Problem Set 7, Part I

Problem 1: Working with stacks and queues

1)
```
public static void remAllStack(Stack<Object> stack, Object item) {
        LLStack temp = new LLStack();
        while(stack.isEmpty() == false){
                Object itemAt == (Object)stack.pop();
                if(itemAt.equals(item) != true) {
                        temp.push(itemAt);
                }
        }
        while(temp.isEmpty() == false) {
                Object itemAt == (Object)temp.pop();
                stack.push(itemAt);
        }
}
```

2)
```
public static void remAllQueue(Queue<Object> queue, Object item) {
        LLstack.temp = new LLStack();
        while(queue.isEmpty() != true) {
                Obejct itemAt = (Object)queue.remove();
                if(itemAt.queue(item) {
                        temp.push(itemAt);
                }
        }
        LLstack.temp2 = new LLStack();
        while(temp.isEmpty() != true){
                Object itemAt = (Object)temp.pop();
                temp2.push(itemAt);
        }
        while(temp2.isEmpty() != true) {
                Object itemAt = (Object)temp2.pop();
                queue.insert(itemAt);
        }
}
```

Problem 2: Using queues to implement a stack

push(x): Enqueue x to Q2, Dequeue every variable from Q2 to Q1, enqueue them to Q1 and change the name of Q1 to Q2, Q2 to Q1. The run time is O(n)

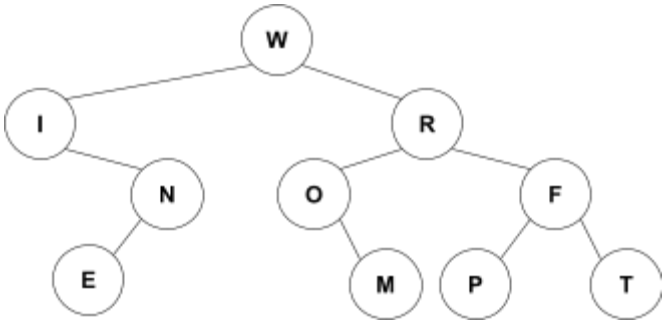
pop(): when the size of Q1 is bigger than 1, insert removed item from Q1 to Q2 remove and return the last item of Q1 and switch the name of Q1 to Q2. The run time is O(1)

peek(): IF Q1 is empty, print “empty stack” and else print the first element of Q1. The run time is O(1)

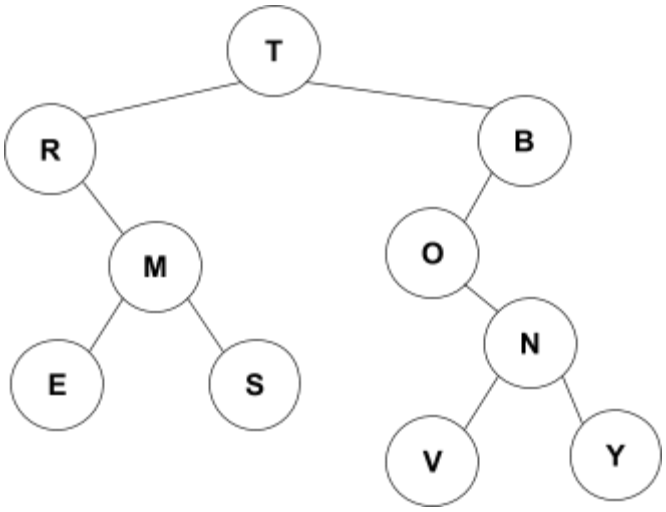
Problem 3: Binary tree basics

1) Height of 3
2) 4 leaf nodes, 4 interior nodes
3) 21 18 7 25 19 27 30 26 35
4) 7 19 25 18 26 35 30 27 21
5) 21 18 27 7 25 30 19 26 35
6) Yes, because all of the left subtree’s key is less than the root and right subtree’s key.
7) Yes, because the definition of balanced is that the left subtree and right subtree’s level difference is at most one level.

Problem 4: Tree traversal puzzles
4-1)

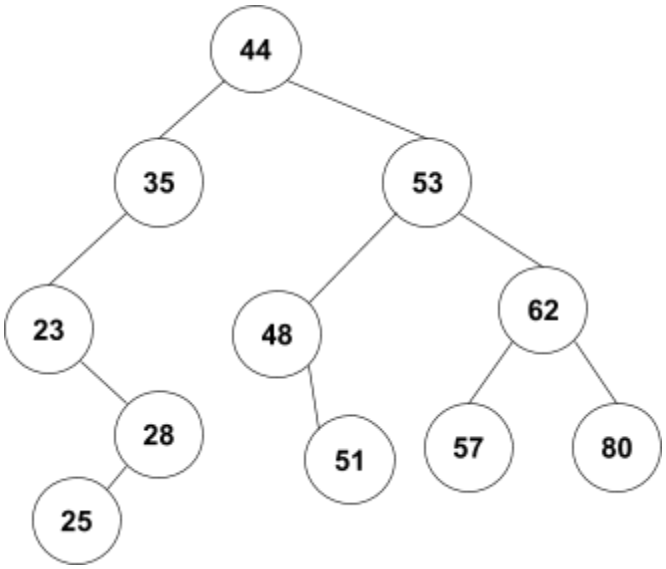


4-2)

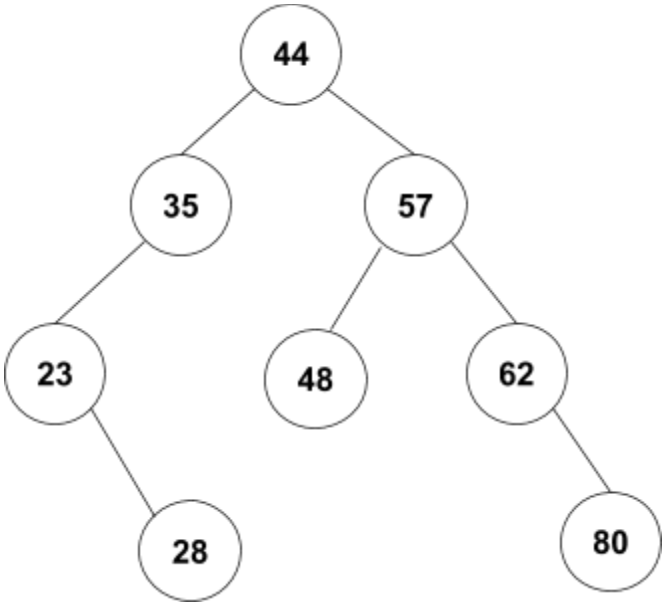


Problem 5: Binary search trees

5-1)



5-2)



Problem 6: Determining the depth of a node

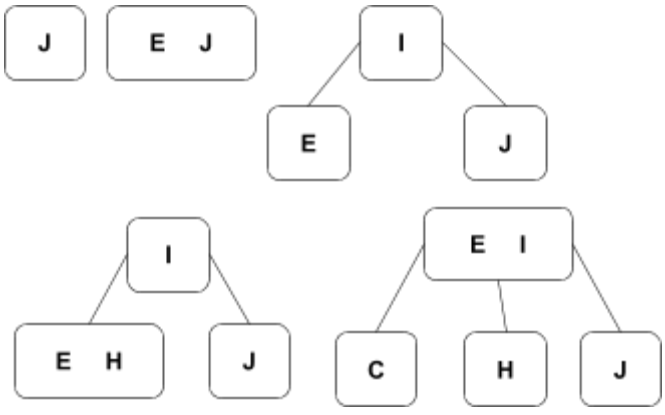
1) The best-case: O(1) → when the depth of the finding key is the root for the binary tree, which is 0.
   The worst-case: O(n) → since the algorithm looks at every node of the binary tree to search for the key, no matter of the tree being balanced or unbalanced, the big O will be O(n)

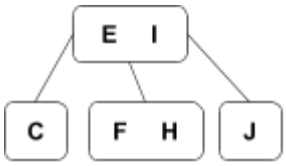
2)
```
private static int depthInTree(int key, Node root) {
        if(key == root.key) {
                return 0;
        }
        if(root.left != null && key < root.key) {
                Int depthInLeft = depthInTree(key.root.left);
                if(depthInLeft != -1) {
                        return depthInLeft + 1;
                }
        } else if(root.right != null && key > root.key) {
                Int depthInRight = depthInTree(key.root.right);
                if(depthInRight != -1) {
                        return depthInRight + 1;
                }
        }

        return -1;
}
```

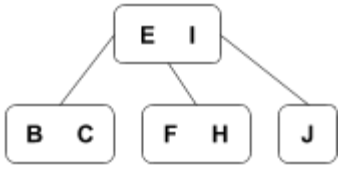
3) The best-case: O(1) → when the depth of the finding key is the root for the binary tree, which is 0.
   The worst-case: When it is balanced O(log n) → since the tree is balanced, the height would be log n.
   When it is not balanced, the big O will be O(n) → the height will be O(n), making the big O notation, O(n).

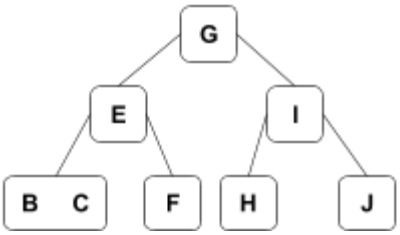
Problem 7: 2-3 Trees and B-trees

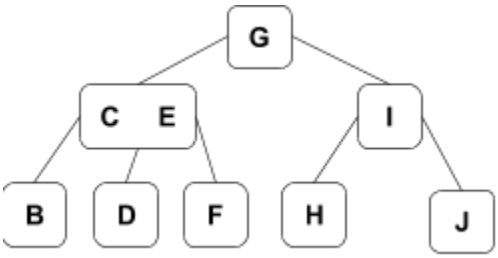
7-1)

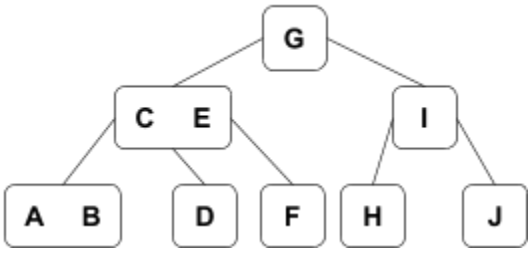



E I
C
F H
J

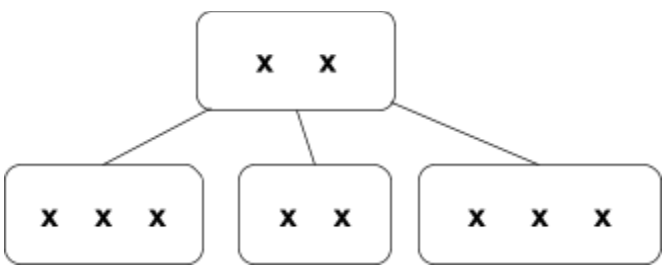

E I
B C
F H
J


G
E
I
B C
F
H
J


G
C E
I
B
D
F
H
J


G
C E
I
A B
D
F
H
J

7-2)


X X
X X X
X X
X X X