Problem Set 7, Part I

Problem 1: Choosing an appropriate representation

1-1) *ArrayList or LLList?* ArrayList

Explanation: As the number of events is roughly the same, we can predict the maximum length. So, the space efficiency of ArrayList always depends on the anticipated maximum length. It is also more time-efficient to use ArrayList because the events are added in order. The ArrayList does not need to shift, so it's the best case that only needs O(1) to add an event.

1-2) *ArrayList or LLList?* ArrayList

Explanation: As we have to frequently access the runners' records through the id number as index, we need to use ArrayList which provides random access with time efficiency of O(1). If we use LLList, we need to traverse the list to find the item, which is not time-efficient. Also, the length of the list is almost constant. It means that we can predict the maximum length of the ArrayList, and the space efficiency depends on the anticipated maximum length. There are very few changes to the list after the deadline, so the ArrayList does not need to spend much time on shifting.

1-3) *ArrayList or LLList?* LLList

Explanation: The number of registrants varies significantly, so it is more space-efficient to use LLList which space efficiency depends on the number of added items in the list. ArrayList is not space-efficient because we have to preallocate the memory needed for the maximum sequence size. For time efficiency, we always add the recent one to the front, so it is more time-efficient to use LLList to add to the front of the list, which only needs O(1).

Problem 2: Scaling a list of integers

2-1)

The for loop will execute for n times. For each for loop, the time efficiency of getItem() is O(1), because ArrayList provides random access. And the time efficiency of addItem depends on the length n, because it needs to traverse the full list to add the item. Therefore, the running time of the algorithm is n*1 + (1 + 2 + … + n) = O(n^2).

2-2)

```
public static LLList scale(int factor, ArrayList vals) {
    LLList scaled = new LLList();

    for (int i = vals.length()-1; i >= 0; i--) {
        int val = (Integer)vals.getItem(i);
        scaled.addItem(val*factor, 0);
    }
    return scaled;
}
```

}

2-3)
The for loop will execute for n times. For each for loop, as we always add the item to the front of the LLList, the time efficiency of addItem() is O(1) because there is no need to walk down. Also, the time efficiency of getItem() is O(1) because ArrayList provides random access. Therefore, the running time of the algorithm is O(n).

Problem 3: Working with stacks and queues
3-1)
```
public static void remAllStack(Stack<Object> stack, Object item) {
    Stack<Object> stack2 = new LLStack<Object>();
    while (!stack.isEmpty()) {
        if (stack.peek().equals(item)) {
            stack.pop();
        } else {
            Object O1 = stack.pop();
            stack2.push(O1);
        }
    }
    while (!stack2.isEmpty()) {
        Object O2 = stack2.pop();
        stack.push(O2);
    }
}
```

3-2)
```
public static void remAllQueue(Queue<Object> queue, Object item) {
    Queue<Object> queue2 = new LLQueue<Object>();
    while (!queue.isEmpty()) {
        if (queue.peek().equals(item)) {
            queue.remove();
        } else {
            Object O1 = queue.remove();
            queue2.insert(O1);
        }
    }
    while (!queue2.isEmpty()) {
        Object O2 = queue2.remove();
        queue.insert(O2);
    }
}
```

Problem 4: Binary tree basics

4-1)
The height is 3.

4-2)
It has 4 leaf nodes and 5 interior nodes.

4-3)
21 18 7 25 19 27 30 26 35

4-4)
7 19 25 18 26 35 30 27 21

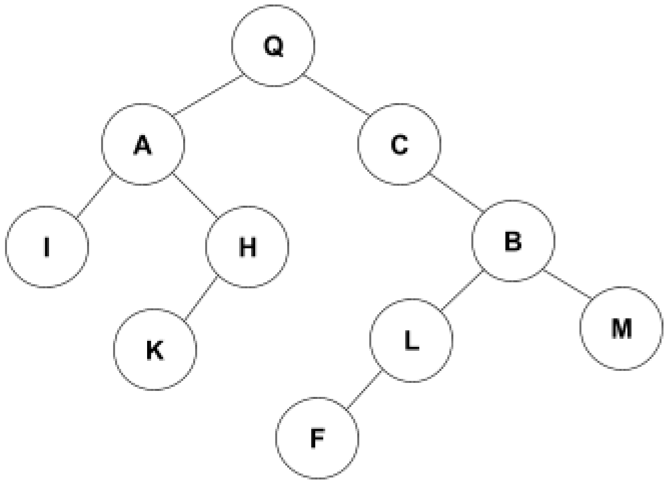
4-5)
21 18 27 7 25 30 19 26 35

4-6)
No. For Node 25, 25 is greater than 21, so it should be on the right side of Node 21. For Node 26, 26 is smaller than 27, so it should be on the left side of Node 27.

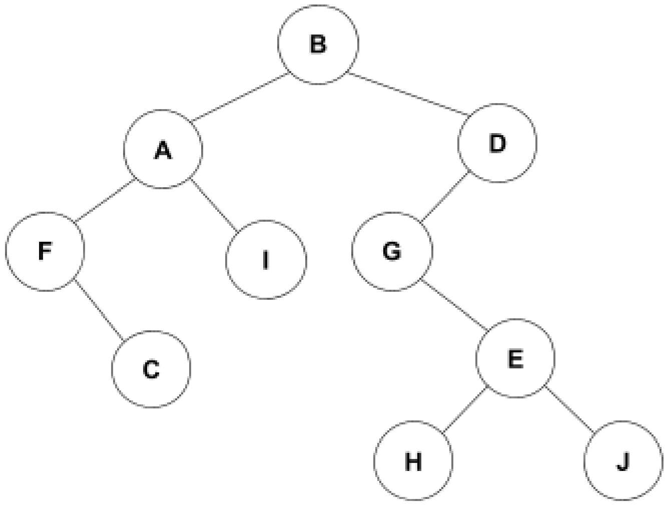
4-7)
Yes. The tree is balanced because the node’s subtrees have the same height or have heights that differ by 1.

Problem 5: Tree traversal puzzles

5-1)

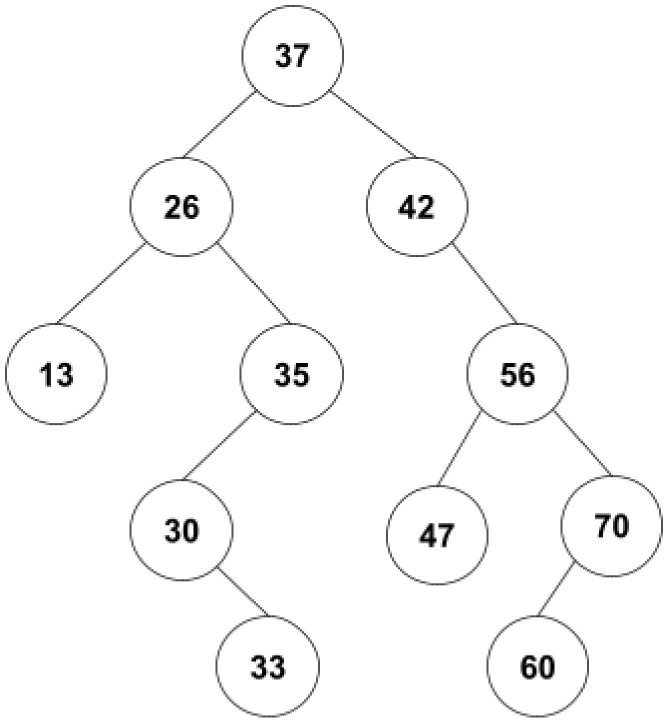


5-2)

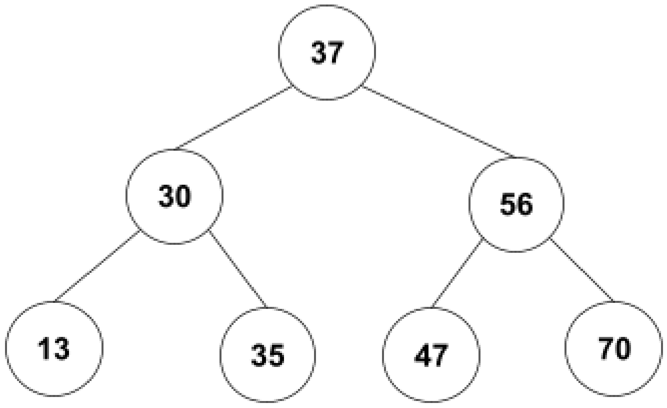


Problem 6: Binary search trees

6-1)



6-2)


37
30
56
13
35
47
70