Com S 228
Spring 2018
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.

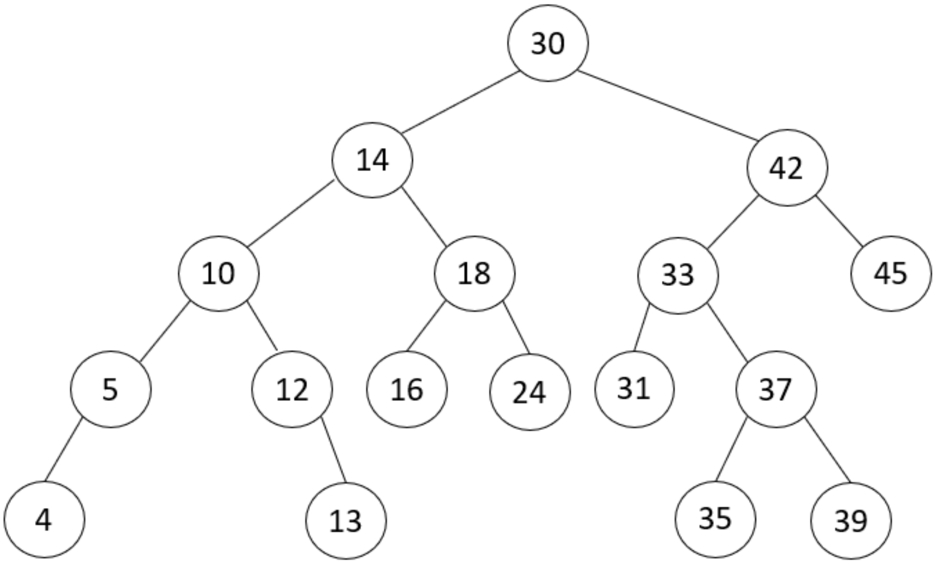
- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

If needed, you can peel off the last one of your exam sheets for scratch purpose.

If you have questions, please ask!

Question	Points	Your Score
1	17	
2	20	
3	16	
4	14	
5	10	
6	23	
Total	100	

1. (17 pts) All the questions in this problem concern the ***same*** splay tree storing 17 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 node 42 has height _____________.

e) (1 pt) The tree has height ______________.

f) (1 pt) The node 16 has depth ______________.

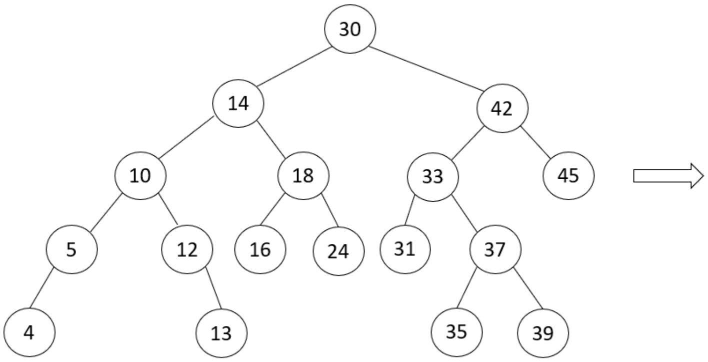
g) (1 pt) The node 24 has the successor node _______________.

h) (1 pt) The node 14 has the predecessor node _________________.

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

j) (4 pts) Insert 40 into the splay tree. Draw the tree at the end of the insertion. In the area to the right of the 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 in the area below and on the next page (if needed), ending with the spay tree after the insertion.

To save time, you may simply draw a node without the circle. 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.



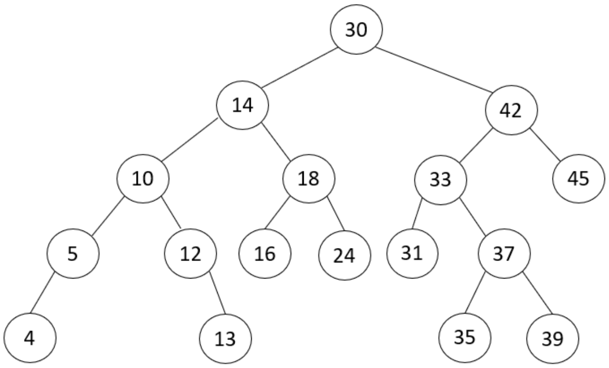
⇨

⇨

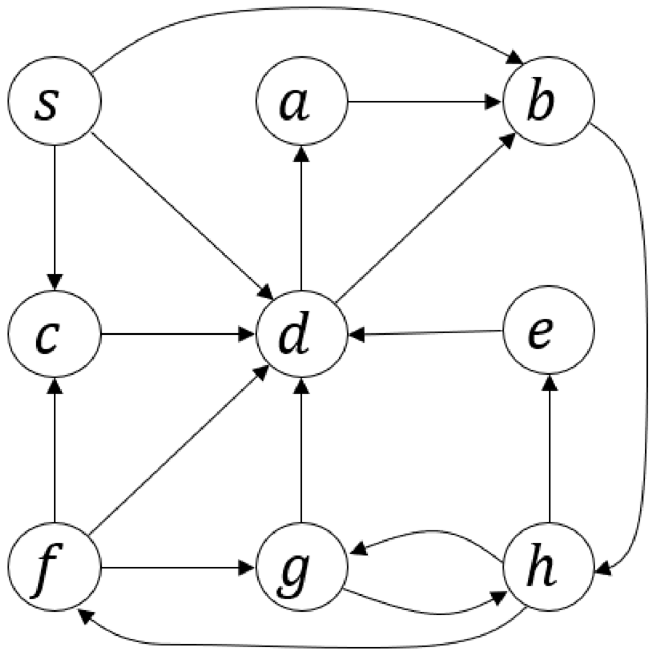
⇨

⇨

k) (3 pts) Delete 42 from the original splay tree. In the area to the right of the 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) to the right and in the area below, ending with the splay tree after the deletion.



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

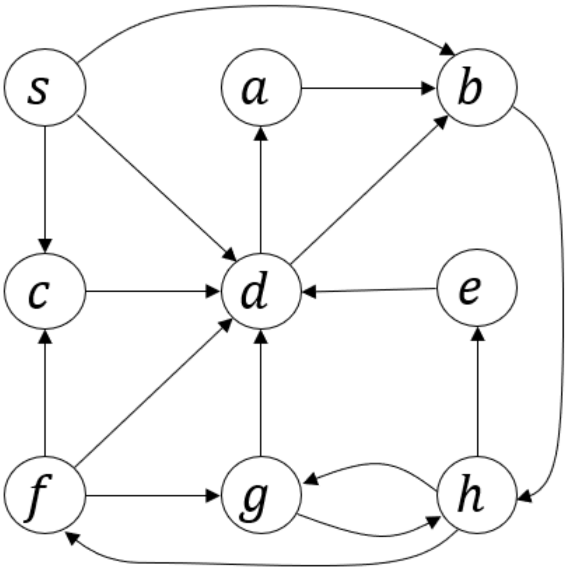


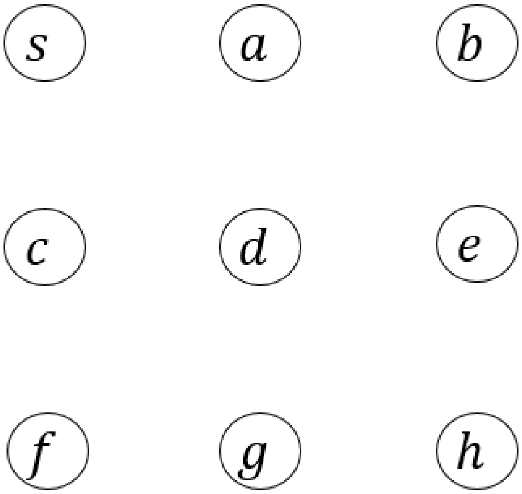
a) (1 pt) The in-degree of the vertex d is _____________.

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

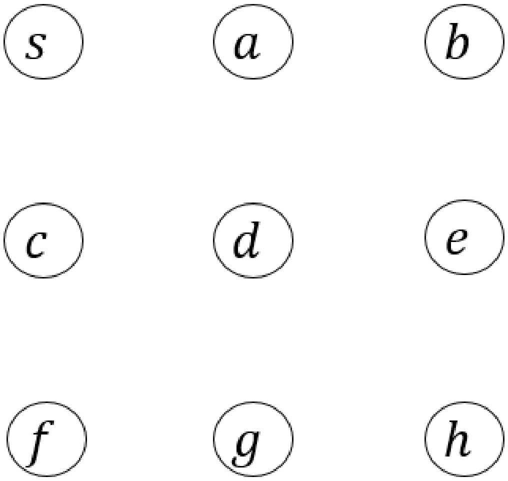
c) (1 pt) Is the directed graph strongly connected? (Yes/No) ______________.

d) (6 pts) Perform a depth-first search (DFS) on the same 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 tree*** or ***forest*** by adding edges to the vertices. Break a tie according to the ***alphabetical order***.

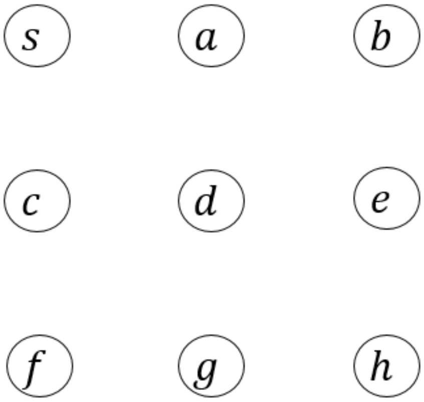




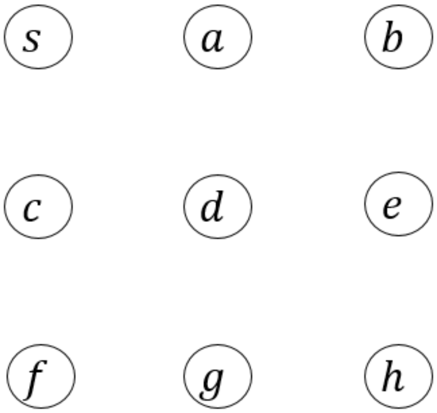
e) (3 pts) Draw all the back edges during the DFS carried out in d).



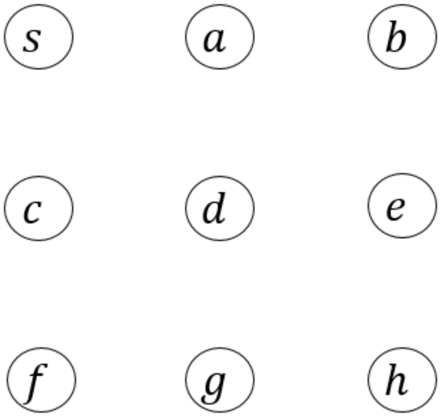
f) (3 pts) Draw all the forward edges during the same DFS.



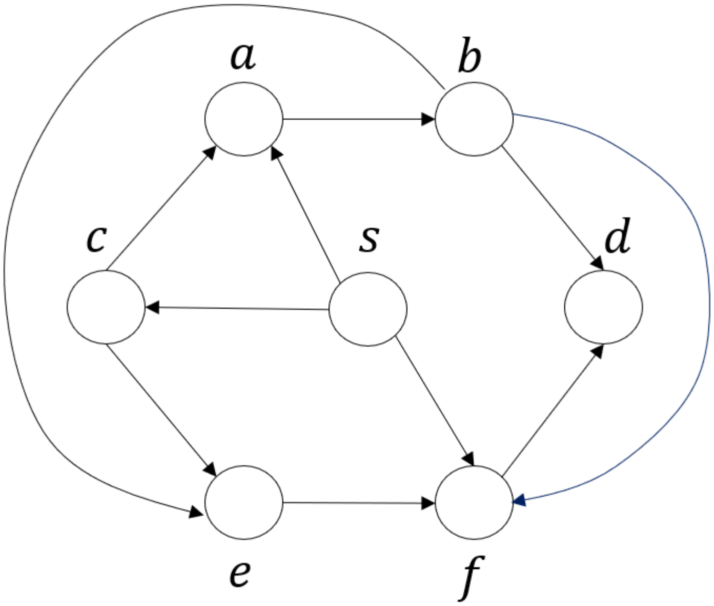
g) (3 pts) Draw all the cross edges during the same DFS.



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

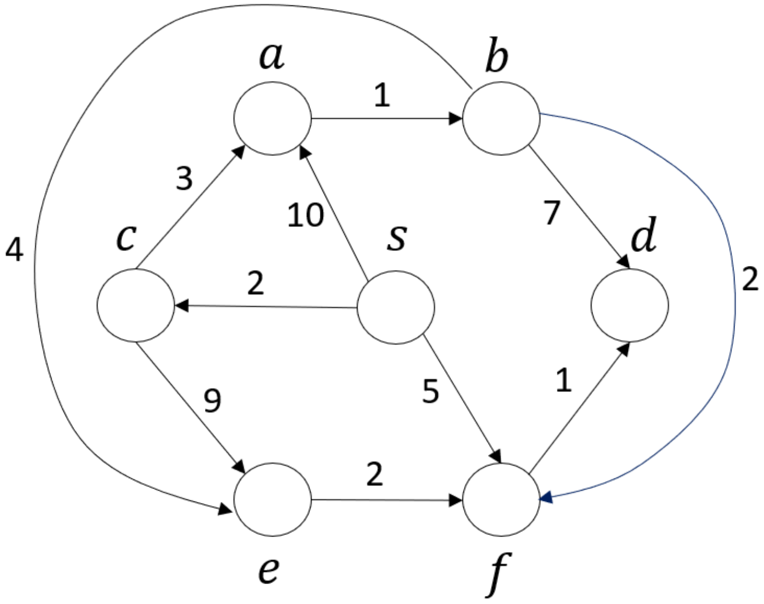


3. (16 pts) Consider the following directed acyclic graph *G*.



a) (5 pts) Give a topological sort of *G* by filling out the following table.

b) (11 pts) Now we add a weight to every edge in *G* as shown below.



Find ***all shortest paths*** from the node s in G using the eight pre-drawn templates starting at the bottom of this page. Perform edge relaxations according to the topological order of vertices that you found in part a). More specifically, for every vertex v chosen in the order, check each of its outgoing edges to see if the edge would result in a shorter distance from s to the edge's destination vertex. Then, do the following on the next template:

- ***circle*** the node v that is selected at that step, and

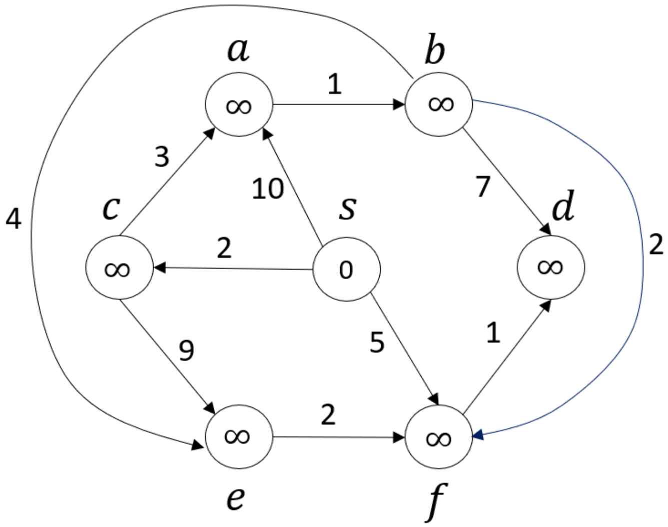
- either mark or darken or double (or draw a line wiggling around) ***every*** edge e starting at v, and, if necessary, update the distance label (d-value) of the edge's destination node, and

- add the distance label of every other node ***inside*** the node.

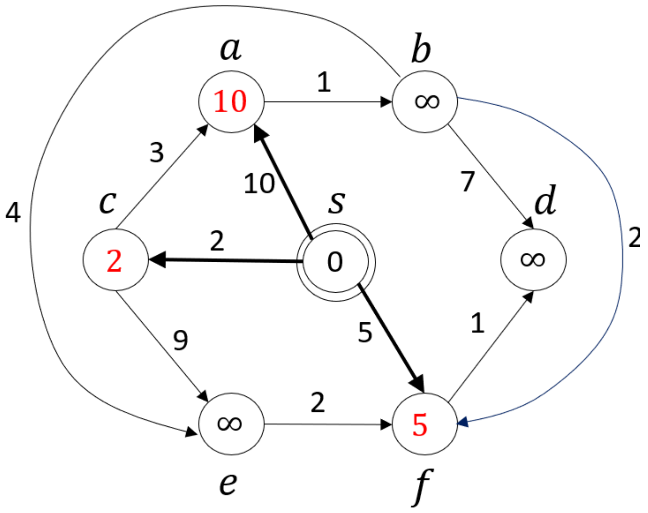
For your reference, we show the initial d-values in the first template, as well as the outcome of the first iteration in the second template. In the second template, the node s is marked and so are the three edges coming out of this vertex. The d-values of their destination nodes a, c, f are also updated.

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

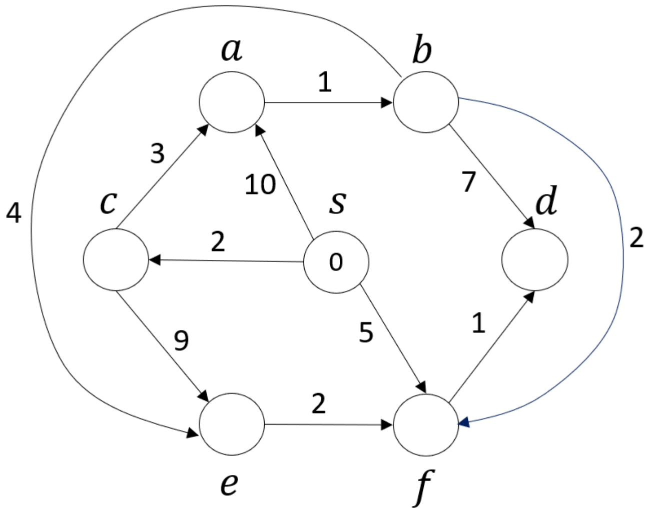
Only partial credit will be awarded if you just give the final answer and/or shortest path tree.



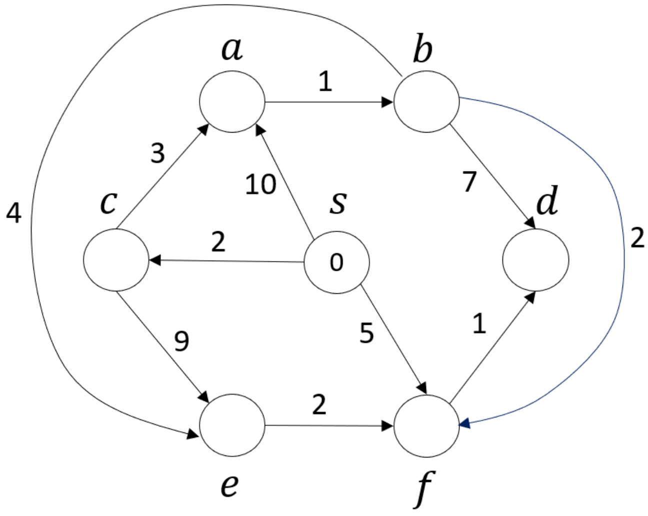
(1)



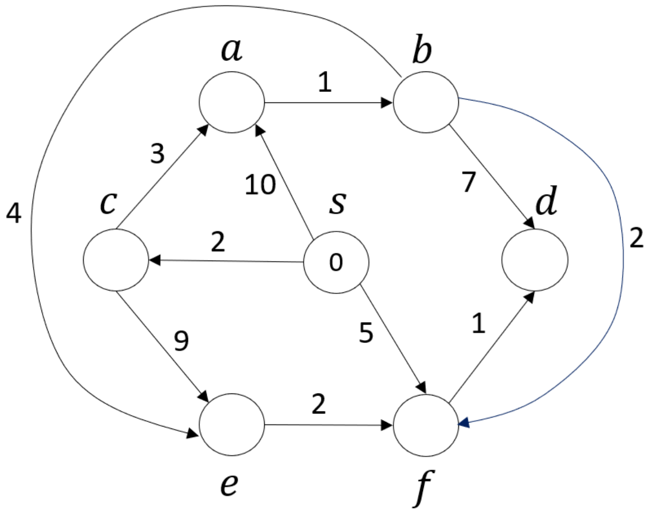
(2)


a
b
1
3
10
s
7
d
4
c
2
0
2
9
5
1
2
e
f

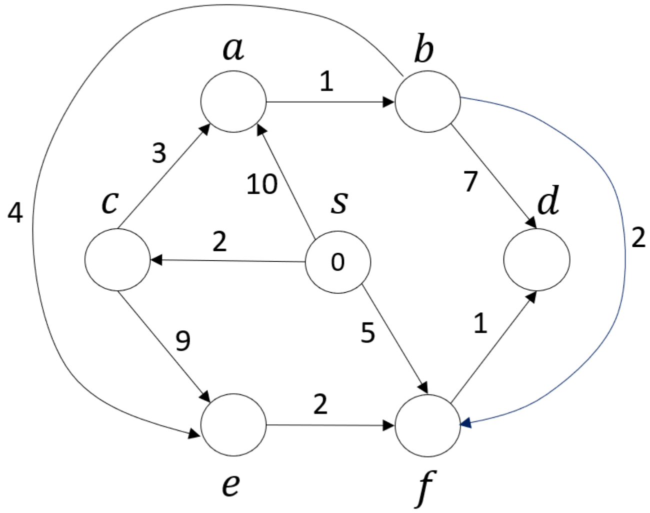
(3)


a
b
1
3
10
s
7
d
4
c
2
0
2
9
5
1
2
e
f

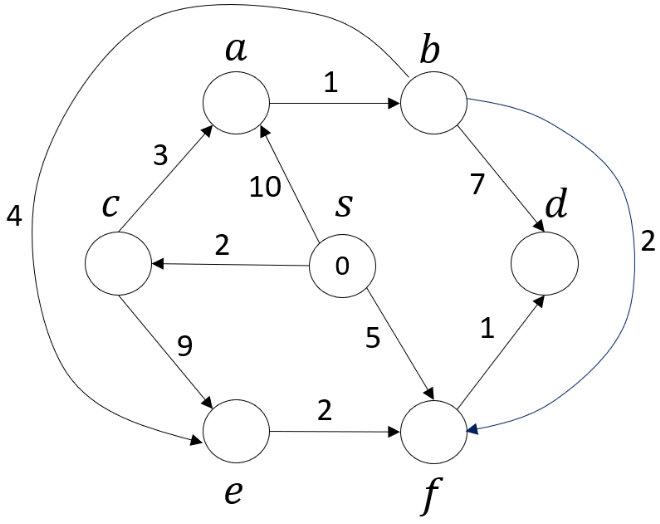
(4)


a
b
1
3
10
s
7
d
4
c
2
0
2
9
5
1
2
e
f

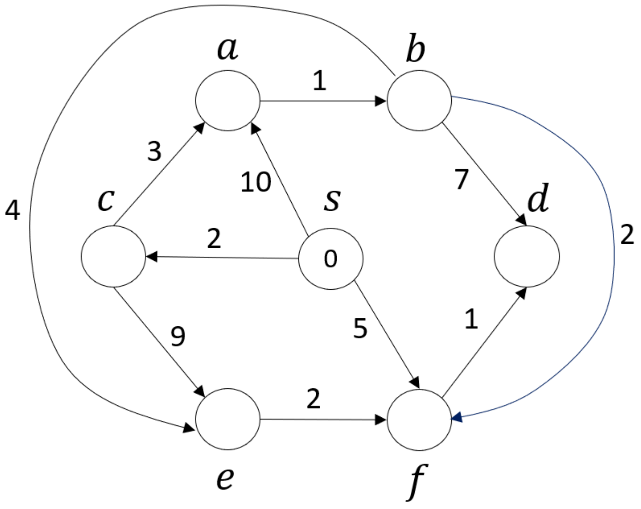
(5)


a
b
1
3
10
s
7
d
4
c
2
0
2
9
5
1
2
e
f

(6)


a
b
1
3
10
s
7
d
4
c
2
0
2
9
5
1
2
e
f

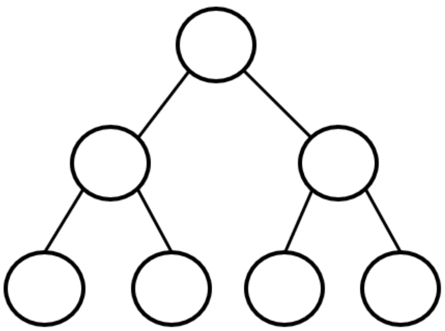
(7)

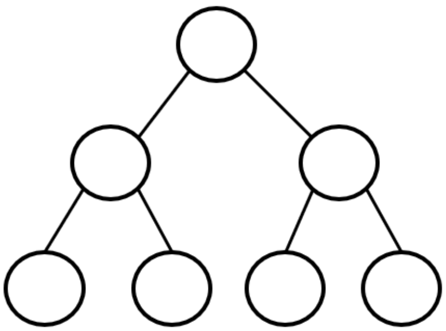

a
b
1
3
10
s
7
d
4
c
2
0
2
9
5
1
2
e
f

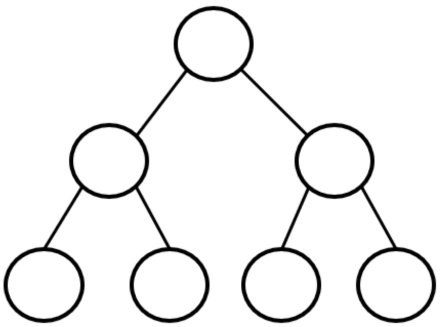
(8)

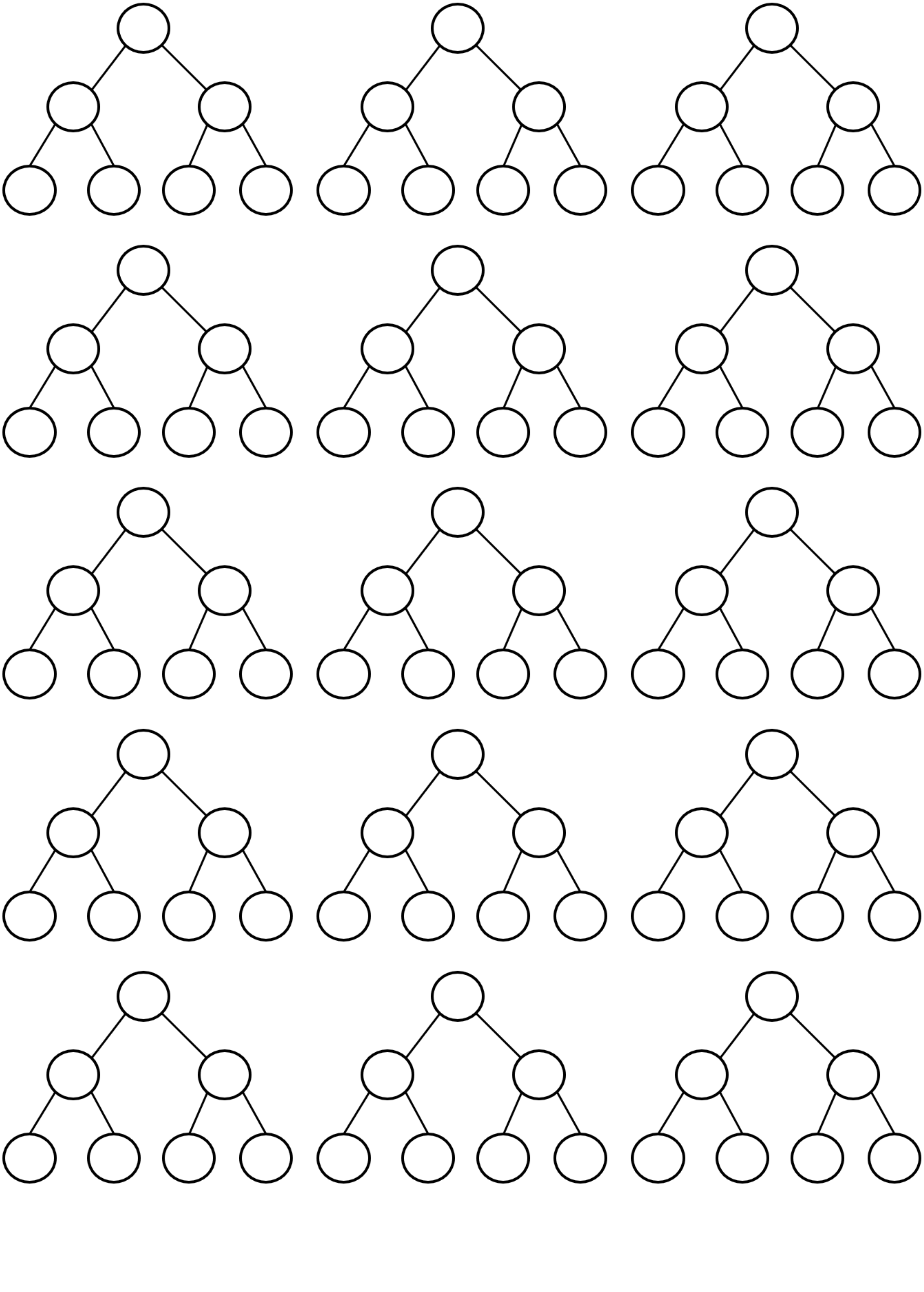
4. (14 pts) The following table illustrates the operation of HeapSort on the array in row 0. Every subsequent row is calculated by performing ***a single swap*** as determined by HeapSort on the preceding line. Fill in the missing lines and underline the first line that is a heap (in other words, demarcate the transition from Heapify to the sort proper). You may use the empty trees below and 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	1	3	6	5	2	0	4
1							
2	6	5	1	3	2	0	4
3							
4	1	5	4	3	2	0	6
5							
6	5	3	4	1	2	0	6
7							
8	4	3	0	1	2	5	6
9	2	3	0	1	4	5	6
10							
11	1	2	0	3	4	5	6
12							
13	0	1	2	3	4	5	6
14							
15	0	1	2	3	4	5	6









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

a) Traversing from the deepest leaf to the root in an arbitrary binary tree with n nodes.

b) Traversing from the deepest leaf to the root in a balanced binary tree with n nodes.

c) Removing the smallest item from a binary min-heap with n items.

d) Finding an item in a perfect hash set with n items.

e) Inserting an item into a hash set holding n keys and implemented with list-based buckets.

f) Heapifying an array of n elements.

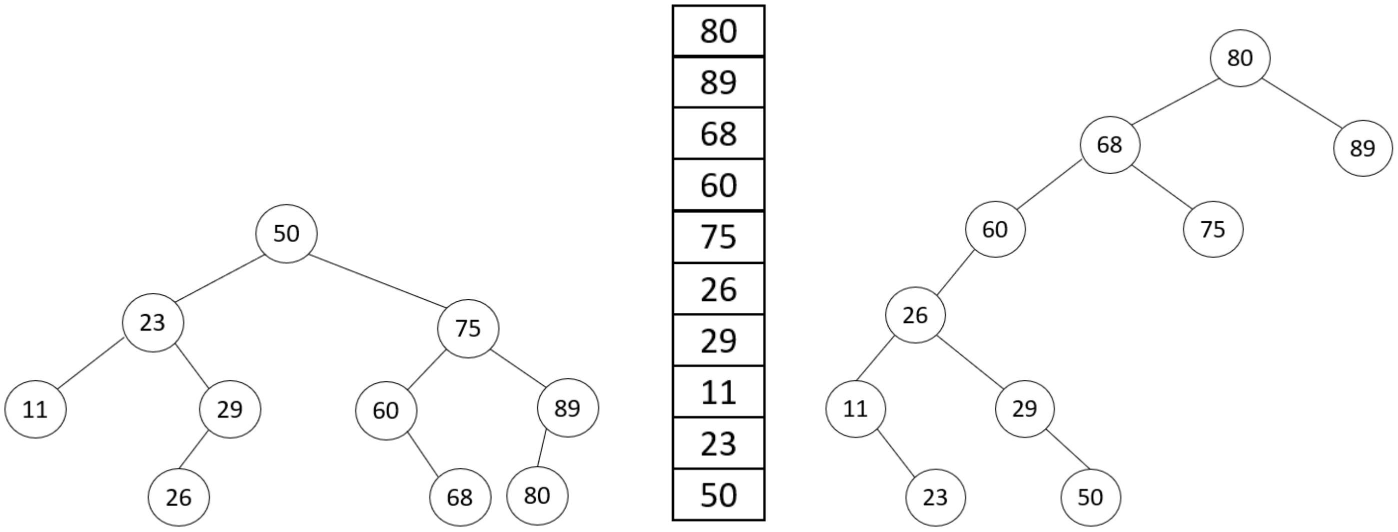
g) Traversing every edge in a connected, undirected graph with V vertices and E edges, built with adjacency lists.

h) Joining two splay trees, both with n nodes.

i) Counting the edges in a complete graph (i.e., a graph in which every two vertices share an edge) with V vertices.

j) Your score on this exam out of 100 points.

6. (23 pts) In this problem, you are asked to perform the ***preorder*** traversal on a binary search tree (BST), and save the visited data items (i.e., keys) onto a stack during the traversal. Then you need to construct a new BST starting from an empty tree by popping keys out of the stack and inserting them into the tree. Note that the keys stored in the original BST are all ***distinct***. The figure below illustrates the operation. Parts (a), (b), (c) respectively display the original BST, the content of the stack after the preorder traversal, and the newly constructed BST.



(a) Original BST

(b) Stack

(c) New BST

To provide the above tree construction utility, the class BSTSet presented in our lectures will be augmented with the following three methods:

```
public BSTSet<E> rearrangeBST();
private void preorderOntoStack();
private void preorderOntoStackRec(Node n);
```

In order to employ a stack, we make use of the interface `PureStack` and the class `ArrayBasedStack`, both of which were also presented in our lectures. The interface is given only partially below up to `push()` and `pop()`, the only two stack-related methods you will need for the tree construction task.

```
public interface PureStack<E> {
      /**
       * Adds an element to the top of the stack.
       *
       * @param item  the element to be added
       */
      void push(E item);

      /**
       * Removes and returns the top element of the stack.
       *
       * @return the top element of the stack.
       * @throws NoSuchElementException
```

```
     *            if the stack is empty
     */
    E pop();

   …
}
```

To facilitate the use of a stack, we have added to the class BSTSet a private instance variable stk to reference an object of the class ArrayBasedStack. The class BSTSet is appended below up to its instance variables and inner Node class.

```
public class BSTSet<E extends Comparable<? super E>> extends AbstractSet<E> {
      protected Node root = null;
      protected int size = 0;

      private ArrayBasedStack<E> stk; // added stack variable

      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;
            }
      }

       …
}
```

a) (5 pts) Complete the implementation of the method preorderOntoStackRec() according to the javadocs and comments, which should be ***read carefully before you add code***. (You need only fill in the blank space below the line of comment // insert code below.) The method is called by the ***fully implemented*** method preorderOntoStack(), which is given first.

```
      /**
       * Perform a preorder traversal on the tree, pushing data items
       * onto the stack stk in the order of visit.
       *
       * Precondition: root != null
       */
      private void preorderOntoStack(){
            stk = new ArrayBasedStack<E>();
            preorderOntoStackRec(root);
      }
```

```
/**
 * This recursive method performs preorder traversal on a subtree
 * rooted at the node n. Data stored at the node is pushed onto
 * the stack stk.
 *
 * Precondition: node != null
 *
 * @param n
 */
private void preorderOntoStackRec(Node n) {
      // insert code below (5 pts)














}
```

b) (18 pts) Complete the implementation of the method `rearrangeBST()` that constructs a new BST from the stack `stk` storing the keys retrieved during the preorder traversal. Fill in the blank below each comment line `// insert code below` or `// complete code below`. ***You must not use the*** `add()` ***method*** provided for the `BSTSet` class. Read all lines of comment carefully.

```
/**
 * Conduct the preorder traversal of this tree object, pushing
 * data items onto the stack stk. Then generate a new BST from
 * scratch by popping the data items out of the stack and
 * adding them one by one.
 *
 * @return rearranged tree copy
 */
public BSTSet<E> rearrangeBST() {
      // initialize an empty new tree.
      // complete code below (1 pt)

      BSTSet<E> newTree =                                ;

      // if this tree object is empty, then return newTree right away.
      // insert code below (1 pt)
// call preorderOntoStack().
// insert code below (1 pt)




// pop a data item from the stack and store it at the root
// of newTree. next, initialize its size variable.
// insert code below (2 pts)






// pop data items from the stack one by one and add them to
// newTree.
while (!stk.isEmpty()) {
      // pop a data item from the stack.
      // complete code below (1 pt)

      E key =                              ;

      // add key to newTree using binary search tree insertion.

      // initialize the current node.
      // complete code below (1 pt)

      Node current =                            ;

      while (current != null) {

            // compare key and the data stored at the current node.
            // complete code below (2 pts)

            if (                                              < 0) {

                  // check beforehand if current has no child down the
                  // direction corresponding to the above condition.
                  // complete code below (1 pt)

                  if (                                            ) {

                        // perform a needed tree update. afterward, set
                        // current to null to exit the inner while
                        // loop.
                        // insert code below (2 pts)
```

```
                } else {
                    // update current
                    // insert code below (1 pt)



                }
            } else {
                // check if current has no child down the other
                // direction.
                // complete code below (1 pt)

                if (                                          ) {

                    // perform a needed tree update. afterward, set
                    // current to null.
                    // insert code below (1 pt)



                } else {
                    // update current
                    // insert code below (1 pt)



                }
            }
        }

        // update size after insertion.
        // insert code below (1 pt)


    }

    // return the newly created tree.
    // insert code below (1 pt)


}
```

(Scratch only)

(Scratch only)