Problem Set 7, Part I

Problem 1: Choosing an appropriate representation

1-1) *ArrayList or LLList?* **ArrayList**

explanation:

The events will be added and displayed in order, so it's better to add items and print a list from the beginning to the end.

LLList is more efficient when added to the front since it doesn't need to go down to the specific position in the middle. However, we need to print the list from the end to the beginning, which is inefficient.

ArrayList is more efficient when added to the last since it doesn't need to shift items. We can also easily print an ArrayList from the beginning to the end.

Furthermore, the number of monthly events is roughly the same, so the maximum space in an ArrayList is specified, and there will not be spaces in waste. Therefore, ArrayList is a more promising and efficient option to add items in order.

1-2) *ArrayList or LLList?* **ArrayList**

explanation:

The range of indexes is fixed (i.e., between 1 and 3000). We also need to frequently access the information of specific runners to add the times of passing various mile markers. So, the time efficiency of accessing an item (i.e., getItem()) is crucial.

The average-case time complexity of LLList's getItem() is O(n), whereas that of ArrayList is O(1). To access a particular node in an LLList, we have to traverse the list till that node, but we can access any item in an ArrayList always with O(1) step.

On the other hand, there are at most 3000 runners, and there will be few changes to the list after the sign-up. It's easy to specify the maximum space of the ArrayList. Therefore, ArrayList is a more promising and efficient option.

1-3) *ArrayList or LLList?* **LLList**

explanation:

The organizer has no idea how many students for each hackathon and wants to display the registrant list from the most recently added one to the least recently added one.

It's not highly inefficient to print from the end of an ArrayList. However, the uncertainty of how many registrants there might be will make it difficult to specify the maximum space and allocate memory for an ArrayList. We might run into issues like 1) the ArrayList is running out of space when there are more students than expected; 2) most spaces of the ArrayList will be wasted when there are fewer students than expected.

Since it's efficient to add items at the front of an LLList, the most recently added item is at the beginning, and the least recently added one is at the last. It's easy to print from the most recently added one to the least recently added one by traversing the LLList in order. Plus, since an LLList is never full, it's flexible to handle the uncertainty of the number of registrants. Therefore, LLList is a more promising and efficient option.

Problem 2: Scaling a list of integers

2-1) The most inefficient portion of this algorithm is the for loop. If the original list has a length of n, there will be n repetitions in total. The time complexity of ArrayList's getItem() is always O(1). The time complexity of LLList's addItem() is O(n), since, for every repetition, addItem() needs to go from the beginning to the specified position to add an item. Therefore, the running time of this algorithm is n * (1 + n) = n + n^2. We take the fastest growing term and derive the big-O notation of O(n^2).

2-2)

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

       for (int i = vals.length()-1; i >= 0; i--) {
           int value = (Integer)vals.getItem(i);  // O(1)
           newList.addItem(value*factor, 0);
       }

       return newList;
   }
```

2-3) The most inefficient portion of this algorithm is the for loop. If the ArrayList has a length of n, there will be n repetitions in total. The time complexity of ArrayList's getItem() is always O(1). For every repetition, a new item is added to the front of the LLList, which only costs O(1) to do so. Therefore, the running time of the new algorithm is O(n).

Problem 3: Working with stacks and queues

3-1)

```
public static void remAllStack(Stack<Object> stack, Object item) {
	Stack<Object> newStack = new LLStack<Object>();
	while (! stack.isEmpty()) {
	    Object top = stack.peek();
	    if (! item.equals(top)) {
	        newStack.push(top);
	    }
	    stack.pop();
	}
	while (! newStack.isEmpty()) {
	    Object newTop = newStack.peek();
	    stack.push(newTop);
	    newStack.pop();
	}

  }
```

3-2)

```
public static void remAllQueue(Queue<Object> queue, Object item) {

	Queue<Object> newQueue = new LLQueue<Object>();
	while (! queue.isEmpty()) {
	    Object first = queue.peek();
	    if (! first.equals(item)) {
	        newQueue.insert(first);
	    }
	    queue.remove();
	}
	while (! newQueue.isEmpty()) {
	    Object newFirst = newQueue.peek();
	    queue.insert(newFirst);
	    newQueue.remove();
	}

  }
```

Problem 4: Binary tree basics

4-1) The height of this tree is **3**.

4-2) There are **four** leaf nodes and **five** interior nodes.

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

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

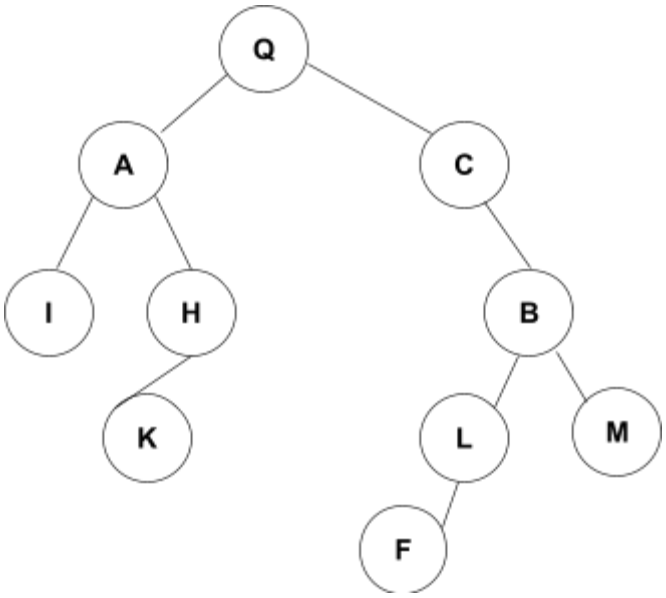
4-5) 21 18 27 7 25 30 19 26 35

4-6) This tree is not a binary search tree. To be a binary search tree, every node in the left subtree is smaller than the parent node, and every node in the right subtree is equal to or larger than the parent node. Node 25 is greater than 21, so in theory, it should go to the right subtree of node 21. However, it is in the left subtree of node 21, so this tree is not a binary search tree.

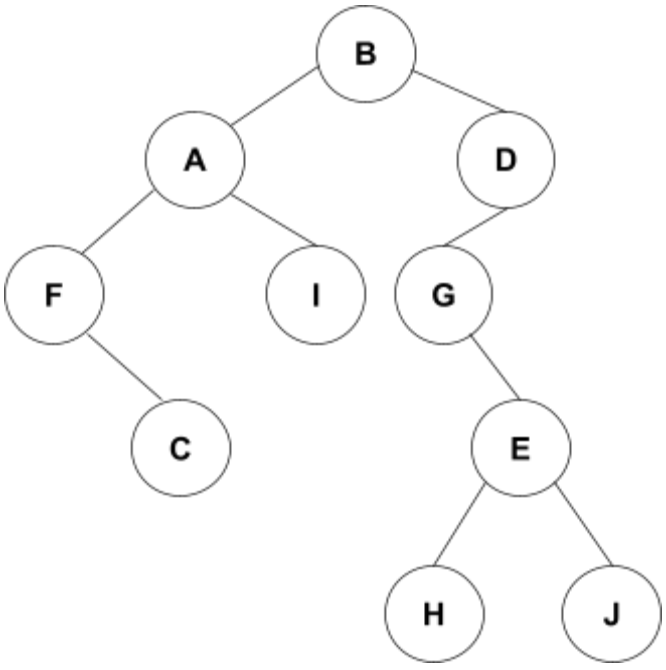
4-7) The tree is not balanced. To be balanced, any node in a tree must have a height difference between left and right subtrees less than or equal to 1. Node 27 does not have a left subtree (height = -1), and its right subtree has a height of 1. The height difference exceeds 1, so the whole tree is not balanced.

Problem 5: Tree traversal puzzles

5-1)

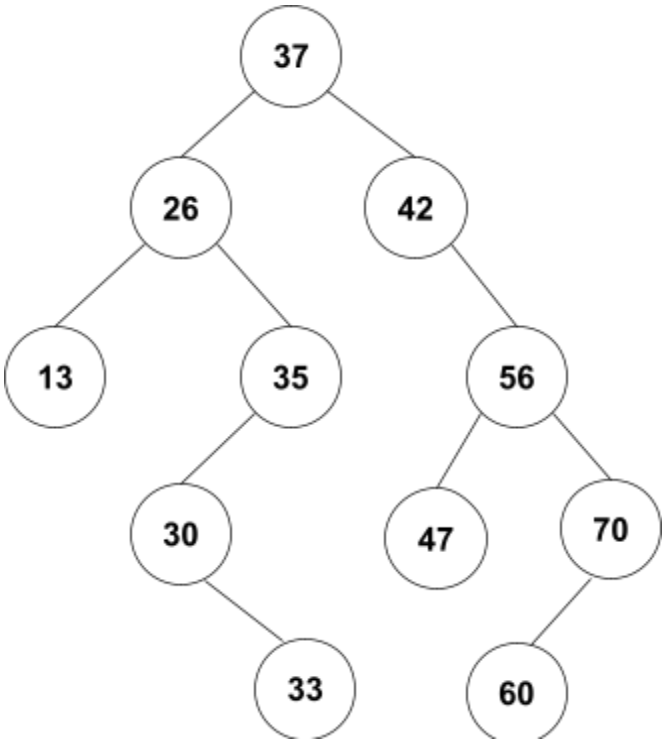


5-2)



Problem 6: Binary search trees

6-1)



6-2)