Com S 228
Spring 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.

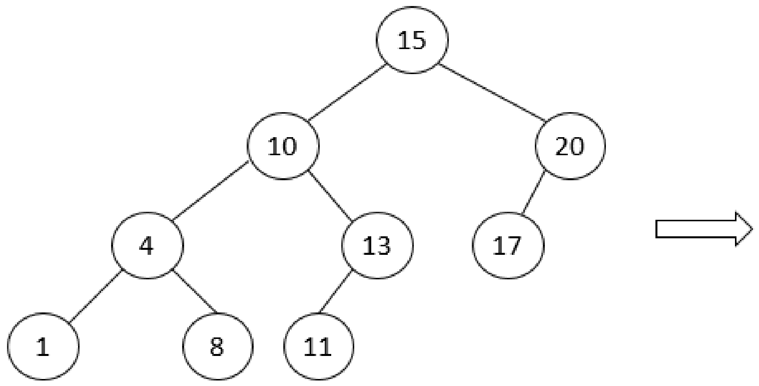
Please *peel off* the **last two** of your exam sheets for scratch purpose.

If you have questions, please ask!

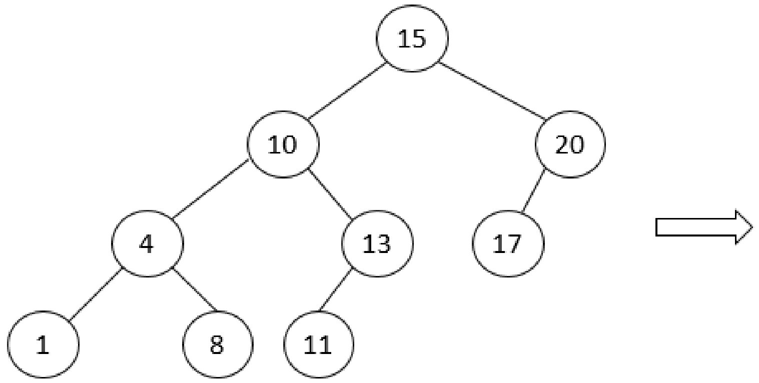
Question	Points	Your Score
1	15	
2	15	
3	10	
4	16	
5	7	
6	16	
7	21	
Total	100	

1. (15 pts) In this problem, you are asked to perform three operations ***separately*** on the ***same*** splay tree that stores integer values. To get partial credit from an incorrect final answer, we suggest you to draw the tree after the initial node operation and after each subsequent splaying step.

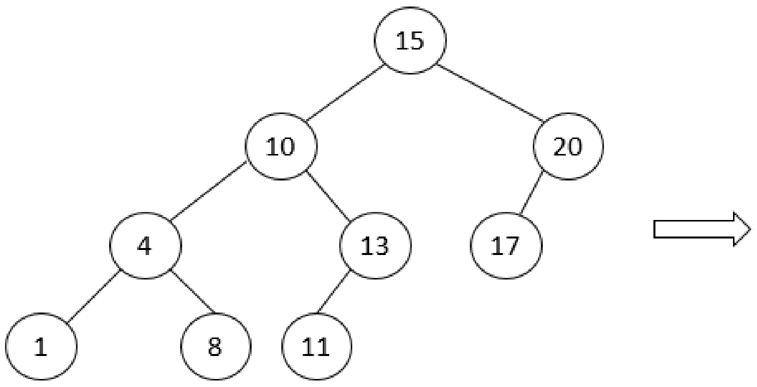
a) (5 pts) Insert 7 into the splay tree below.



b) (5 pts) Delete 15 from the same splay tree.

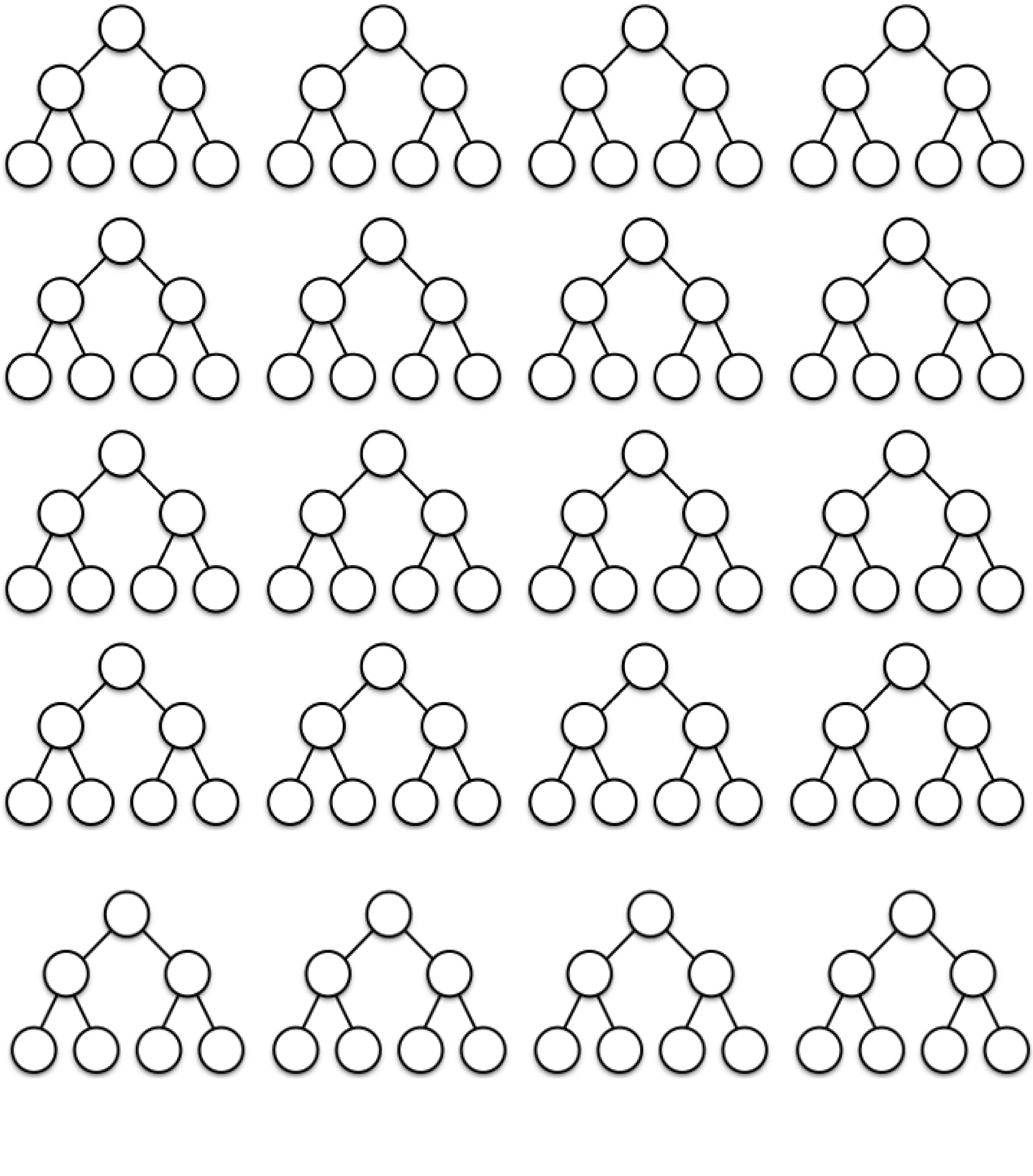


c) (5 pts) Search for 12. First, in the original tree, ***circle the node*** to be splayed at.



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



3. (10 pts) A ***perfect hash function*** $h(\,)$ for a set of keys yields ***different*** values on different keys.

a) (4 pts) Consider the set of integer keys 11, 20, 2, 8, 12, 16, and the hash function

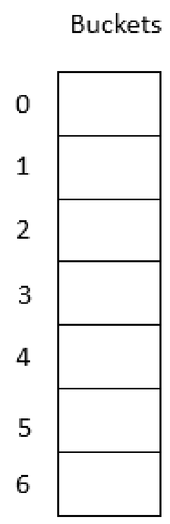
$$h(k) = (k+2)\%\,7$$

Note that % is the remainder operator. Compute the hash codes of these six keys, and fill them in the table below.

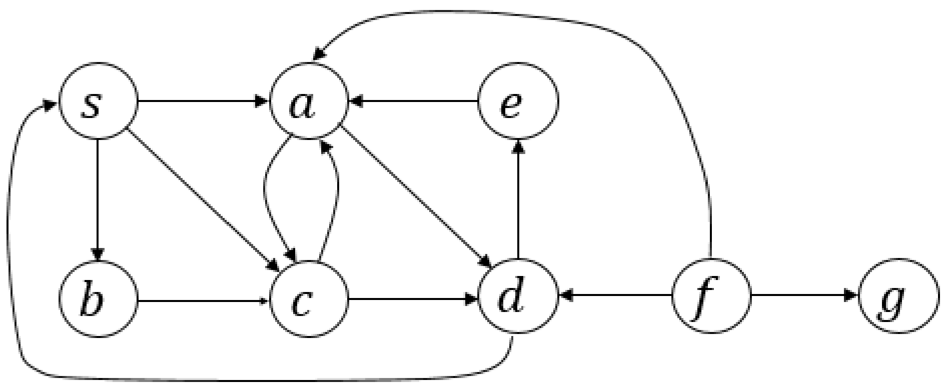
key	Hash code
11	
20	
2	
8	
12	
16	

b) (1 pt) Is $h()$ a perfect hash function?

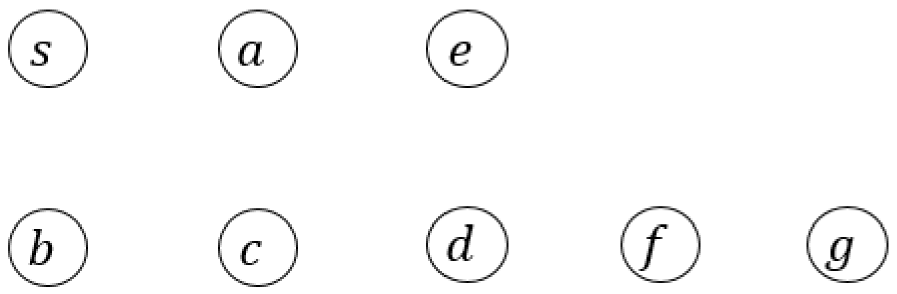
c) (5 pts) The hash table consists of 7 buckets, whose references are stored in the array below. Add the keys in the order 11, 20, 2, 8, 12, 16. Plot the contents of the buckets. What data structure is used to represent each bucket?



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

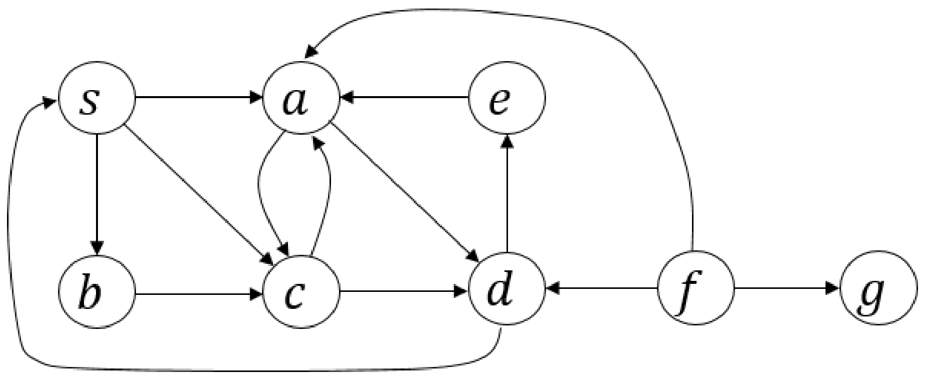


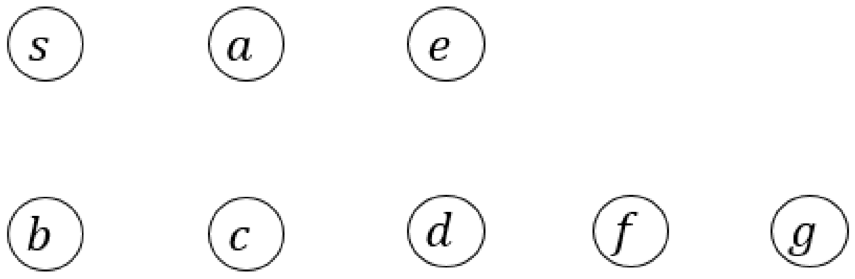
a) (6 pts) Perform a breadth-first search (BFS) starting at the vertex ***s***. Always choose a vertex in the ***alphabetical order*** in case of a tie. Draw the BFS tree by adding edges to the vertices below. Also, fill in the table the predecessor pred(v) of every vertex v on the path from s. Fill in `null` if the vertex does not have a predecessor.



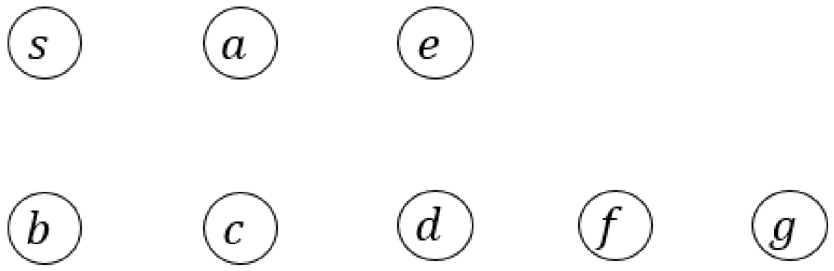
vertex v	*s*	*a*	*b*	*c*	*d*	*e*	*f*	*g*
pred(v)								

b) (6 pts) Perform a depth-first search (DFS). 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. In the below draw the DFS forest by adding edges to the vertices. Again, break a tie according to the ***alphabetical order***. (For scratch convenience, the same graph is copied over first.)



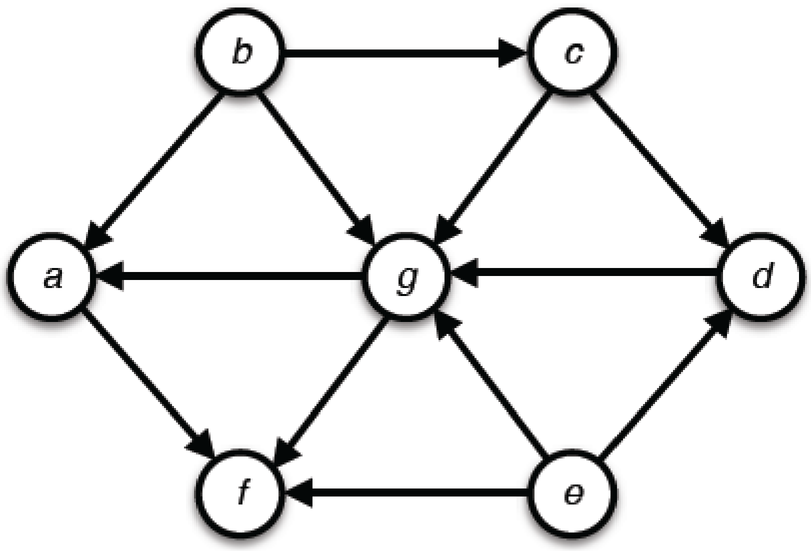


c) (3 pts) Draw all the back edges during the DFS carried out in (b).



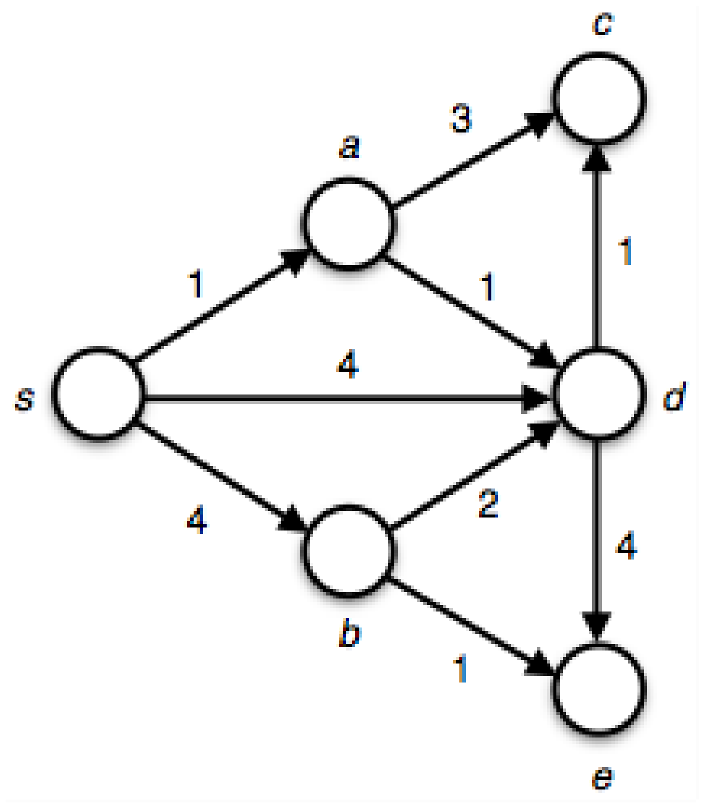
d) (1 pt) Is the graph acyclic? ____________ Yes ___________ No

5. (7 pts) Consider the following directed acyclic graph *G*.



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

6. (16 pts) Consider the weighted graph *G* below.

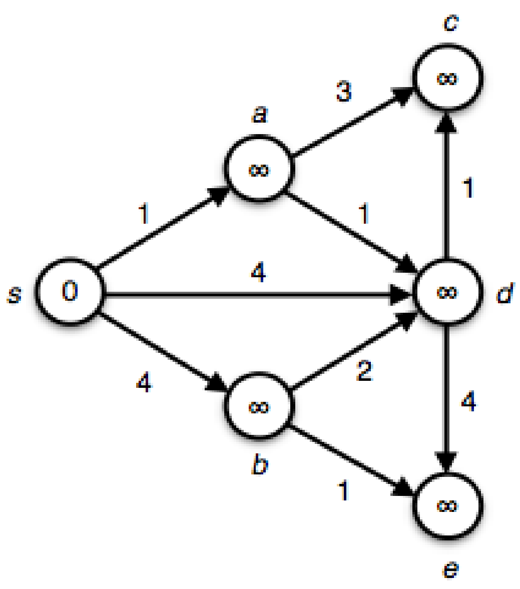


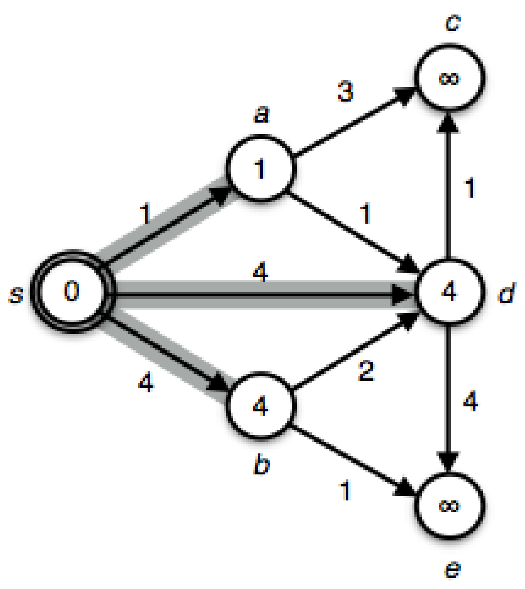
Show the execution of Dijkstra's algorithm for finding all shortest paths from node ***s*** in *G* using the pre-drawn templates on the next page. At each step,

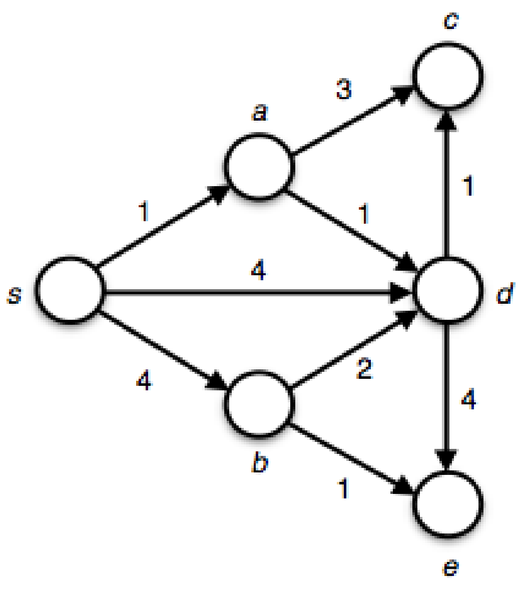
- write the ***distance label (d-value)*** of each node inside the node,
- ***circle the node*** that is selected at that step, and
- for each node v, either ***mark*** or ***darken*** or ***double*** (or ***draw a line wiggling around***) ***the edge*** from $\texttt{pred}(v)$ to v.

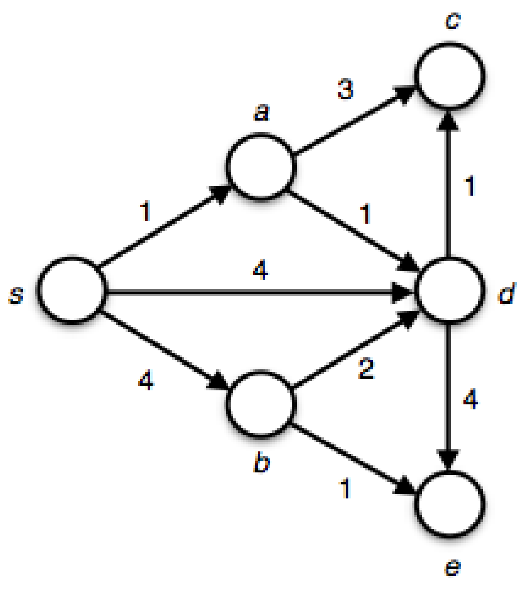
For your reference, we show the initial *d*-values, as well as the outcome of the first iteration. In the first step (shown in the second template on the next page), the node **s** is marked, so are the three edges coming out of this vertex. You need to fill out the remaining templates. (Note that the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

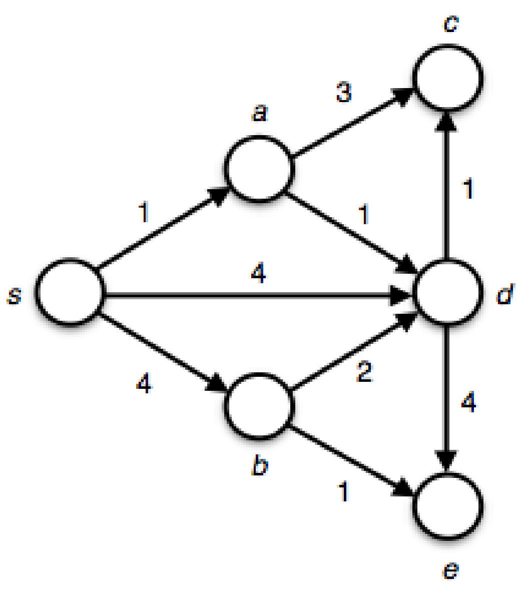
Partial credit will be awarded if you just give the final answer and/or shortest path tree.

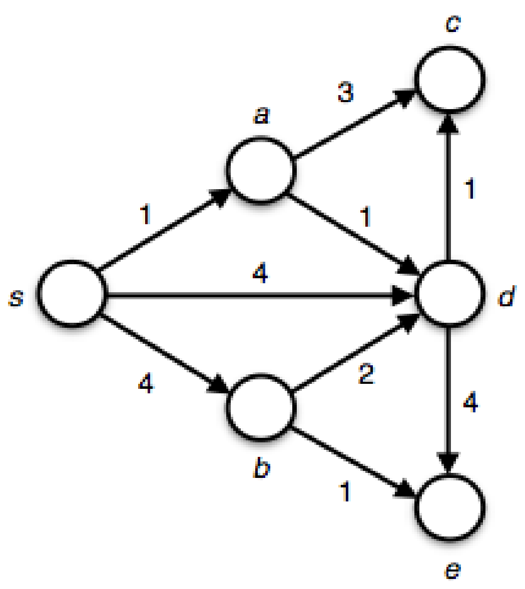

s
a
b
c
d
e
0
∞
∞
∞
∞
∞
1
4
4
3
1
2
1
1
4


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


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


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


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


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

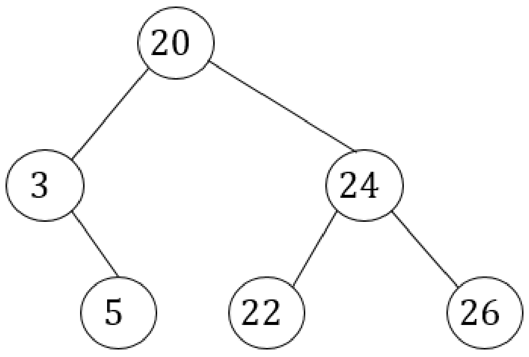
7. (21 pts) Two binary search trees T_1 and T_2 are considered to be ***equal*** if they are identical in structure and content. Recursively, T_1 and T_2 are equal if

1) they are both empty or
2) they are both non-empty and the following three conditions are all satisfied:
    a. the keys stored at their roots are equal;
    b. their left subtrees are equal;
    c. their right subtrees are equal.

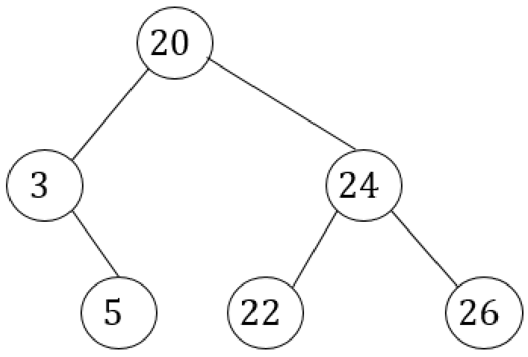
The trees T_1 and T_2 are ***set equal*** if they store equal sets of keys (essentially, the same key set).

As an example, for the four trees below, T_1 and T_2 are equal (and thus set equal); T_1 and T_3 are set equal but not equal; and T_1 and T_4 are neither equal nor set equal.

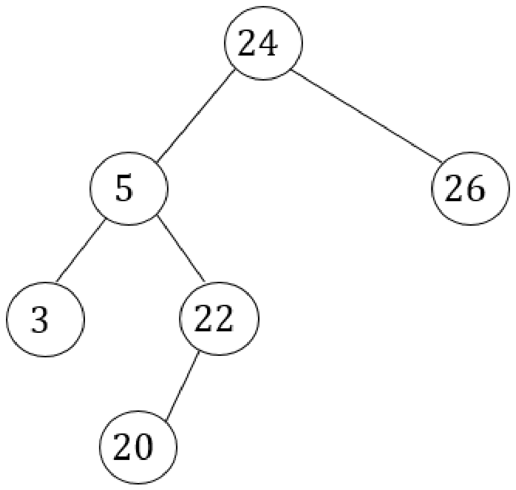
T_1:



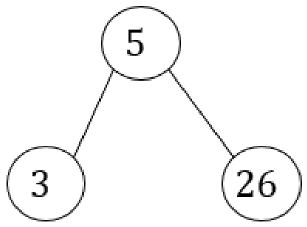
T_2:



T_3:



T_4:



In this problem, you are asked to implement two methods `treeEqual()` and `setEqual()` for the binary search tree (**BST**) class below. The method `treeEqual()` determines if **this** tree is ***equal*** to a second tree. It employs a method `subtreeEqual()` which takes two nodes as parameters and determines whether the subtrees rooted at them are equal. The method `setEqual()` determines if **this** tree is ***set equal*** to a second tree.

The instance variable `data` of the private **Node** class represents the key value. The `data` items stored in the tree are ***always distinct***.

The class **BST** has a fully implemented iterator class `BSTIterator` plus a method `iterator()` to create an object of this class.

```
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;          // key value

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


  // other methods
  // ...

  @Override
  public Iterator<E> iterator()
  {
    return new BSTIterator();
  }

  /**
   * Iterator implementation for this binary search tree.  The elements
   * are returned in ascending order according to their natural ordering.
   */
  private class BSTIterator implements Iterator<E>
  {
    // ...
    //@Override
    public boolean hasNext()
    {
      // ...
    }

    //@Override
    public E next()
    {
       // ...
    }

    // ...
  }


  /**
   * Determine if the tree is equal to another tree (rooted at tree2),
   * that is, if the two trees are identical in structure and content.
   *
   * Precondition: tree2 != null
   *
   * @param   tree2  tree to be compared with this tree
   * @return  true if equal and false otherwise
   */
  public boolean treeEqual(BST<E> tree2)
  {
        // handle the situation where the two trees have different sizes
        // insert code below (2 pts)






        // the two trees have the same size
        // insert code below (2 pts)








  }


  /**
   * Recursively determine if two subtrees are equal.
   *
```

```java
 * @param node1  root of the first subtree
 * @param node2  root of the second subtree
 * @return true if equal and false otherwise
 */
private boolean subtreeEqual(Node node1, Node node2)
{
      // handle the situation(s) where one or both of the nodes are null.
      // insert code below (3 pts)









      // neither node is null
      // insert code below (4 pts)








}


/**
 * Determine if the tree and another tree store the same set of keys.
 *
 * Precondition: tree2 != null
 *
 * @param  tree2  tree to be compared with this tree
 * @return  true  if set equal and false otherwise
 */
public boolean setEqual(BST<E> tree2)
```

```
  {
        // handle the situation where the two trees have different sizes
        // insert code below (1 pt)







        // the trees are of identical size. initialize two iterators
        // iter1 and iter2.
        // insert code below (2 pts)







        // compare whether the two sets of keys are equal.
        // insert code below (7 pts)






  }
}
```

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)