CSCI 1933 Lab 10

Midterm II Review

Rules

There is nothing to submit via Moodle for this lab.

You must work **alone** on this lab. In this lab, you will solve a set of problems in preparation for the second Midterm later this week. You **cannot** use your computer/tablets/smartwatches or the CSE machines to solve these problems. You must solve them on a piece of paper.

Lab Overview

As you all are hopefully already aware at this point, you have your second midterm this week. But if this is news, don't panic! This review lab is here to help you out The problems are sectioned off as follows:

1. Array List
2. Linked List
3. Queues
4. Stacks
5. Dictionaries
6. Complexity Analysis

You **do not** have to complete all of the problems in this lab to get it checked off. Note that this review contains substantially more problems than would be included on the midterm just to provide more opportunity for you to practice. You will get points for this lab if you either:

1. stay the entire lab and show your work to a TA
2. finish everything before lab starts/ends.

1 Array List

These problems will require you to recall our discussion of array lists in lecture and your implementation of one, called `AList`, in lab 8.

1.1 Binary Search, Binary, ary, ry, y I found the y!

Binary search is a pretty neat searching algorithm that runs in `O(log n)` time. The only problem is that it only works on a sorted collection, here's why. Let's say you are looking for 34 in the following list: $\langle -12, -11, -3, 5, 10, 11, 34, 64, 100, 1933\rangle$. Binary search looks at the element in the middle of the list, in this case 11. Since $34 > 11$, we can rule out the lower half of the list since it is sorted. Binary search then shifts its focus to the upper half: $\langle 34, 64, 100, 1933\rangle$. Again, looking at the middle element, $34 < 64$. Focusing on the lower half $\langle 34\rangle$ and looking at the element in the middle, we see that it is what we are looking for! Notice how we only had to do 3 comparisons to determine whether or not 34 was in our list. If we were to do a linear search, we'd have to make 7 comparisons! For this problem, implement a `boolean binarySearch(T element)` method in your `AList` class and explain why binary search runs in $log_2 n$ time. The method should return true if the element is in your list and false if not. You may assume that your elements are sorted in increasing order.

1.2 If I may intersect for a moment

The intersection of two collections A and B is defined to be the collection that contains all elements of A that also belong to B (and vice versa). For this problem, implement `public void intersect (AList<T> other)` method in your `AList` class. Instead of returning the intersection, it should modify the data in your list to be the result of the intersection.

> **Example:** Suppose `mixture` contains: $\langle 1, 2, 3, 4\rangle$ and `other` contains $\langle 2, 4, 6\rangle$. Then `mixture.intersect(other)` should modify `mixture` such that it only contains $\langle 2, 4\rangle$.

2 Linked List

These problems will require you to recall our `LinkedList` implementation we discussed in lecture and given in Lab 9. Remember that you will not have electronics by your side during the midterm, so it is best to have a good working memory of the key properties of our linked list implementation.

2.1 You get removed! You get removed! Every-two gets removed!

For this problem, implement `public void removeEvery(int n)` in your `LinkedList` class. This method should remove every n^{th} element from your list. You may assume that $n \geq 0$; if $n = 0$ or $n > size()$, do nothing.

> **Example:** Suppose we have the following list: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ and we call `removeEvery(2)`. We should end up with the following list: $A \rightarrow C \rightarrow E$.

2.2 Alright, break it up!

For this problem, implement `public LinkedList<LinkedList<T>> extractGroupsOf(int n)` in the `LinkedList` class from Lab 9. This method will return a list of lists, each of $size \leq n$. You may assume that $n \geq 0$; if $n = 0$ return an empty `LinkedList`.

> **Example:** Suppose we have the following list: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ and we call `extractGroupsOf(2)`. The following list is returned: $\langle A \rightarrow B,\ C \rightarrow D,\ E \rangle$.

3 Queues

3.1 A Stacked Queue

You know how you've seen a `Queue` being implemented using an `array` and `Nodes`? Well, now you have to implement it using a `Stack`. Create a `StackedQueue` class and implement the following methods: `enqueue(T element)`, `dequeue()`, `peek()`, `isEmpty()`. You may use Java's built in `Stack`.

> **Constraint:** If you are using an `array`-based `Stack`, you **cannot** touch the underlying `array`. Similarly, if you are using a `Node`-based `Stack`, you **cannot** touch the underlying `Nodes`

3.2 A Biased Queue

Budgers are terrible people, especially during Black Friday. Fortunately, we have a biased queue that prevents budgers from leaving the queue before non-budgers. For simplicity sake, our biased queue will be a queue of Strings. Each budger will be represented as the String `"Budger"` and each non-budger as some other String. Create a `BiasedQueue` class and implement the following methods:

- `public void enqueue(String person)` – This will put the person into the queue. If the person is a budger, allow the person to budge the person at the end of the queue.

  > **Example:** Suppose this is the state of our queue: `Andy, Leslie, Ron, Tom` and we call `enqueue("Budger")`. Then the queue becomes: `Andy, Leslie, Ron, Budger, Tom`.

- `public String dequeue()` – This will remove the person at the front of the line. If the person at the front of the line is a budger, then you should remove the first non-budger in line. If there are only budgers in line, then remove the first budger.

  > **Example:** Suppose this is our queue: `Budger, Budger, Andy, Leslie, Ron`
  > Suppose we call `dequeue()`. The queue becomes: `Budger, Budger, Leslie, Ron`

  > **Example:** Suppose this our queue: `Budger, Budger` and we call `dequeue()`. Then the queue becomes: `Budger`.

You may use **any** data-structure you've learned so far as the underlying data structure for your `BiasedQueue`.

4 Applying Stacks - Postfix Evaluation

In this section, you will be implementing a postfix evaluation algorithm using a stack. You can assume the standard stack interface we discussed in lecture. The following is an example of how you would declare a stack of `Integers`:

Example: `Stack<Integer> nums = new Stack<>();`

Keep in mind that `T` below is the type of data going into your stack – here are some stack methods that will be beneficial to you:

- `T push(T element)` – pushes an element onto the stack and returns it
- `T pop()` – removes the element from the top of the stack and returns it.
- `T peek()` – returns the element at the top of the stack.
- `boolean isEmpty()` – returns true if the stack is empty and false otherwise
- `void clear()` – clears the stack

Create a `public class Postfix` class with the following method:

- `static double evaluate(String[] expression)` – given an expression in postfix notation, evaluate it and return the resulting number.

Here are some examples of infix expressions followed by their postfix equivalents:

```
5 + 2            =  {"5", "2", "+"}
1 - 2            =  {"1", "2", "-"}
3 + (4 * 5)      =  {"4", "5", "*", "3", "+"}
(1 + 2)/ (3 + 4) =  {"1", "2", "+", "3", "4", "+", "/"}
```

Some simplifying assumptions

- You do not have to worry about parentheses.
- You do not have to worry about dividing by zero. However, think about an appropriate course of action to take should there be one.
- You only have to worry about the four arithmetic operators: $+, -, *, /$
- Your method must account for negatives and decimal numbers (e.g. $\mathbf{-3.14}$)

5 Dictionaries

You have learned in class that dictionaries are an abstract data type. Dictionaries (also called Maps) are implemented in Java as a HashMap and they contain entries that have two parts:

- A search key
- A value associated with the key

Keeping this in mind, write a program to find to check if two strings are anagrams of each other. For example: `"Listen"` and `"Silent"` are anagrams because the letters in `"Listen"` can be rearranged to get `"Silent"`. For this problem, you may use Java's built-in hash map. The following is an example of how to declare a hash map with `Integer`s as keys and `BankAccount`s as values:

Example: `Map<Integer, BankAccount> idToAccount = new HashMap<>();`

The `Map` interface provides the following methods that may be beneficial to you. Note that `K` and `V` below represent the types of your keys and values. In the example above, our keys are of type `Integer` and our values are of type `BankAccount`s

- `V put(K key, V value)` – This associates the given value with the given key. If the key is already in the map, then its old value will be replaced and returned.
- `boolean contains(K key)` – This determines whether or not the given key is in the map.
- `V get(K key)` – If the key is in the map, then it returns its associated value. If the key is not in the map, then it returns `null`

6 Complexity Analysis

6.1 Sorting Basics

Complete the following table (assume the data to be sorted is stored in an array):

Runtime Complexity	Best-Case	Average-Case	Worst-Case
Selection-Sort			
Merge-Sort			
Quick-Sort			

Provide explanations for your answers.

6.2 Big-O

Consider the following snippet of code, what is its time complexity in terms of Big-O? Explain

```
//checks int array for subsequences of k matching ints
public static int containsMatch(int[] array, int k) {
    int countMatches = 0;
    for (int i = 0; i < array.length - (k - 1); i++) {
        boolean containsMatch = true;
        for (int j = i + 1; j < i + k; j++) {
            if (array[i] != array[j]) {
                containsMatch = false;
            }
        }
        if (containsMatch) {
            countMatches++;
        }
    }
    return countMatches;
}
```

6.3 More Big-O

What is the worst-case time complexity for the method `add(T element)` in the following Java code? Explain your answer using the Big-O notation.

```
public class ArrayList<T extends Comparable<T>> implements List<T> {
    private T[] list;
    private int size;

    public ArrayList() {
        list = (T[]) new Comparable[2];
        size = 0;
    }

    private void doubleListLength() {
        T[] newList = (T[]) new Comparable[list.length * 2];
        for (int i = 0; i < size; i++) {
          newList[i] = list[i];
        }
        list = newList;
    }

    public boolean add(T element) {
        if (element == null) {
          return false;
        }
        if (size == list.length) {
          doubleListLength();
        }
        list[size] = element;
        size++;
        return true;
    }
}
```

6.4 Trade-Offs

When might you want to use an `ArrayList` over a `LinkedList`? What about a `LinkedList` over an `ArrayList`?