Section 01: Solutions

Note: We have omitted solutions to all problems in part 1 because they are (slightly tweaked) variations of practice-it problems (`https://practiceit.cs.washington.edu/`).

If you would like to verify your answers to those problems, we recommend you (a) find the corresponding question on practice-it, (b) solve the (non-generic) version of the problem, then (c) modify your solution so it uses generics.

You can also try visiting office hours and ask one of the course staff to verify your answers for you.

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);

        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 int 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;

}
```

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:

```java
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 3, 4, and 5 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; throw an exception if the given list is 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:

$$\texttt{front [}\underbrace{\texttt{4, 6, 84, 16}}_{\text{evens}}\texttt{, }\underbrace{\texttt{3, 5, 17, 83, 1, 37}}_{\text{odds}}\texttt{] back}$$

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.

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

2.1. 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.

Solution:

We'd use the `Stack` ADT, because we want to match the *most recent* bracket we've seen first.

Since `Stacks` push and pop on the same end, there is no reason to use an implementation with two pointers. (We don't need access to the "back" ever.)

Asymptotically, there is no difference between the `LinkedList` with a `front` pointer and the `Array` implementation, but *cache locality* will likely be a problem with the `LinkedList`. (Arrays are contiguous in memory, but linked lists are stored using arbitrary pointers.)

(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.

Solution:

We'd use the `Queue` ADT here, because we're dealing with…a line.

The important thing to note here is that if we try to use the implementation of a `LinkedList` with *only a* `front` *pointer*, either *add* or *next* will be very slow. That is clearly not a good choice.

Arguably, the `LinkedList` implementation with both pointers is better than the array implementation because we will never have to resize it.

(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.

Solution:

This is still clearly the `Queue` ADT, but it's unclear that any of these implementations are a good choice!

One of the cool things about data structures is that if only one isn't good enough, you can use *two*. If we only care about the "normal queue features", then we would probably use the `LinkedList` implementation with one pointer. However, we can **ALSO** simultaneously use a *Map* to store the "number of times a food item appears in the queue".

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?

Solution:

One way to solve this would be using a `HashMap` where the keys are the area codes and the values is a list of corresponding phone numbers. We will need to parse the phone number to get the first three numbers.

Another way to solve this is by using a kind of data structure called a `Trie`. A trie is a kind of data structure based around the concept of a *tree*. However, unlike standard binary trees, tries have:

- A variable number of children (instead of just two)
- Have "labels" associated with each branch of the tree

Usually, we store the children of each node in a Map: the keys are the labels, the values are references to children node.

What we can do here is to make each node store exactly 10 children, labeling each branch from 0 to 9. We then take the first digit in the phone number and pick the branch with that label. From that node, we take the second digit and pick that branch.

If there are any nodes missing during this process, we create them as we go.

We repeat, using the entire phone number as the "route" to traverse the trie. Once we reach the end of the phone number, we stop, and set that node's data field to `true` to indicate that "route" corresponds to a valid phone number.

Then, to find all the phone numbers to call, we would use the area code to partially travel down the `Trie` then visit all children nodes to find the phone numbers to print.

If we compare these two approaches, both will have the same runtime efficiency, but the `Trie` will be more space-efficient in the average case.

If we let n be the total number of phone numbers and e be the expected number of phone numbers per area code, we can find that it takes $\Theta(n)$ time to build either the `HashMap` or the `Trie`. Likewise, given some area code, it takes $\Theta(e)$ time to visit and call each phone number.

(Initially, it may seem like the `Trie` would be slower due to the traversals. However, recall that the depth of the trie is always equal to the length of a phone number, which is a constant value.)

The reason why the `Trie` turns out to be more space-efficient on average is because the `Trie` is capable of storing near-duplicate phone numbers in less space then the `HashMap`. If we have the phone numbers `123-456-7890`, `123-456-7891`, and `123-456-7892`, the map must store each number individually whereas the `Trie` is able to combine them together and only branch for the very last number.

That said, in the absolute worst case where we try and insert every single possible 10-digit permutation of numbers into either data structure, both the `HashMap` and the `Trie` will end up taking up the same amount of space. However, this is an unlikely scenario, given how phone numbers are typically structured.

(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?

Solution:

One way to implement this would be by using a `Trie`. The routes (branches) are represented by the digits and the node's values are collection of words. So if you typed in 2, 6, 6, 5, you would choose the child representing 2, then 6, then 6, then 5, traveling four layers deep into the `Trie`.

Then, that child node's value would contain a collection of all dictionary words corresponding to this particular sequence of numbers.

To populate the `Trie`, you would iterate through each word in the dictionary, and first convert the word into the appropriate sequence of numbers.

Then, you would use that sequence as the key or "route" to traverse the `Trie` and add the word.

(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?

Solution:

One possible way of doing this would be to store, at each node, how frequently each word was accessed and use that information to sort the list of results.

To help provide reasonable results, we could provide "pre-trained" weights and adjust them over time as users use their phones more and more frequently.

The main issue we run into this solution is that eventually, our word counts will overflow (if we increment the counts very frequently). If our word pattern habits start changing, the T9 system will also be relatively slow to update, especially if some counts are very high and some are very low.

One way we could work around this problem is to keep track of the counts for just the past n messages or n days. This would solve the overflow and "slow update" problem – the challenge now would be to efficiently keep track of and manage this constantly changing data structure.