Topic: Queue Interface
Ref: CSC20 PAL Stacks

The stack and queue are some of the simplest collections in Java and are important and widely used data structures. Be sure to go over the worksheet on Stacks before you start this one. Let's go over some of the basic design and operations of a queue first.

In contrast to a stack, a queue is called a called 'First In First Out' data structure because the item that gets added in first stays at the front of the queue and is the first to be removed. The analogy of a queue is an actual queue at a restaurant which serves customers on a first-come, first-serve basis. The customers who entered the queue first also leave the queue first.

What's an application where a queue is preferable to a stack? A database which collects customer queries – you want to address the queries which came in first, because they've been waiting the longest.

Let's see how queue methods work. Note that in Java, Queue is an interface, and thus needs a concrete class when instantiating an object – we can use LinkedList class as below.

```
Queue<Integer> myQue = new LinkedList<Integer>();

//add items to the queue
myQue.add(1);
myQue.add(3);
myQue.add(5);

// retrieve most recently added item
Integer firstItem = myQue.remove() // this has value 1

// peek at first item without removing it
Integer nextItem = myQue.peek() // this has value 3
```

Now recall the palindrome problem that was solved in the Stacks worksheet in which given an input string, uses a stack to determine if it is a palindrome, i.e. is the same even when read in reverse. An example would be 'racecar'.

Q1) Solve the palindrome problem with a queue instead of a stack.

Design-wise, which do you think is more suitable for the palindrome problem?

Let's now move onto using queues for common operations like search, insertion, sort, and reversal.

Search

Searching a queue means using *remove()* to remove the top/front element repeatedly so we can examine all the items and compare to our search value. However, this erodes the collection and may leave it empty. We need to be mindful of restoring the queue to its original state after the search.

Restoration looks different for each collection. Given a queue like [2,4,6,8], a good strategy is to simply *add()* back each removed element, which sends it to the back of the list.

If we want to search for elements, we can simply check each removed element, before it's sent to the back of the queue. What do you need to keep in mind while implementing the solution if we are changing the queue size as we traverse it?

Q2) Write a method *searchQueue* which takes in an Integer queue and an Integer value, then verifies if the value exists in the queue or not. It returns *boolean* true if value is found and false otherwise

Insertion

Insertion is similar to search in that a queue can uniquely keep rotating through its elements, moving them to the back while searching, while a stack would likely require another stack to help it stored removed elements before returning them.

Try doing a queue insertion problem, following this theme.

Q3) Write a method *insertQueue* which takes in an Integer queue, an Integer value, and an integer position and then inserts the value into the queue at the given position. You are not allowed to use any auxiliary storage. You may presume that the position is between 1 and the size of the queue. This method returns void. E.g. If the queue *q* was originally [2,5,1,0] then a call to *insertQueue*(q,1,4) would change it to [2,5,1,0,1]

Q4) Write a method which takes in an ordered Integer queue and an Integer value, then inserts the value into the queue while preserving its ascending order. This method returns void.

Reversal

Q4) Write a method which takes in an Integer queue and returns it with its elements in reverse order. Think about how using a stack can help you accomplish this.

Q5) Solve Practice-it Exercises Chapter 14 Ex14.14 (*reverseFirstK)* and Ex14.19 (*RemoveMin*)