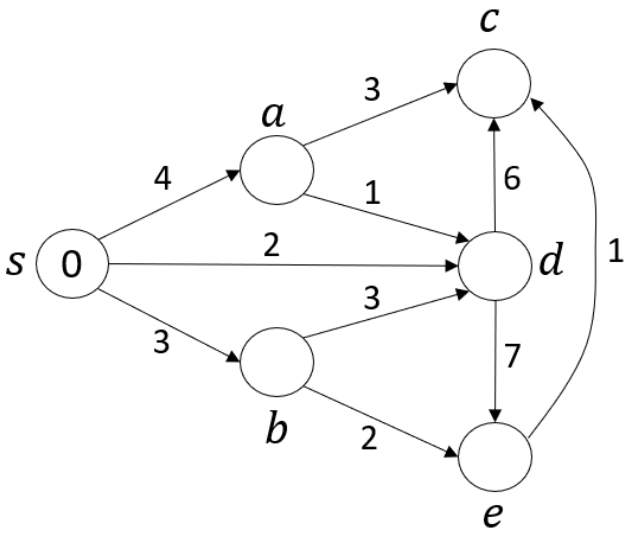
(2)


c
a
4
3
1
6
s
0
2
d
1
3
3
7
b
2
e

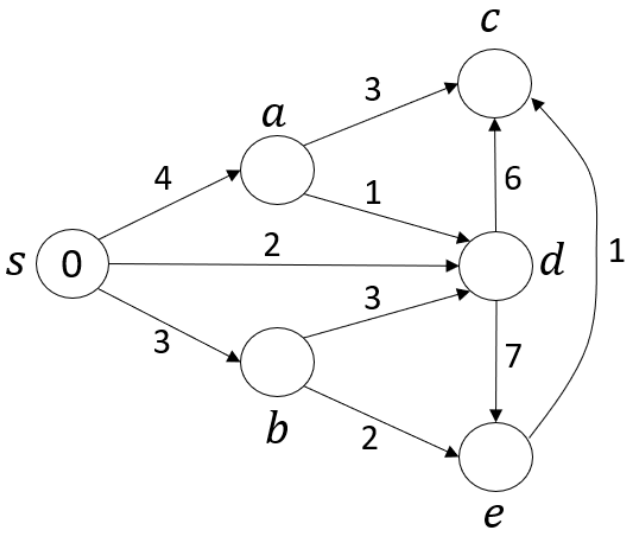
(3)


c
a
4
3
1
6
s
0
2
d
1
3
3
7
b
2
e

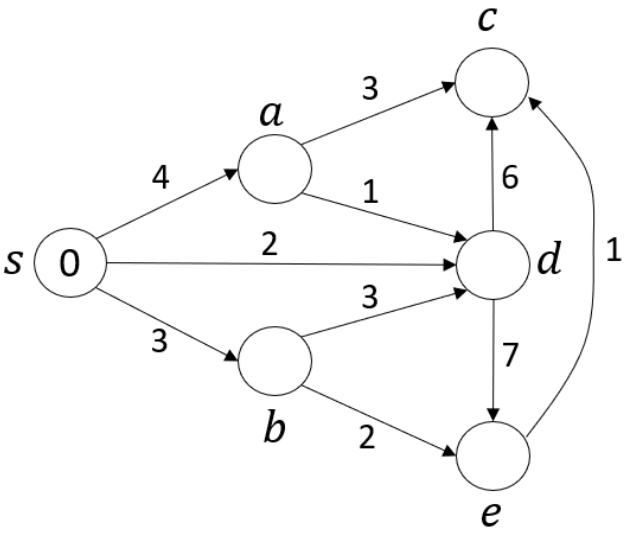
(4)


c
a
4
3
1
6
s
0
2
d
1
3
3
7
b
2
e

(5)


c
a
4
3
1
6
s
0
2
d
1
3
3
7
b
2
e

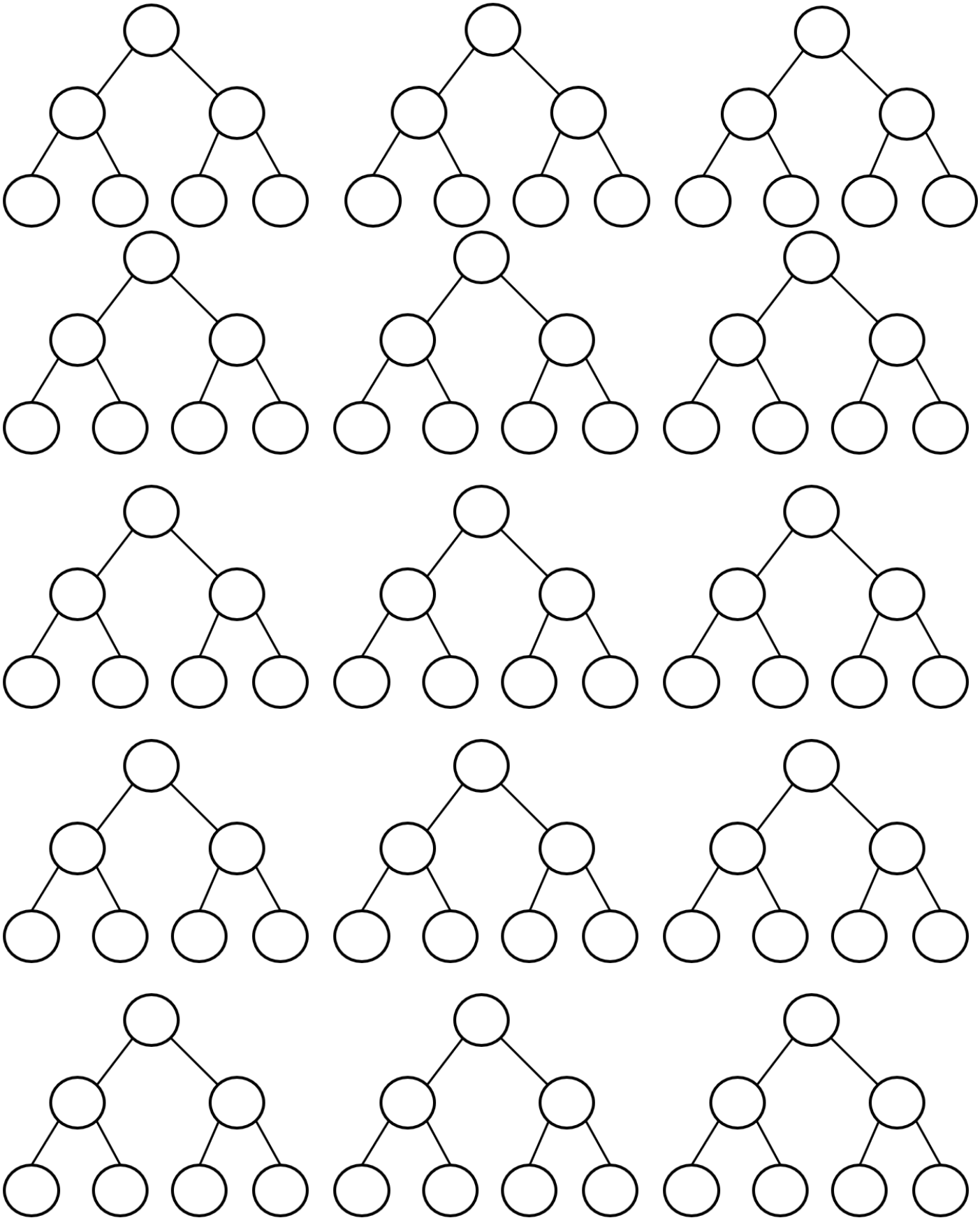
(6)


c
a
4
3
1
6
s
0
2
d
1
3
3
7
b
2
e

(7)

4. (14 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. Fill in the missing lines and underline the first line that is a heap (in other words, demarcate the transition from `Heapify` to the sort proper). 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.***

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



5. (10 pts) Give the tightest-possible asymptotic complexities of the following operations. ***Use the correct big-O notation!***

a) Finding the node containing a key in an arbitrary binary tree with n nodes.

b) Finding the node containing a key in a binary search tree with n nodes.

c) Determine the ***exact*** height of a balanced binary search tree with n nodes.

d) Carrying out a sequence of n operations, each either an addition or a removal, on a splay tree initially with $O(n)$ nodes.

e) Evaluating a postfix string with n operators and operands.

f) Using Graham's scan to construct the convex hull of n points in the plane.

g) Running Insertion Sort on n integers.

h) Searching for a number in an array of numbers in the non-decreasing order.

i) Detecting if a directed graph with V vertices and E edges has a cycle.

j) Finding the shortest path in a weighted directed acyclic graph with V vertices and E edges from a vertex u to another vertex v, assuming that at least one path exists.

6. (20 pts) Write a public method named **removeRoot()** for the binary search tree (**BST**) class below. The method deletes the root from the tree and returns the item stored there. It also ensures that the modified tree at the completion of the method is still a binary search tree by doing the following:

- The predecessor of the deleted root in the original tree will become the new root after the deletion.
- In case the root does not have a predecessor (i.e., it has no left subtree), its right child will become the new root.

All should be done via link updates. You are ***not allowed*** to copy the data stored at the predecessor to the root. You are ***not allowed*** to use any known method from the binary search tree class.

Note that for any node in the tree, its **data** item is known to be greater than all the **data** items stored in its left subtree, and less than all the **data** items in its right subtree.

```
public class BST<E extends Comparable<? super E>> extends AbstractSet<E>
{

  protected Node root;
  protected int size;

  protected class Node
  {
      public Node left;
      public Node right;
      public Node parent;
      public E data;

      public Node(E key, Node parent)
      {
            this.data = key;
            this.parent = parent;
      }
  }



  // other methods
  // ...


  public E removeRoot() throws IllegalStateException
  {

        E data;
        if (size == 0)
              throw new IllegalStateException("No root removal on an empty tree");

        data = root.data;
```

```
// BST has only one node.
if (size == 1)
{
    // insert code below (1 pt)




}

// BST has two or more nodes but no left subtree.
else if (root.left == null)
{
    // insert code below (3 pts)









}

// BST has two or more nodes and a left subtree.
// find the predecessor of the root and promote it to the new root.
else
{
      Node cur = root.left;
      Node prnt = root;

      // find the predecessor of the root.
      while (cur.right != null)
      {
          // insert code below (2 pts)









      }

      // left child of root has a right subtree.
      if (prnt != root)
      {
            // update links related to the predecessor's subtree(s).
            // insert code below (4 pts)
```

```
                // set up the new root's left subtree.
                // insert code below (3 pts)









            }

            // set up the new root's right subtree.
            // insert code below (4 pts)









            // set up the new root.
            // insert code below (2 pts)









        }

        // other updates if needed
        // insert code below (1 pt)









        return data;
    }

    // other methods and iteration implementation
    // ...
}
```

(Scratch only)

(Scratch only)