Problem Set 5, Part I

Problem 1: Choosing an appropriate representation

1-1) *ArrayList or LLList?* ArrayList

Explanation: Since the indices can be ordered by month, we want fast access to elements by index, and since the number of events is roughly the same each month, we can estimate the amount of memory we would require. This minimises the need for resizing and uses less memory overhead than the LLList because it does not require additional memory for storing pointers.

1-2) *ArrayList or LLList?* ArrayList

Explanation: We need to frequently access the runner's records and add the times during the race without a guaranteed array size and the insertion order. Given the static nature of the array after the sign-up deadline, an ArrayList is preferred as the primary operations are accessing and updating elements. The ArrayList provides O(1) time complexity for accessing elements by their index and minimises memory overhead due to its contiguous memory storage.

1-3) *ArrayList or LLList?* LLList

Explanation: LLList is ideal because it allows O(1) insertions at the beginning and the end, dynamically scaling the memory allocation for each node without resizing operations, fitting well with the need to add varying numbers of registrants as they sign up. Additionally, a doubly linked LLList enables efficient traversal from the most recently added element to the least recent.

Problem 2: Scaling a list of integers

2-1) $O(n^2)$. The outer loop runs n times assuming length() is a method of ArrayList that returns the size of the array operating at O(1) time. The getItem() method which accesses an element at index i, provides O(1) time complexity since it is an ArrayList. However, the addItem() method of LLList adds an item at specified position i, and traversing from the head node to the i-th node takes $O(n^2)$ time. Therefore, the overall time complexity of the scale method is $O(n^2)$.

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 running time of the improved algorithm is O(n). the addItem() operation on the LLList is now O(1) because it inserts the element at the head of the list in reverse order instead. Thus, given that both getItem() and addItem() operations are O(1), that is performed n times from the outer loop, the overall time complexity is now O(n).

Problem 3: Working with stacks and queues

3-1)

```
public static void remAllStack(Stack<Object> stack, Object item) {
      Stack<Object> tempStack = new Stack<>();

      while (!stack.isEmpty()) {
            Object current = stack.pop();
            if (!current.equals(item)) {
                  tempStack.push(current);
            }
      }

      while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
      }
}
```

3-2)

```
public static void remAllQueue(Queue<Object> queue, Object item) {
      Queue<Object> tempQueue = new LinkedList<>();

      while (!queue.isEmpty()) {
            Object current = queue.remove();
            if (!current.equals(item)) {
                  tempQueue.add(current);
            }
      }

      while (!tempQueue.isEmpty()) {
            queue.add(tempQueue.remove());
      }
}
```

Problem 4: Binary tree basics

4-1) 3

4-2) 4

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

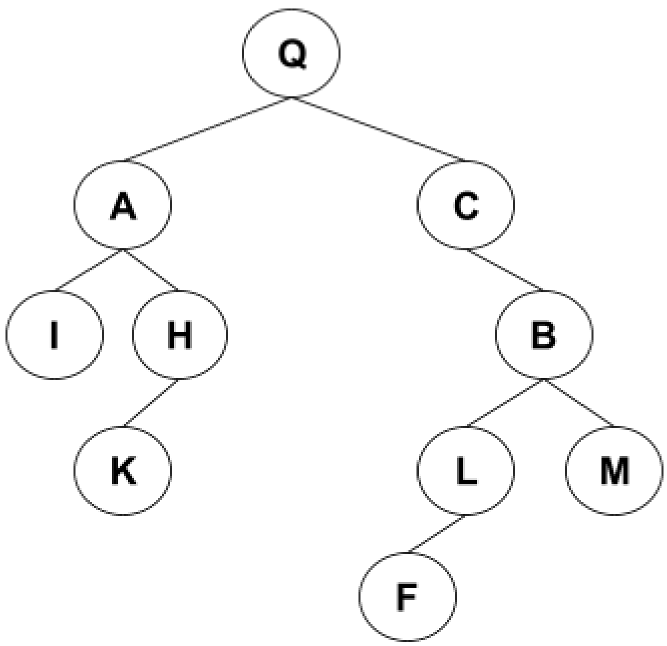
4-4) 7, 19, 25, 18, 26, 35, 30, 27, 21

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

4-6) Yes, because the left child is smaller than the root node and the right child is larger than the root node.

4-7) Yes, because the height difference of the two subtrees for each node is at most 1.

Problem 5: Tree traversal puzzles
5-1)



5-2)