Com S 228
Spring 2018
Final Exam Sample Solution

1 a) 8

b) 9

c) 16

d) 3

e) 4

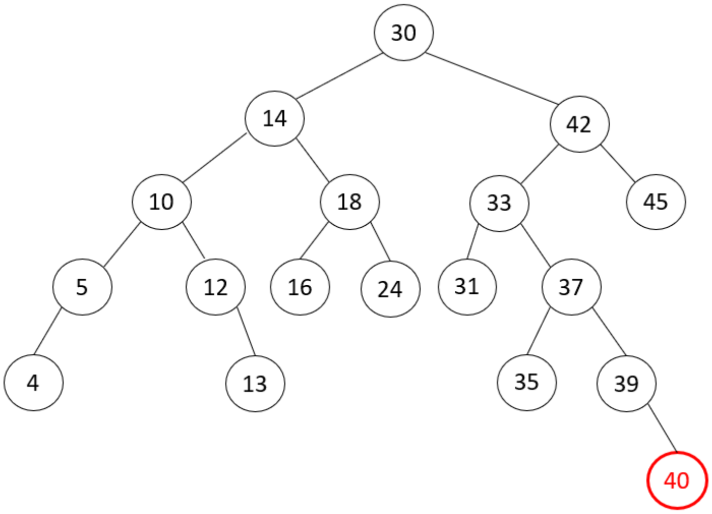
f) 3

g) 30

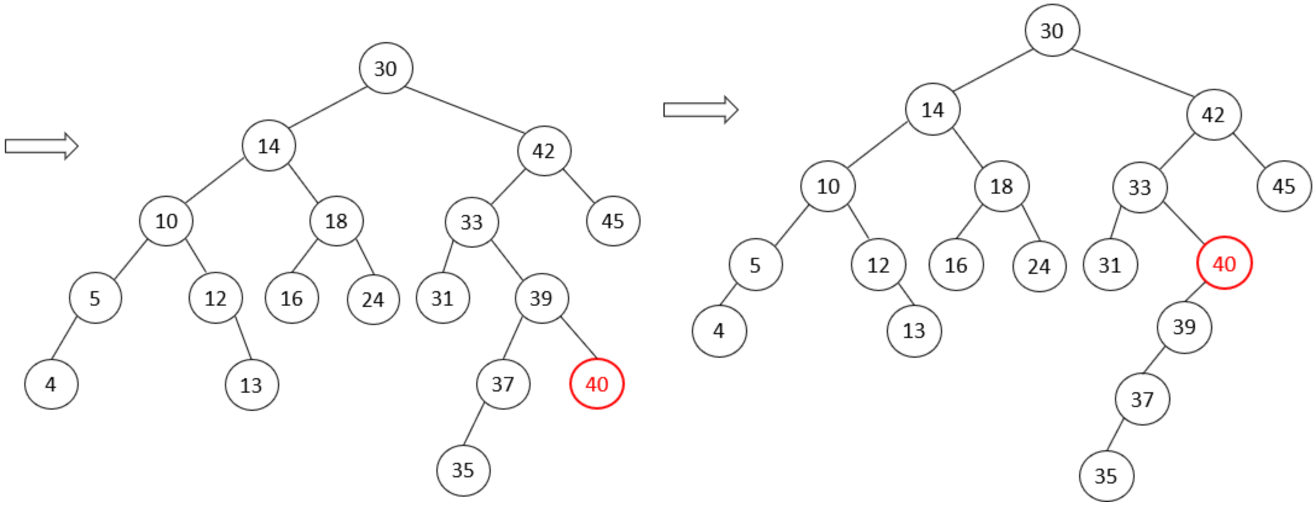
h) 13

i) 4, 5, 13, 12, 10, 16, 24, 18, 14, 31, 35, 39, 37, 33, 45, 42, 30.

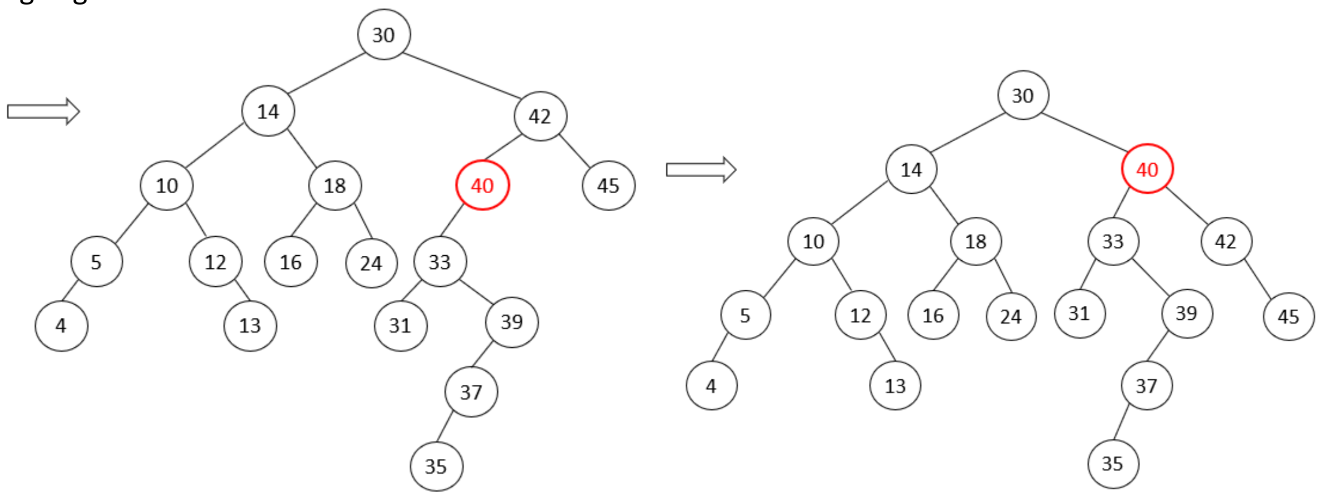
j) BST insert:



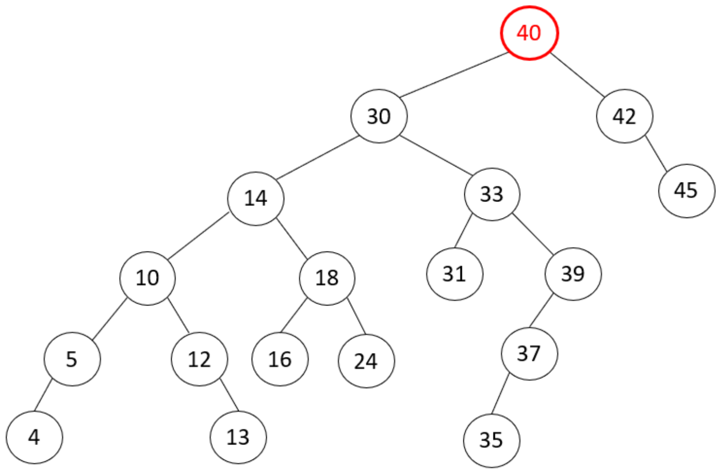
Zig-zig:



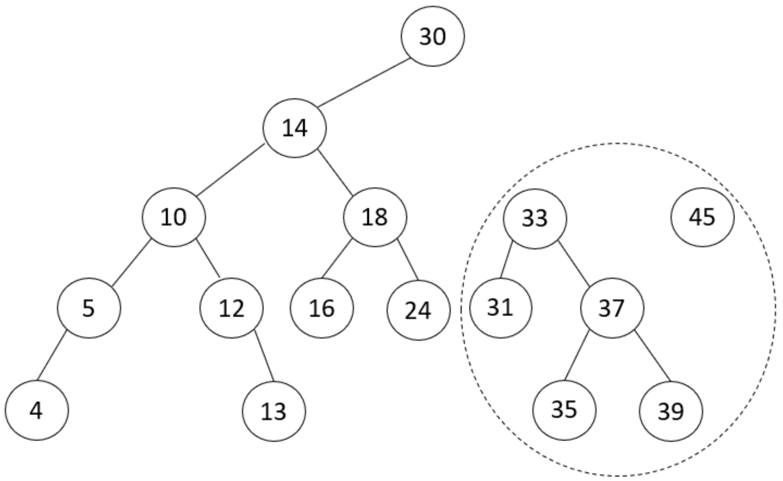
Zig-zag:



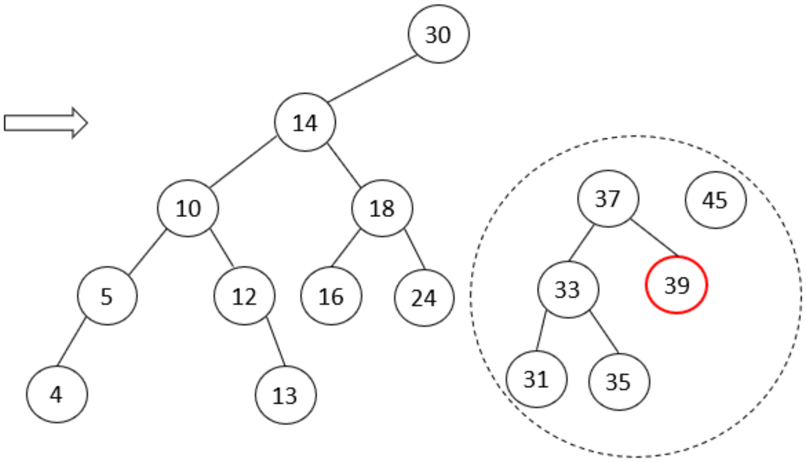
Zig:

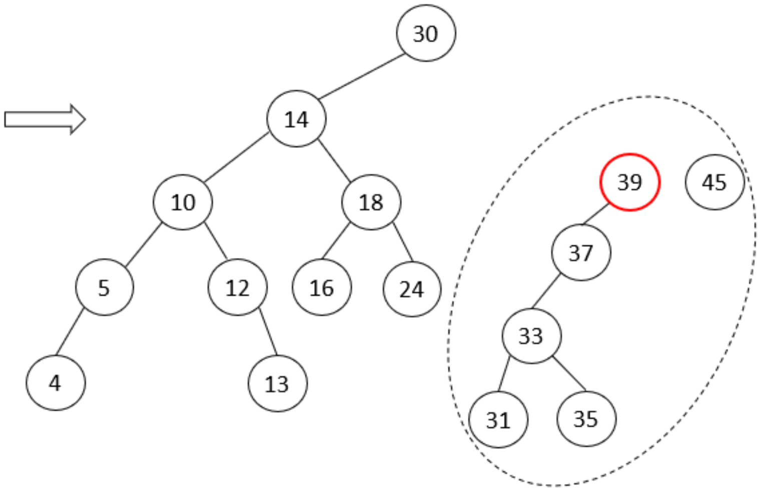


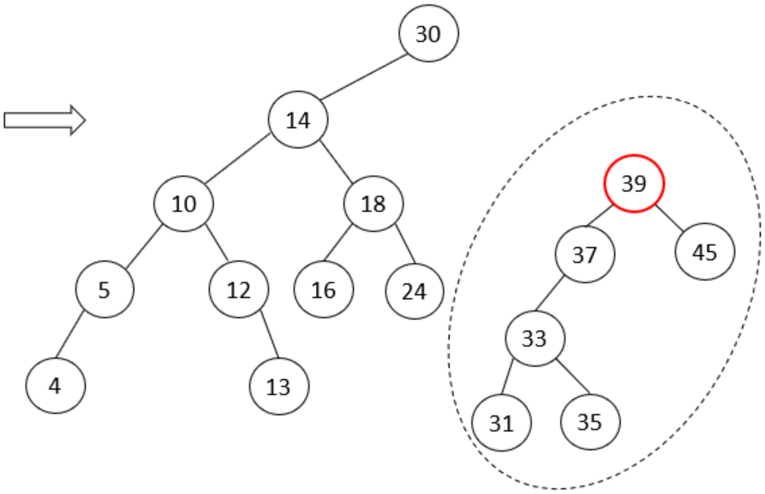
k) Remove 42:



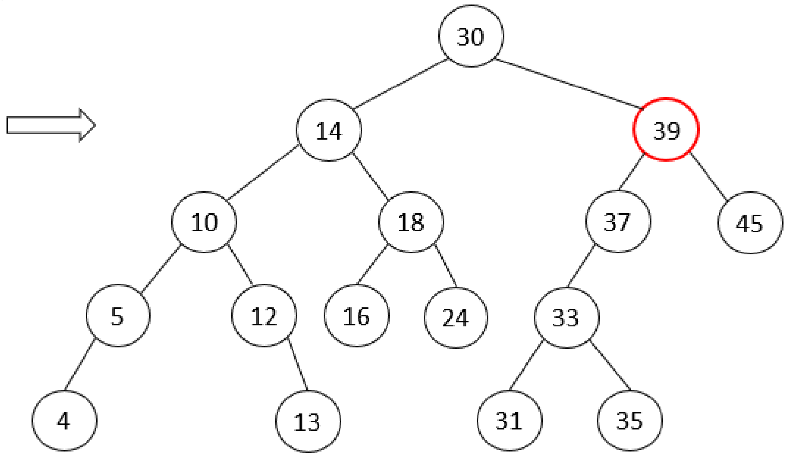
Join the subtrees of 42, starting with accessing the rightmost node in the left subtree:





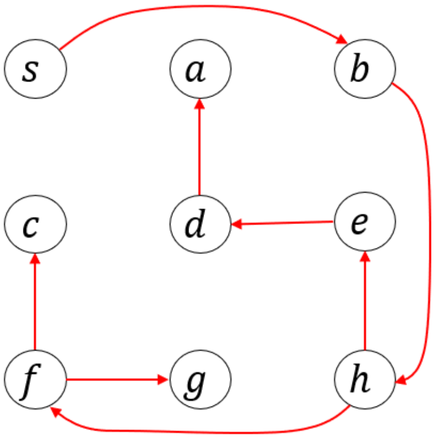


Replace 42 with the join.

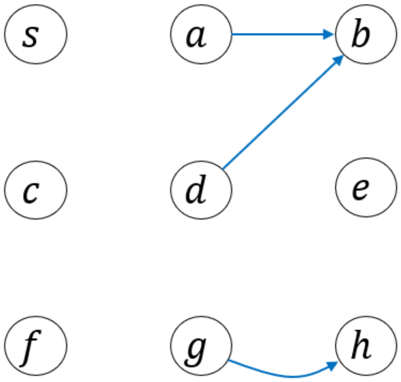


No need to splay at the parent node 30 of 39 since 30 is the root.

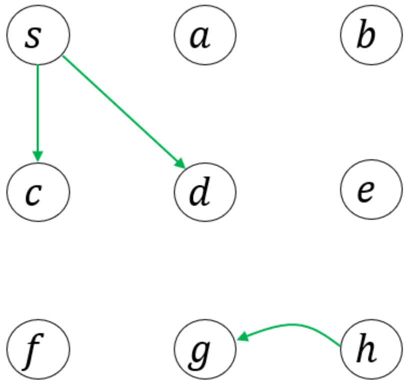
2a) 5
b) 3
c) No
d) DFS forest



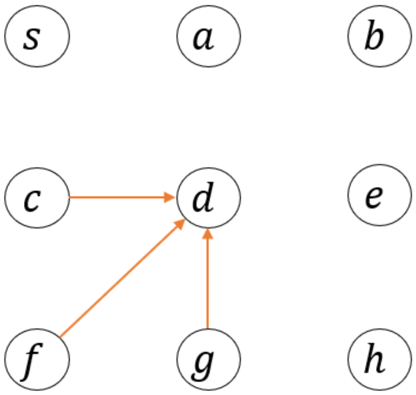
e) Back edges



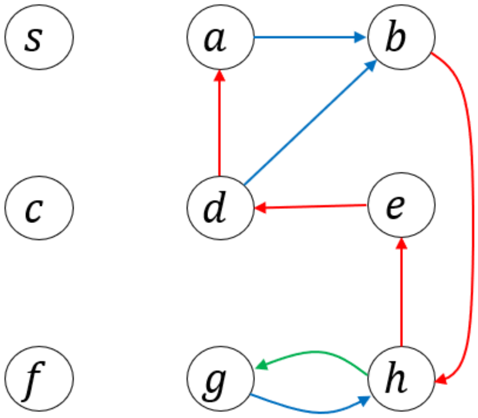
f) Forward edges



g) Cross edges



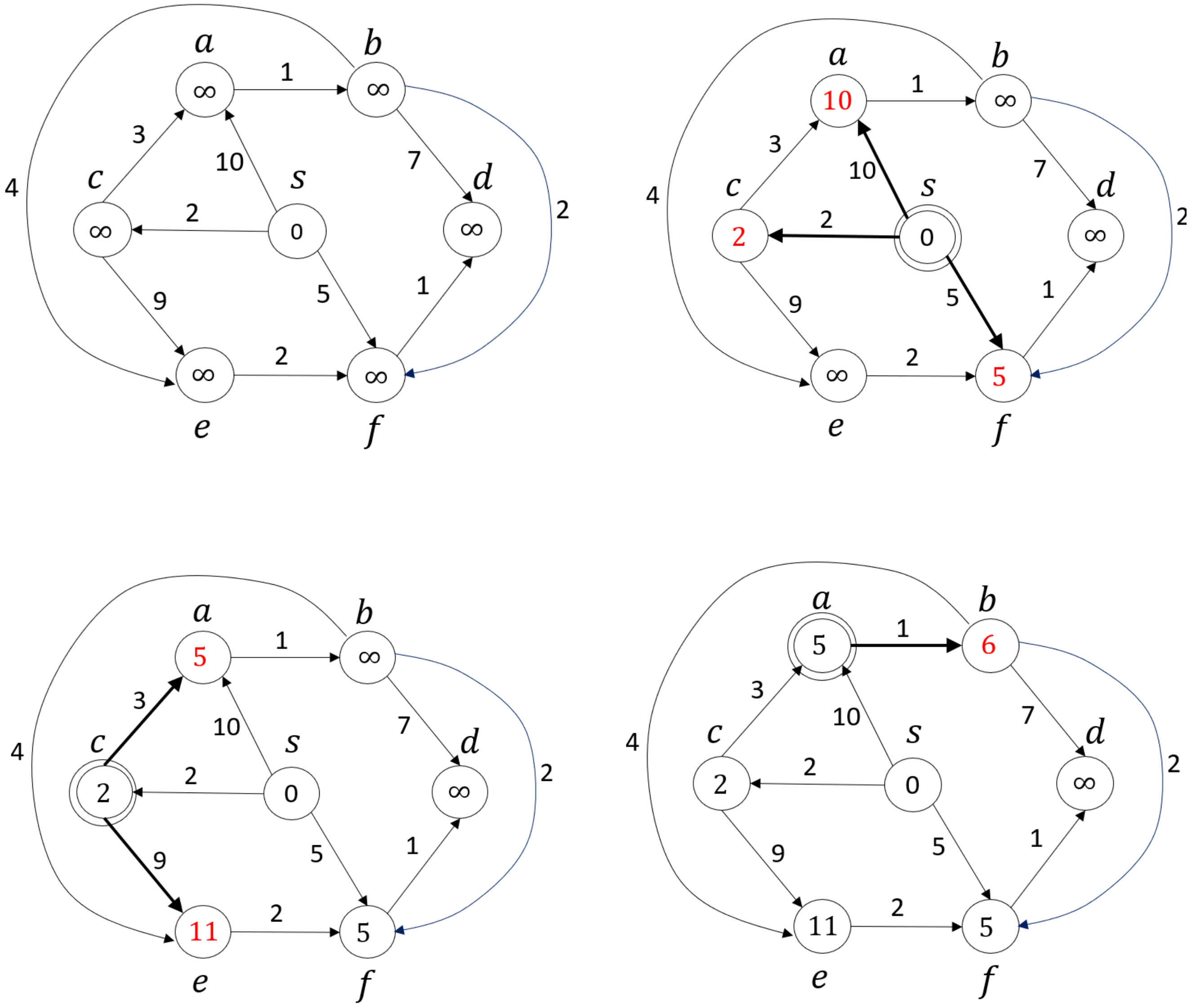
h) Three of the cycles.

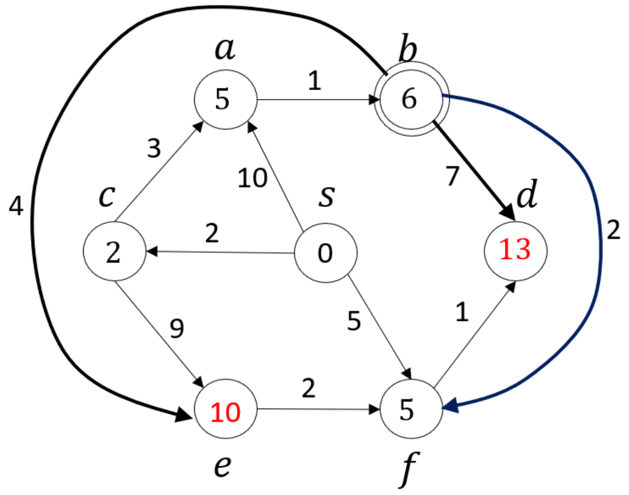


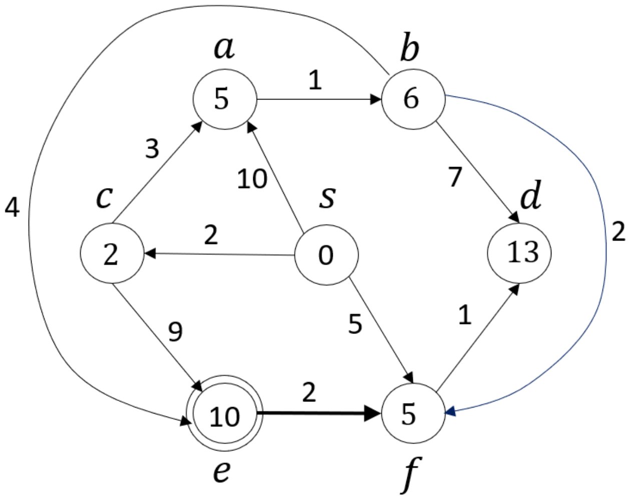
3a) There exists a unique sorting result.

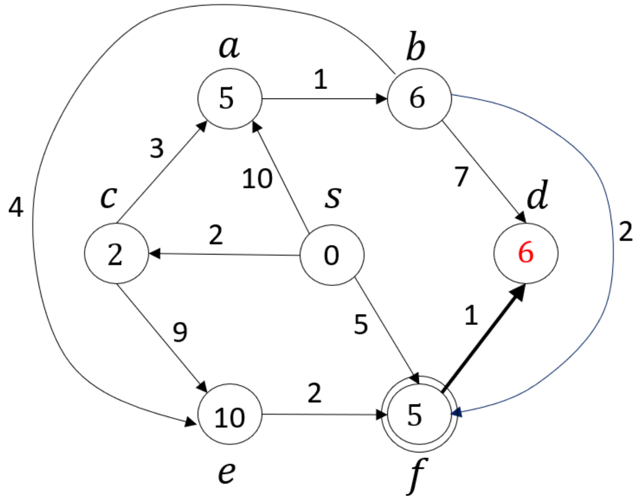
s	c	a	b	e	f	d

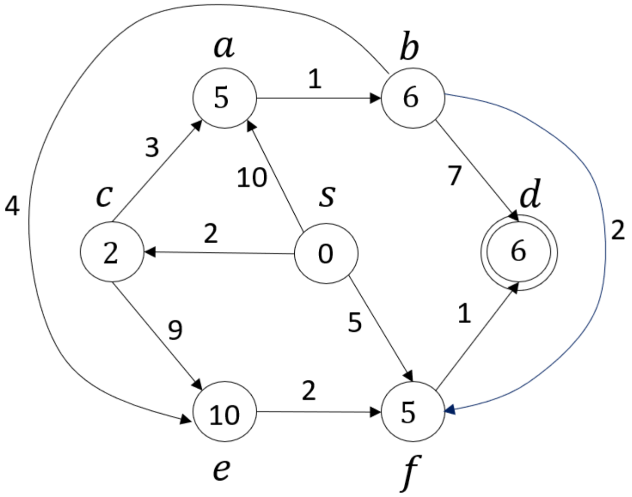
b)











4. Row 3 with its entries underlined is the first line that is a heap.

Row				Array			
0	1	3	6	5	2	0	4
1	1	5	6	3	2	0	4
2	6	5	1	3	2	0	4
3	6	5	4	3	2	0	1
4	1	5	4	3	2	0	6
5	5	1	4	3	2	0	6
6	5	3	4	1	2	0	6

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

5a) $O(n)$
b) $O(\log n)$
c) $O(\log n)$
d) $O(1)$
e) $O(n)$
f) $O(n)$
g) $O(V+E)$
h) $O(\log n)$ (amortized time) or $O(n)$ (worst-case time)
i) $O(V^2)$
j) $O(1)$

6a)

```
/**
 * 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)
     stk.push(n.data);
     if (n.left != null)
          preorderOntoStackRec(n.left);
     if (n.right != null)
          preorderOntoStackRec(n.right);
}
```

b)

```
/**
 * 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.
     // insert code below (1 pt)
     BSTSet<E> newTree = new BSTSet<E>();

     // if this tree object is empty, then return newTree right away.
     // insert code below (1 pt)
     if (root == null)   // or size == 0 or stk.isEmpty()
          return newTree;

     // call preorderOntoStack().
     // insert code below (1 pt)
     preorderOntoStack();

     // 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)
     newTree.root = new Node(stk.pop(), null);
     newTree.size = 1;
```

```
// 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 = stk.pop();

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

    // initialize the current node.
    // complete code below (1 pt)
    Node current = newTree.root;

    while (current != null) {

        // compare key and the data stored at the current
        // node.
        // complete code below (2 pts)
        if (key.compareTo(current.data) < 0) {
            // check beforehand if current has no child
            // down the direction corresponding to the
            // above condition.
            // complete code below (1 pt)
            if (current.left == null) {
                // perform a needed tree update.
                // afterward, set current to null to exit
                // the inner while loop.
                // insert code below (2 pts)
                current.left = new Node(key, current);
                current = null;
            } else {
                // update current
                // insert code below (1 pt)
                current = current.left;
            }
        } else {
            // check if current has no child down the other
            // direction.
            // complete code below (1 pt)
            if (current.right == null) {
                // perform a needed tree update.
                // afterward, set current to null.
                // insert code below (1 pt)
                current.right = new Node(key, current);
                current = null;
            } else {
```

```
                        // update current
                        // insert code below (1 pt)
                        current = current.right;
                    }
                }
            }

            // update size after insertion.
            // insert code below (1 pt)
            ++newTree.size;
        }

        // return the newly created tree.
        // insert code below (1 pt)
        return newTree;
    }
```