Section 01: CSE 143 review, design decisions

1. CSE 143 review

1.1. Reference semantics

(a) What is the output of this program?

```
public class Mystery2 {
    public static void main(String[] args) {
        Point p = new Point(11, 22);
        System.out.println(p);

        int n = 5;
        mystery(p, n);
        System.out.println(p);

        p.x = p.y;
        mystery(p, n);
        System.out.println(p);

        Point p2 = new Point(100, 200);
        p = p2;
        mystery(p2, n);
        System.out.println(p + " :: " + p2);
    }

    public static void mystery(Point p, int n) {
        n = 0;
        p.x = p.x + 33;
        System.out.println(p.x + ", " + p.y + " " + n);
    }

    public static class Point {
        public int x;
        public int y;

        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public String toString() {
            return "(" + this.x + ", " + this.y + ")";
        }
    }
}
```

1.2. Being an implementor of ArrayLists

Note: in this section, you are acting as an *implementor* of the `MyArrayList` data structure. Assume you are adding to the following `MyArrayList` class with the following fields:

```
public class MyArrayList<T> {
    private T[] data;
    private int size;

    // constructors and other methods ommitted for space
}
```

For simplicity, assume the list does not contain any null items.

(a) Write a method `removeFront` that takes an integer `n` as a parameter and removes the first `n` values. For example, if a variable called `list` stores this sequence of values:

`[8, 17, 9, 24, 42, 3, 8]`

If we call `list.removeFront(4)`, the list should now store:

`[42, 3, 8]`

Assume that the parameter value passed is between 0 and the size of the list inclusive.

(b) Write a method `removeAll` that takes in an item of type `T` and removes all occurrences of that value from the list.

For example, if the variable named `list` stores the following values:

`["a", "b", "c", "d", "a", "d", "d", "e", "f", "d"]`

If we call `list.removeAll("d")`, the list should now store:

`["a", "b", "c", "a", "e", "f"]`

Assume you have previously implemented a method called `remove` that takes an index as a parameter and removes the value at the given index.

(c) Write a method `stretch` that takes an integer `n` as a parameter and that increases a list by a factor of `n` by taking each element in the original list and replacing it with `n` copies of that integer. For example, if a variable called `list` stores this squence of values:

`[18.2, 7.5, 4.2, 24.9]`

If we call `list.stretch(3)`, the list should now store:

`[18.2, 18.2, 18.2, 7.5, 7.5, 7.5, 4.2, 4.2, 4.2, 24.9, 24.9, 24.9]`

Make sure to implement the logic to resize the array as necessary.

1.3. Being an implementor of LinkedLists

Note: in this section, you are acting as an *implementor* of the `MyLinkedList` data structure. Assume you are adding to the following `MyLinkedList` class:

```
public class MyLinkedList<T> {
    private Node<T> front;
    private static class Node<T> {
        public final T data;
        public Node<T> next;

        // constructors omitted for space
    }
}
```

Do not create any new nodes (unless you are trying to implement `stretch`). The focus of these problems is to practice manipulating the links of list nodes. For simplicity, assume the list does not contain any null items.

(a) Try implementing the methods described in questions 1.2.a, 1.2.b, and 1.2.c for `MyLinkedList`.

(b) Write a method `switchPairs` that switches the order of each pair of elements: your method should switch the first two values, then the next two, and so forth. For example, if a variable called `list` stores this sequence of values:

`[3, 7, 4, 9, 8, 12]`

If we call `list.switchPairs()`, the list should now store:

`[7, 3, 9, 4, 12, 8]`

If there are an odd number of values, the final element is not moved.

(c) Write a method `reverse` that reverses the order of elements in a linked list. For example, if a variable called `list` stores this sequence of values:

`["a", "b", "c", "d", "e"]`

If we call `list.reverse()`, the list should now store:

`["e", "d", "c", "b", "a"]`

(d) Write a method `transferFrom` that accepts a second `MyLinkedList` as a parameter that moves values from the second list to this list (and empties the second list). For example, suppose the two lists store these sequences of values:

```
list1: [8, 17, 2, 4]
list2: [1, 2, 3]
```

The call of `list1.transferFrom(list2);` should leave the lists as follows:

```
list1: [8, 17, 2, 4, 1, 2, 3]
list2: []
```

Note that order matters: doing `list2.transferFrom(list1)` will NOT produce the same output as above. Either list may be empty. Assume, however, that the given list will not be null.

1.4. Being a client of Stacks and Queues

For the following problems, you will practice being the *client* of a data structure and two ADTs. Assume that you are given a class named `DoubleLinkedList` that implements both the Stack and Queue interfaces.

For simplicity, assume the stacks and queues you are given will never contain any null elements.

(a) Write a static method `copyStack` that accepts an `Stack<T>` as a parameter and returns a new stack containing exactly the same elements as the original. Your method should leave the original stack unchanged after the method is over. You may use one `Queue<T>` as auxiliary storage.

(b) Write a static method named `rearrange` that accepts an `Queue<Integer>` and rearranges the values so that all of the even values appear before the odd values and otherwise preserves the original order of the list. For example, suppose a queue called `q` stores this sequence of values:

```
front [3, 5, 4, 17, 6, 83, 1, 84, 16, 37] back
```

If we call `q.rearrange()`, the queue should now store:

```
front [4, 6, 84, 16, 3, 5, 17, 83, 1, 37] back
       evens          odds
```

Note that all of the evens are at the front, the odds are in the back, and that the order of the evens and the odds are the same as the original list. You may use one `Stack<Integer>` as auxiliary storage.

(c) Write a static method named `isPalindrome` that accepts a `Queue<T>` as a parameter and returns `true` if those values form a palindrome and `false` otherwise. For example, suppose a queue of ints called `q` stores this sequence of values:

```
front [3, 5, 4, 17, 6, 6, 17, 4, 5, 3] back
```

Then calling `isPalindrome` would return true, because the queue is exactly the same forwards and backwards (the definition of a palindromic object).

The queue must remain in its original state once the method is over. Assume that the empty queue is a palindrome. You may use one `Stack<T>` as auxiliary storage.

2. Design decisions: Selecting ADTs and data structures

For each of the following scenarios, choose:

(a) An ADT: `Stack` or `Queue`

(b) A data structure: `array list` or `linked list with front` or `linked list with front and back`

Justify your choice.

(a) You're designing a tool that checks code to verify all opening brackets, braces, parenthesis, etc... have closing counterparts.

(b) Disneyland has hired you to find a way to improve the processing efficiency of their long lines at attractions. There is no way to forecast how long the lines will be.

(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to look ahead to know what people have ordered and how many times to maximize efficiency in the kitchen.

3. Adapting ADTs and data structures

Choose appropriate ADTs, data structures, and algorithms to solve the following problems. You may use any ADT and data structure you can think of, including ones covered in CSE 143. Feel free to be creative!

For your reference, between CSE 143 and 373 we've covered the following ADTs: Lists, Stacks, Queues, Sets, Maps, PriorityQueues. We've also discussed the following data structures: array lists, linked lists, trees, hash maps.

(a) We want to call all the phone numbers with a particular area code in someone's phone book.

Describe how you would implement this. What is the time complexity of your solution? The space complexity?

(b) Long long ago, before smartphones were a thing, people who wanted to enter text using phones needed some way of entering arbitrary text using just 9 keys (the digits 1 through 9).

One such system is called "Text on nine keys" (T9). It associates 3 or 4 letters per each digit and lets you type words using just a single keypress per letter. To do this, it takes the sequence of digits entered and looks up all words corresponding to that sequence of keypresses within a fast-access dictionary of words and orders them by frequency of use.

For example, if the user types in '2665', the output could be the words `[book, cook, cool]`. Describe how you would implement a T9 dictionary for a mobile phone.

(For reference, the number '2' is associated with the letters 'abc', the number '3' is associated with 'def', etc... The number '9' is associated with 'wxyz'. The numbers '1' and '0' are used for other purposes.)

Describe how you would implement this. What is the time complexity of your solution? The space complexity?

(c) One refinement we could make to our T9 system is to train it so it "gains familiarity" with the words and phrases the current user likes to commonly used. So, if a particular user uses the word "cool" more frequently then the word "book", eventually the T9 system, given the input '2665', will learn to return `[cool, book, cook]` instead of `[book, cook, cool]`.

Describe how you would implement this. What is the time complexity of your solution? The space complexity?