Problem Set 7, Part I

Problem 1: Choosing an appropriate list implementation

1-1)
ArrayList is better for this gardening tools online store case. Since items are seldom added or removed from the list, we don't need to consider the space efficiency that used to resizing a list in this case. A product's position in the list is its product ID which is the index (ArrayList) or the address (LLList) in the list, and the products are the elements (ArrayList) or characters (LLList). When getting and modifying each element (ArrayList) or character (LLList) in the list, ArrayList has a better efficiency O(1) which is faster than the efficiency of LLList O(n). Therefore, ArrayList is better for this case.

1-2)
LLList is better for this tweets case. Since the number of tweets can vary widely from week to week in a way that is hard to predict, the space efficiency that used to resizing a list is an important factor in this case. For ArrayList, every time we resize the list, the length of the array would be double, so many empty spaces would be wasted if the length of the final list that we need is not to the any power of two, which is also the most of the time. For LLList, we can add one value and address each of the time, so there would be no empty spaces to be wasted. Also, in this case, we only need to read the list from the first to last in order to display the tweets in reverse chronological order, which gives the same time efficiency O(n) for both ArrayList and LLList. At the same time, we don't need to modify and get a selected tweet in the list, so we don't need to consider the time efficiency for this. Therefore, LLList is better for this case.

1-3)
Similar with the last question, the monthly list of events for an online arts calendar needs to add items and read the whole list in order. The only difference is that the resizing of the list is not needed in this case because the number of events is fairly consistent from month to month. Hence, the wasting of empty spaces when using ArrayList is not existed this time. Keep the other factors same as the last question, we got the same time efficiency and space efficiency for both ArrayList and LLList. Therefore, either ArrayList or LLList would be nice in this case.

Problem 2: Improving the Efficiency of an algorithm

getItem(): O(n) or O(m)
addItem(): O(n) or O(m)

2-1)
O(m, n) = m * (m + n^2)
Explain: Under the worse-case, the inner loop will run n times which equals to the length of list2, the getItem function will run n times, and the addItem function will be called only once if item1 = item2 in the inner loop, then break, so the whole inner loop will run n * (n + n) times; the outer loop will run m times which equals to the length of list1, and the getItem function will run m times, so the algorithm as a function will run m * (m + inner loop) = m * (m + (n * (n + n)) times.

2-2)

```
/*
 * intersect function that takes two unsorted lists, list1 and list2
 * (both instances of class LLList),
 * and creates a third list containing the intersection of list1 and list2
 */
public static LLList intersect(LLList list1, LLList list2) {
    LLList inters = new LLList();
    /*
    for (int i = 0; i < list1.length(); i++) {
        Object item1 = list1.getItem(i);
        for (int j = 0; j < list2.length(); j++) {
            Object item2 = list2.getItem(j);
            if (item2.equals(item1)) {
                inters.addItem(item2, inters.length());
                break;   // move onto the next item from list1
            }
        }
    }
    */
    ListIterator iterator1 = list1.iterator();
    ListIterator iterator2 = list2.iterator();
    while (iterator1.hasNext()) {
        Object item1 = iterator1.next();
        while (iterator2.hasNext()) {
            Object item2 = iterator1.next();
            if (item2.equals(item1)) {
                inters.addItem(item2, inters.length());
                break;   // move onto the next item from list1
            }
        }
    }

    return inters;
}
```

2-3)
O(m, n) = m * n^2
Explain: Under the worse-case, the inner loop will run n times which equals to the length of list2, and the addItem function will be called only once if item1 = item2 in the inner loop, then break, so the whole inner loop will run n + n times; the outer loop will run m times which equals to the length of list1, so the algorithm as a function will run m * (inner loop) = m * (n + n) times.

Problem 3: Working with stacks and queues

3-1)

```
/*
 * doubleAllStack method that takes a stack and an item, and that doubles
 * (i.e., adds an extra copy)
 * of all occurrences of that item in the stack.
 */
public static void doubleAllStack(Stack<Object> stack, Object item) {
    LLStack<Object> doubles = new LLStack<Object>();
    // push all items from stack to doubles, if is equal item, then push two times
    while (!stack.isEmpty()) {
        Object items = stack.pop();
        if (items == item) {
            doubles.push(items);
            doubles.push(items);
        } else {
            doubles.push(items);
        }
    }
    //push all items from doubles to stack
    while (!doubles.isEmpty()) {
        stack.push(doubles.pop());
    }
}
```

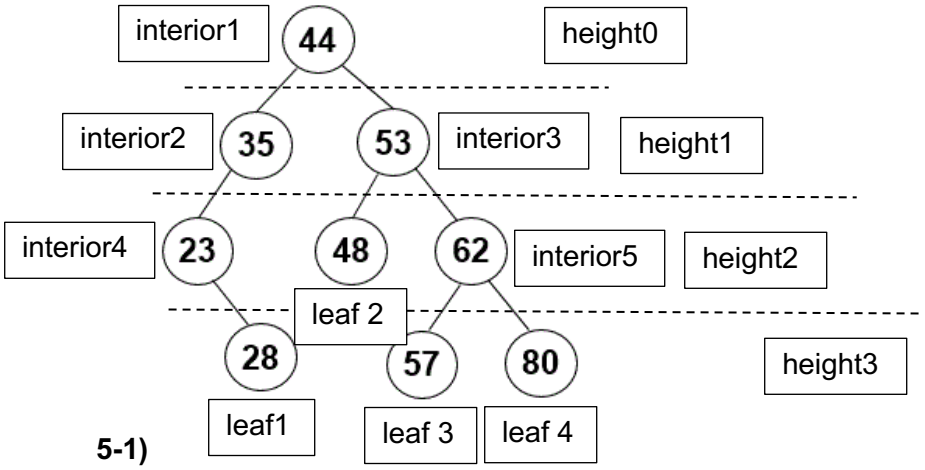
3-2)

```
/*
 * doubleAllQueue method that takes a stack and an item, and that doubles
 * (i.e., adds an extra copy)
 * of all occurrences of that item in the stack.
 */
public static void doubleAllQueue(Queue<Object> stack, Object item) {
    LLQueue<Object> doubles = new LLQueue<Object>();
    while (!stack.isEmpty()) {
        Object items = stack.remove();
        if (items == item) {
            doubles.insert(items);
            doubles.insert(items);
        } else {
            doubles.insert(items);
        }
    }
    while (!doubles.isEmpty()) {
        stack.insert(doubles.remove());
    }
}
```

Problem 4: Using a queue to search a stack

```
public static boolean search(Stack<Object> S, Object I) {
        Queue<Object> Q = new Queue<Object>();
        boolean result = false;
        // put items from S to Q (reverse order)
        while (!S.isEmpty()) {
            Q.insert(S.pop());
        }
        // search for I and put items from Q back to S (original order)
        while (!Q.isEmpty()) {
            Object item = Q.remove();
            if (item == I) {
                S.push(item);
                result = true;
            } else {
                S.push(item);
            }
        }
        return result;
    }
```

Problem 5: Binary tree basics



5-1)
3

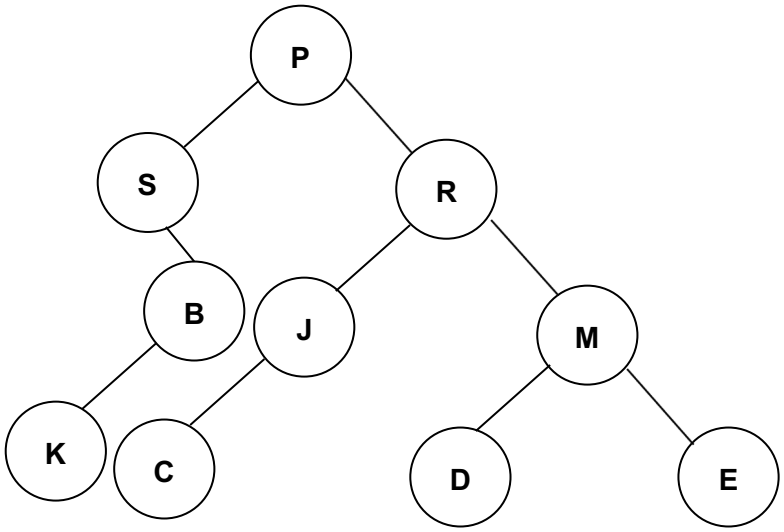
5-2)
4 leaf nodes:
28, 48, 57, 80
5 interior nodes:
44, 35, 53, 23, 62

5-3)
44 35 23 28 53 48 62 57 80

5-4)
28 23 35 48 57 80 62 53 44

5-5)
44 35 53 23 48 62 28 57 80

Problem 6: Tree traversal puzzles
6-1)



6-2)