Problem Set 7, Part I

Problem 1: Working with stacks and queues

```java
public static void remAllStack(Stack<Object> stack, Object item) {
    LLStack reminder = new LLStack();
    while (!stack.isEmpty()) {
        Object currentItem = stack.pop();
        if (!currentItem.equals(item)) {
            reminder.push(currentItem);
        }
    }
    while (!reminder.isEmpty()) {
        Object addMe = reminder.pop();
        stack.push(addMe);
    }
    System.out.println("My stack: " + stack.toString());
}
```

Problem 2: Using queues to implement a stack

```
// Q1 is used to store items, and Q2 is the temp helping process items.
```

```
public boolean push(T item) {
    Q2.insert(item);
    while (!Q1.isEmpty()) {
        Q2.insert(Q1.remove());
    }
    Q1 = Q2;
    return true;
}
```

For this push method, O(n) because we need to go through the whole Q1 to copy its items to Q2.

```
public T peek() {
    return Q1.peek();
}
```

For this peek method, O(1) because no iteration is used and queue's peek() method is O(1).

```
public T pop() {
    return Q1.remove();
}
```

For this pop method, O(1) because no iteration is used and queue's remove() method is O(1).

```
public boolean isEmpty() {
    return Q1.isEmpty();
}
```

For this isEmpty method, O(1) because no iteration is used and queue's isEmpty () method is O(1).

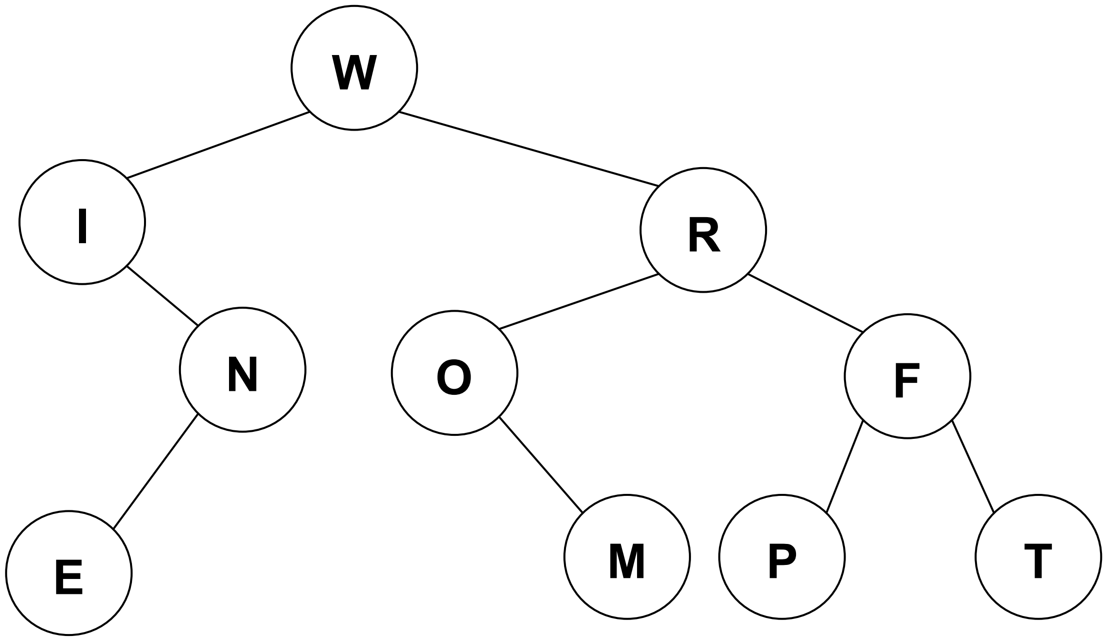
```
public boolean isFull() {
    return Q1.isFull();
}
```

For this isFull method, O(1) because no iteration is used and queue's isFull () method is O(1).

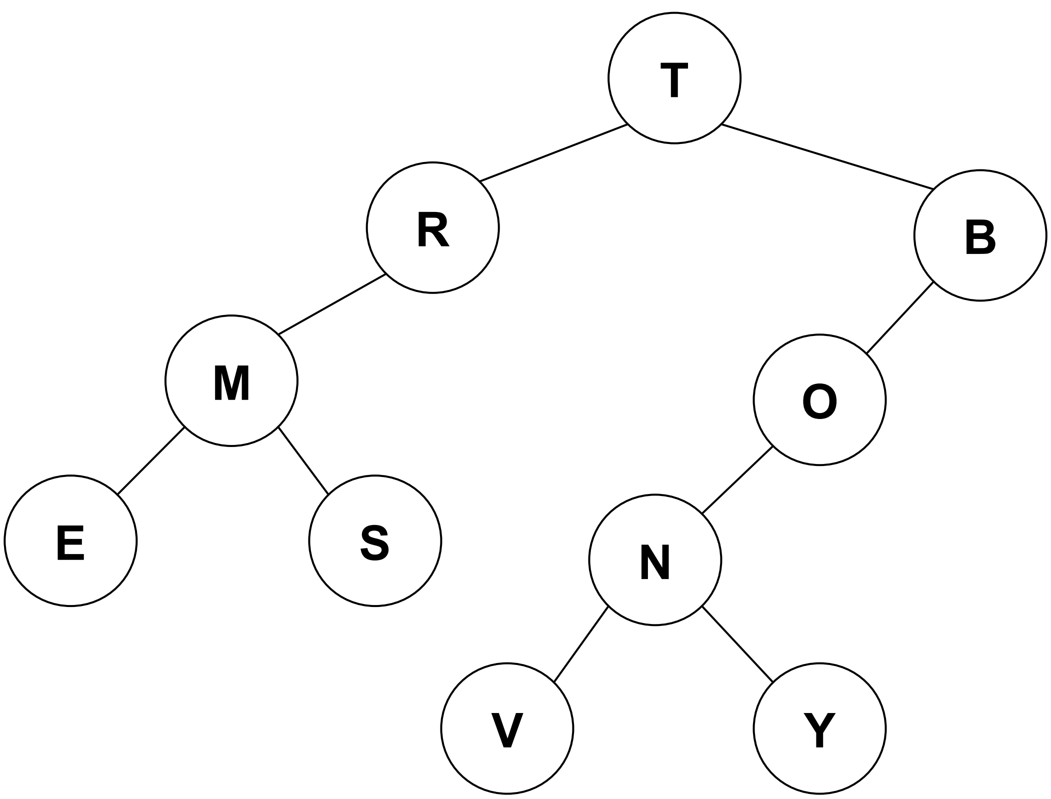
Problem 3: Binary tree basics

1. Height = 3
2. Leaf node = 4, interior node = 5
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. NO, 25 > 21, but it is at the left side
7. No, the height of 27’s left is -1 while the height of 27’s right is 1. Their difference is 2, greater than 1.

Problem 4: Tree traversal puzzles
4-1)

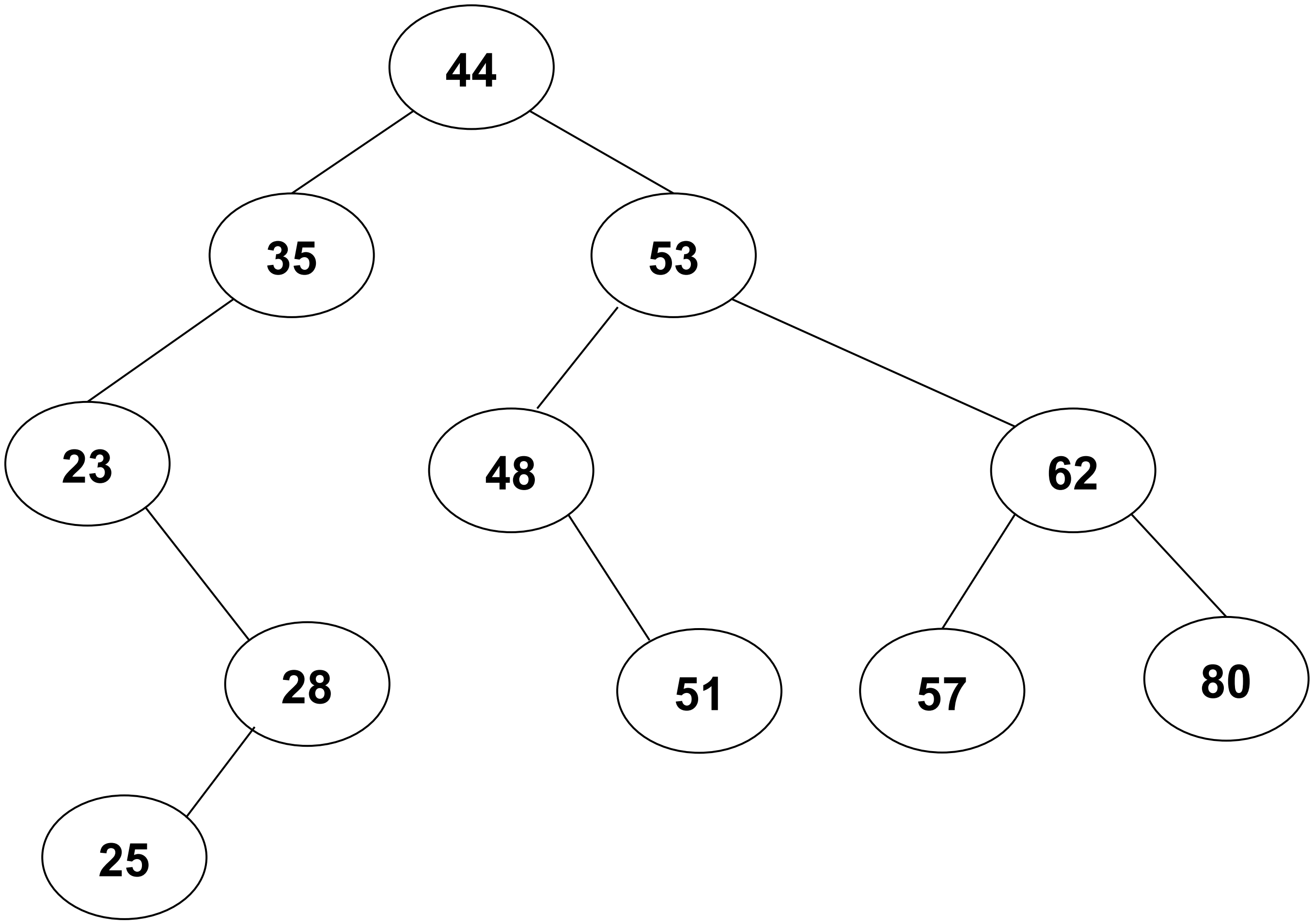


4-2)

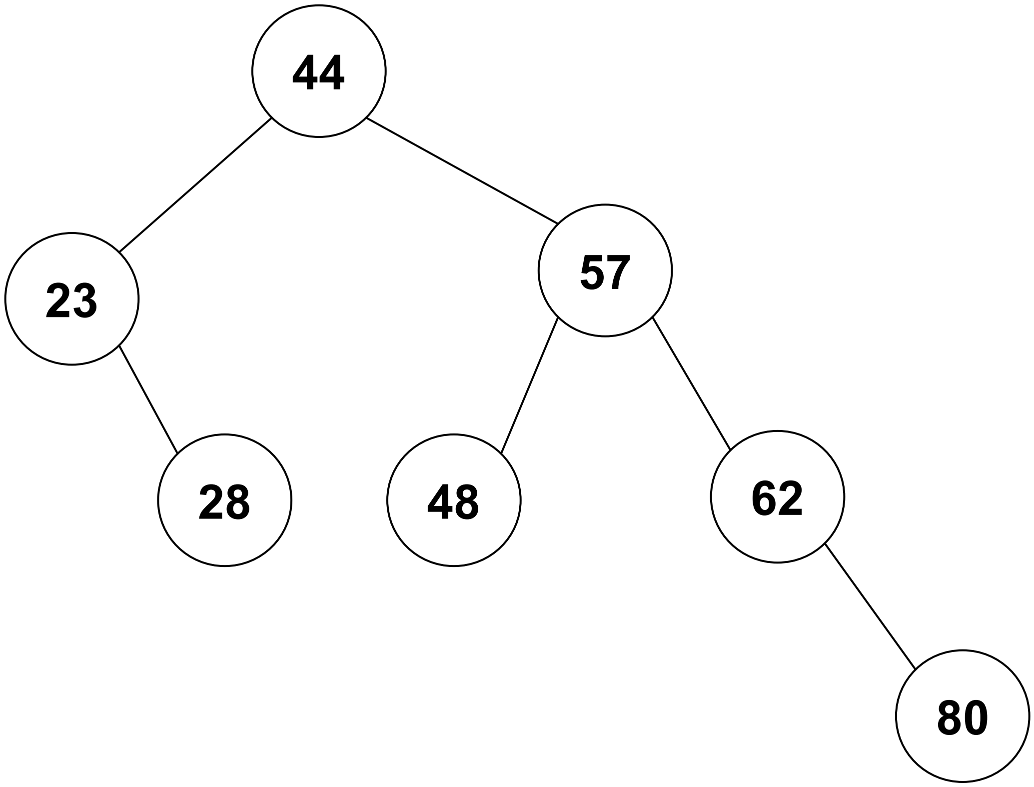


Problem 5: Binary search trees

5-1)



5-2)



Problem 6: Determining the depth of a node

1).

- **Best case:** O(1) when the key we need to find is exactly the root.
- **Worst case:** O(n) when the key we need to find is at the deepest and most right leaf.
    - **Balanced:** O(n) even though it is balanced, we have to go over every single element (whatever in left or right subtrees) in the whole tree to find the key.
    - **Not balanced:** O(n) when the tree is equivalent to a linked list (height == n-1).

2).

```java
private static int depthInTree(int key, Node root) {
    if (key == root.key) {
        return 0;
    }

    if (key < root.key) {
        if (root.left != null) {
            int depthInLeft = depthInTree(key, root.left);
            if (depthInLeft != -1) {
                return depthInLeft + 1;
            }
        }
    } else {
        if (root.right != null) {
            int depthInRight = depthInTree(key, root.right);
            if (depthInRight != -1) {
                return depthInRight + 1;
            }
        }
    }
    return -1;
}
```

3).

- **Best case:** O(1) when the key we need to find is exactly the root.
- **Worst case:** O(h) when the key we need to find is at the deepest leaf.
    - **Balanced:** O(log n) because every time following an edge down the longest path, cutting the problem size roughly in half!
    - **Not balanced:** O(n) when the tree is equivalent to a linked list (height == n-1).

Problem 7: 2-3 Trees and B-trees

7-1)