Com S 228
Fall 2015
Final Exam

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: _____

ISU NetID (username): _____

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

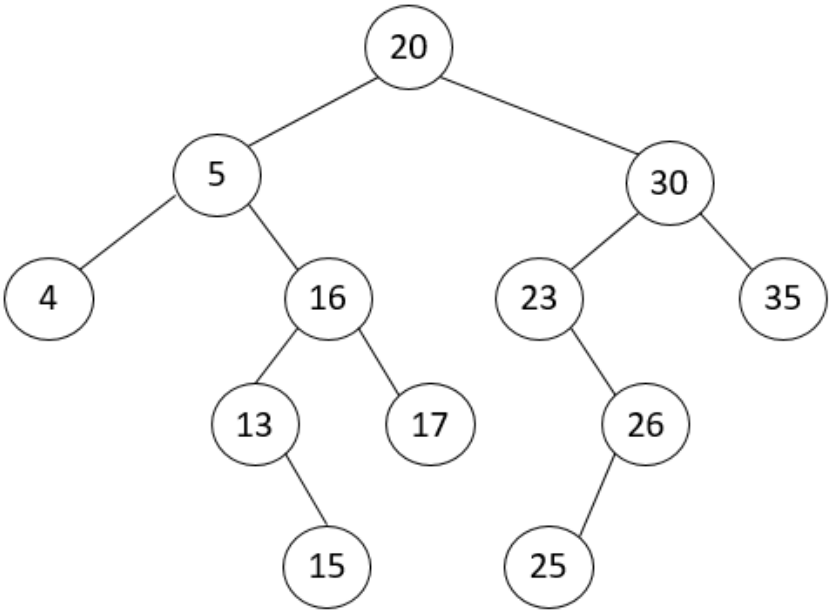
- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

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

If you have questions, please ask!

Question	Points	Your Score
1	15	
2	10	
3	15	
4	16	
5	8	
6	12	
7	24	
Total	100	

1. (15 pts) All the questions in this problem concern the ***same*** splay tree storing 12 integer values as shown below. While working on parts e) and f), 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.



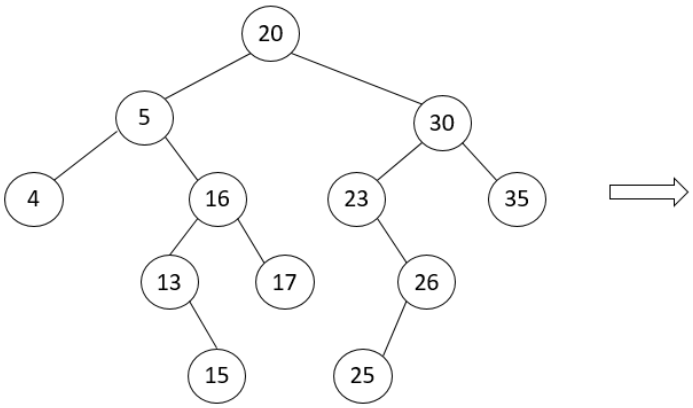
a) (1 pt) The above tree has height _____.

b) (1 pt) The node 16 has height _____.

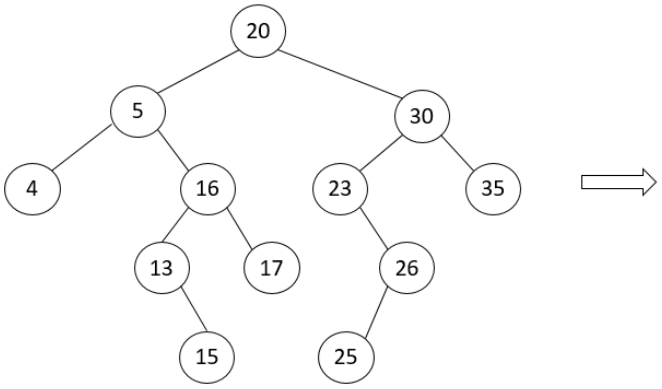
c) (1 pt) The node 26 has depth _____.

d) (2 pts) Output all the nodes in the postorder. (No splaying at any node.)

e) (5 pts) Insert 28 into the splay tree. (To save time, you may simply draw a node without the circle.)



f) (5 pts) Delete 16 from the same original splay tree.



2. (10 pts) A ***hash function*** $h(\,)$ for a set of keys takes every key k and computes an integer value $h(k)$ referred to as the key's ***hash code***.

a) (5 pts) Consider the set of seven integer keys 14, 12, 23, 11, 35, 40, 5, and the hash function below:

$$h(k) = (k - 3)\%\ 11$$

Note that % is the remainder operator. Fill in the table below the hash codes of these keys.

Key k	14	12	23	11	35	40	5
$h(k)$							

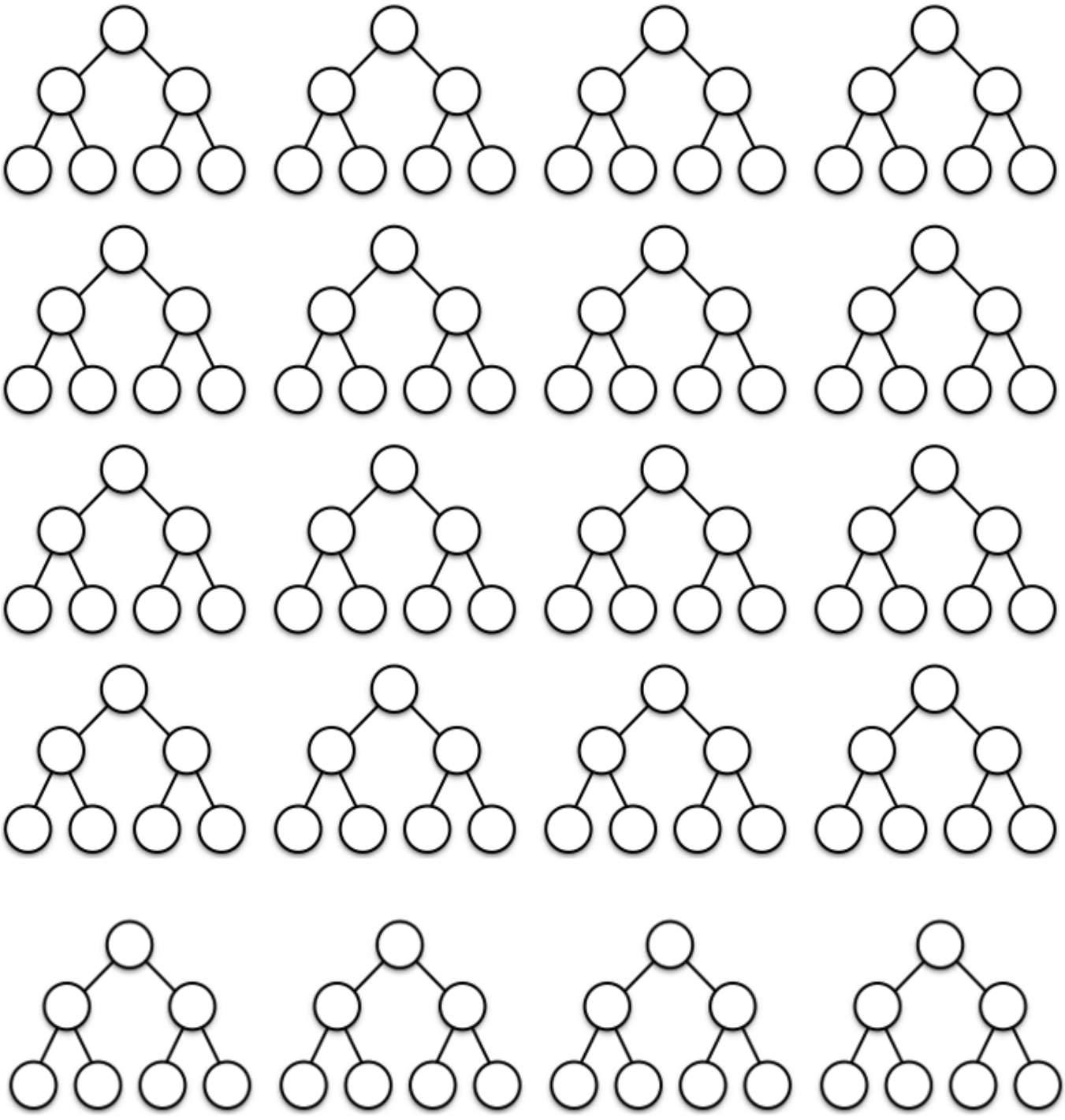
b) (5 pts) Suppose that an array of size 11 is used as the hash table to store the seven integers given in part a). Initially, every entry in the table is empty. The integers 14, 12, 23, 11, 35, 40, 5 are added sequentially to the table as follows. To add each integer k, we check the location indexed $h(k)$ in the table. Insert k at that location if it is empty. If the location is already occupied, then we "probe" the table for the first open slot. The search wraps around to index 0 after it probes the last entry. More precisely, we look for the ***smallest*** integer $i > h(k)$ such that the location indexed $\boldsymbol{i}\ \%\ \mathbf{11}$ is empty, and insert k there. This algorithm is called "***linear probing***".

Apply the linear probing algorithm to insert the integers 14, 12, 23, 11, 35, 40, 5 sequentially into the hash table (drawn below), based on their hash codes from the table in part a).

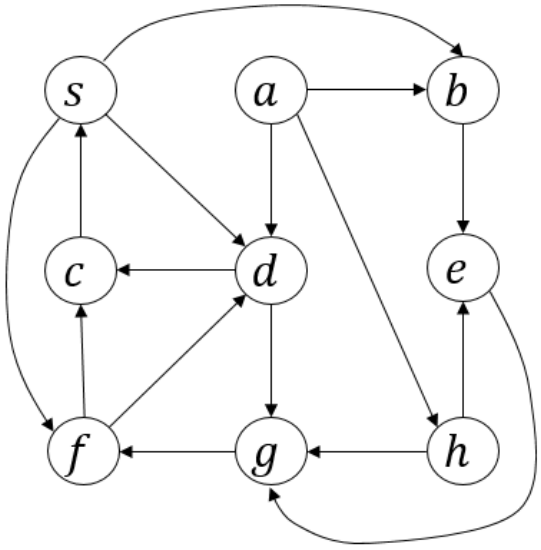
0	1	2	3	4	5	6	7	8	9	10

3. (15 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.***

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



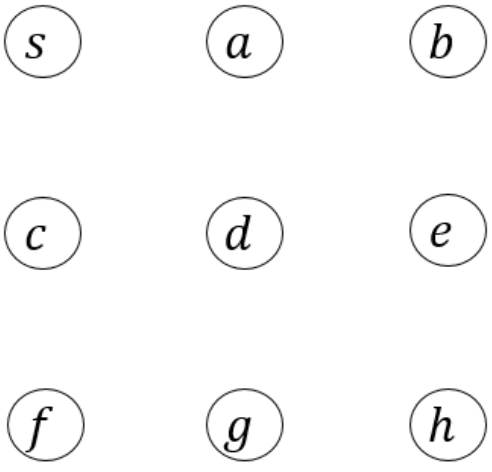
4. (16 pts) Consider the simple directed graph below with nine vertices.



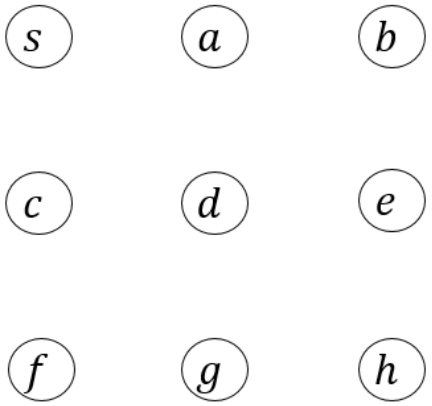
a) (1 pt) The in-degree of the vertex d is _____.

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

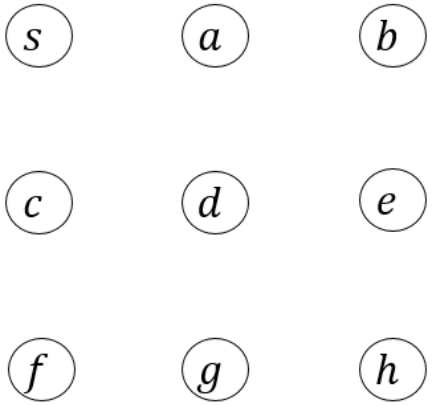
c) (8 pts) Perform a depth-first search (DFS) on the graph. Recall that the search is carried out by the method **`dfs()`** which iterates over the vertices, and calls a recursive method **`dfsVisit()`** on a vertex when necessary. Suppose that **`dfs()`** processes the vertices in the order $s, a, b, c, d, e, f, g, h$. In the below draw the DFS forest by adding edges to the vertices. Again, break a tie according to the ***alphabetical order***.



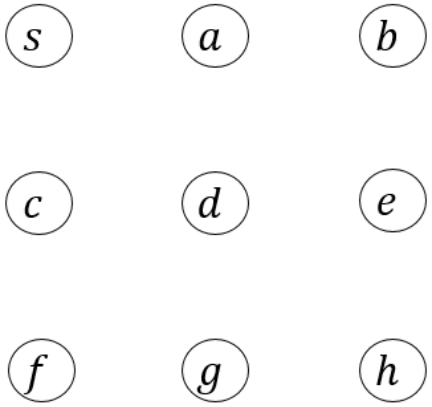
d) (2 pts) Draw all the back edges during the DFS carried out in (b).



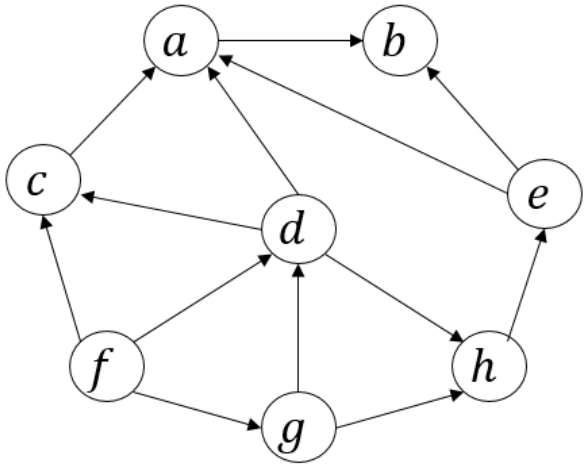
e) (2 pts) Draw all the forward edges during the same DFS.



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



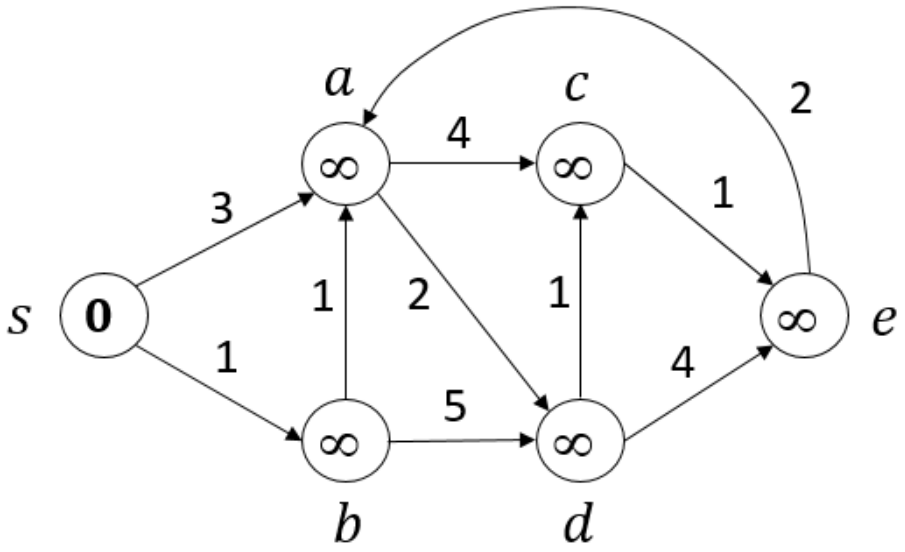
5. (8 pts) Consider the following directed acyclic graph *G*.



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

b) (3 pts) Now consider G to be any directed graph with m vertices and n edges. Suppose that the graph is represented by adjacency lists. An algorithm first checks if G contains a cycle, and in the case that it is acyclic, outputs a topological sort of G. In big-O notation give the total running time of this algorithm.

6. (12 pts) Consider the weighted graph *G* below.

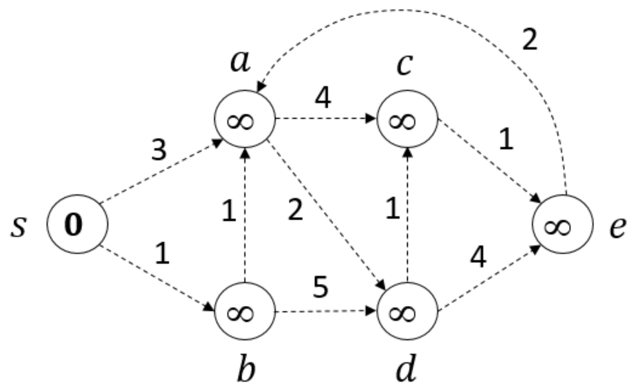


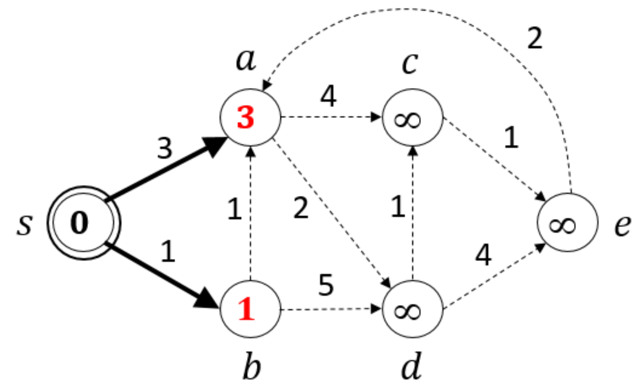
Show the execution of Dijkstra's algorithm for finding all shortest paths from node ***s*** in *G* using the pre-drawn templates on the next page. At each step,

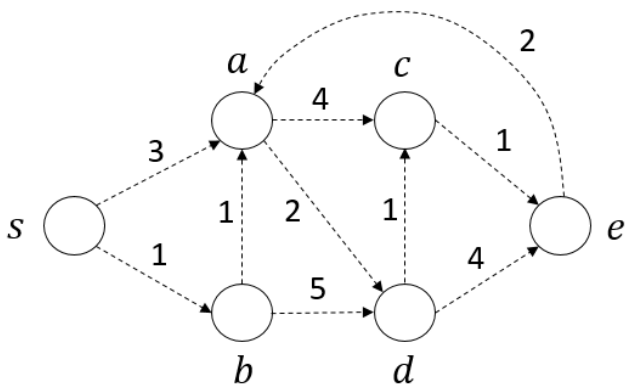
- write the ***distance label (d-value)*** of each node inside the node,
- ***circle the node*** that is selected at that step, and
- for each node v, ***darken the dashed edge*** from $\texttt{pred}(v)$ to v to make it a solid edge.

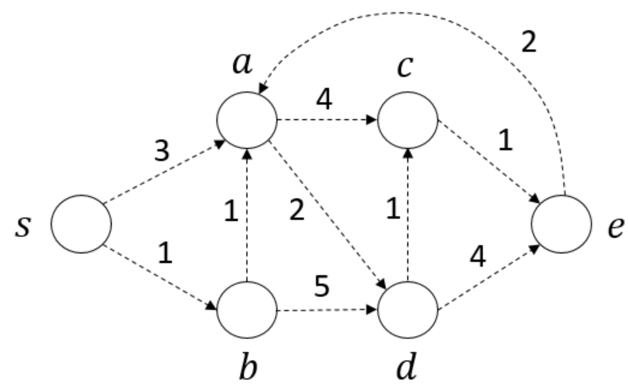
For your reference, we show the initial d-values, as well as the outcome of the first iteration. In the first step (shown in the second template on the next page), the node s is marked, and the two edges coming out of this vertex are darkened. You need to fill out the remaining templates. (Note that 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.)

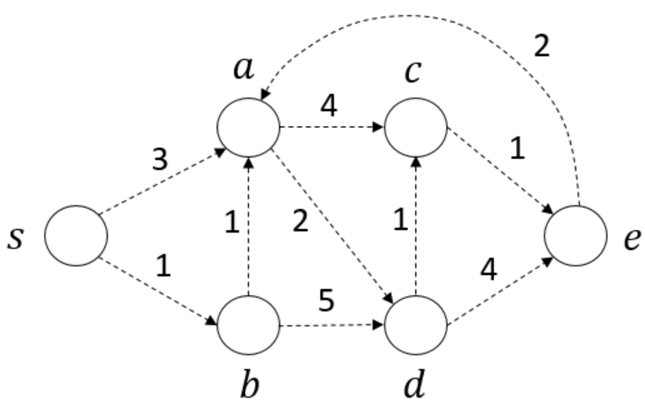
Partial credit will be awarded if you just give the final answer and/or shortest path tree.

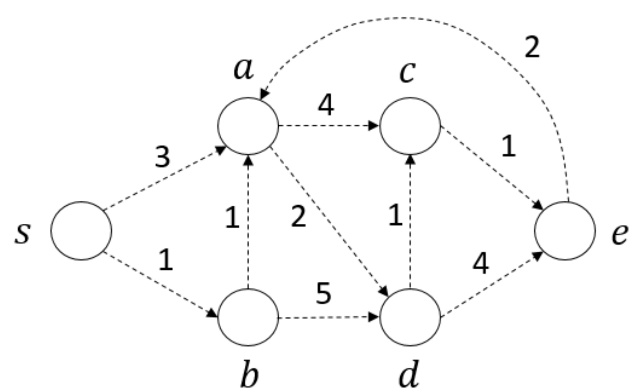

s
0
a
∞
c
∞
e
∞
b
∞
d
∞
3
1
4
1
2
1
1
2
5
4
2


s
0
a
3
c
∞
e
∞
b
1
d
∞
3
1
4
1
2
1
1
2
5
4
2


s
a
c
e
b
d
3
1
4
1
2
1
1
2
5
4
2


s
a
c
e
b
d
3
1
4
1
2
1
1
2
5
4
2

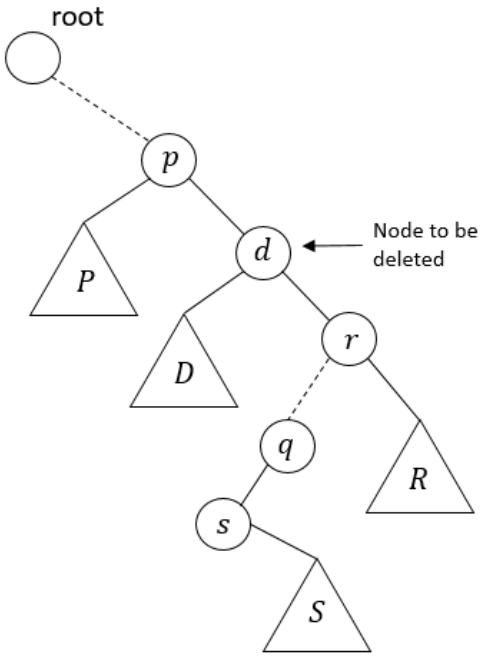

s
a
c
e
b
d
3
1
4
1
2
1
1
2
5
4
2


s
a
c
e
b
d
3
1
4
1
2
1
1
2
5
4
2

7. (24 pts) In this problem, you are asked to augment an existing implementation of the binary search tree (***BST***) class below.

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

  protected Node root;
  protected int size;

  protected class Node
  {
    public Node left;       // left child
    public Node right;       // right child
    public Node parent;
    public E data;          // key value

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


  // other methods
  // ...









}
```

a) (12 pts) Recall that ***deletion*** of a node d ***with both children*** is carried out by replacing the node with its successor node s.



On the left is an illustration for ***only*** the part of the tree relevant to this deletion. The following assumptions are made:

- Dashed lines represent paths not necessarily edges.
- The node d has a parent node p, and may be its either left or right child.
- The node d has a right child r.
- The left subtree of the node r is not empty.
- The successor node s has a parent node q.
- P, D, R, S denote subtrees that may or may not be empty.

Complete the ***code snippet*** below to carry out all the link updates involved in the deletion of the node d. Treat p, d, r, q, s (but not *P*, *D*, *R*, *S*) as reference variables of the Node class. The first line of the snippet is already completed for your reference.

```
// make the subtree S a new subtree of q
// insert code below (2 pts)
q.left = s.right;
```

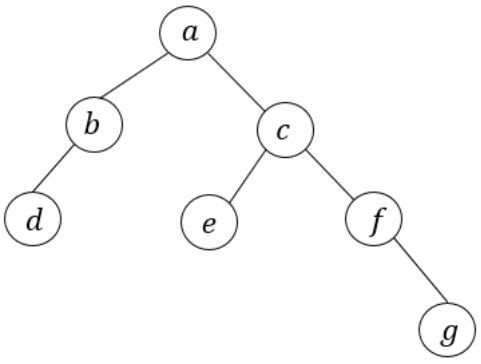
```
// make the left subtree of d the new left subtree of s
// insert code below (3 pts)
```

```
// make the right subtree of d the new right subtree of s
// insert code below (3 pts)
```

```
// make s the new child of the node p to occupy the original position of d
// insert code below (4 pts)
```

b) (12 pts) An internal node is called a ***one-child node*** if it has ***exactly one*** child. Add to the class a public method **`countOneChildNodes()`** counting the number of one-child nodes in the tree. The method calls a recursive private method **`countOneChildNodesRec()`** counting one-child nodes in a subtree rooted at its argument node. Implement these two methods on the next page.

For example, the tree shown below has two internal nodes b and f with one child each. So a call to **`countOneChildNodes()`** on the tree will return 2. The subtree rooted at c has only one internal node f with a single child. So a call to **`countOneChildNodesRec()`** on the node c will return 1.



```
    /**
     * @return number of internal nodes having one child
     */
    public int countOneChildNodes()
    {
        // insert code below (1 pt)




    }

    /**
     * @param  n
     * @return number of one-child nodes in the subtree rooted at n
     */
    private int countOneChildNodesRec(Node n)
    {
        // n is null (base case for recursion)
        // insert code below (1 pt)




        // count the one-child nodes in the left and right subtrees.
        // insert code below (4 pts)









        // return the count for the subtree rooted at n.
        // insert code below (6 pts)









}
```

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)